Volume 1

MICROSOFT®

OS/2

Programmer's Reference

Including Presentation Manager

Microsoft

OS|2

PROGRAMMER'S
REFERENCE
LIBRARY

1

Microsoft® Operating System/2 Programmer's Reference

Version 1.1

Written, edited, and produced
by Microsoft Corporation
Distributed by Microsoft Press

Microsoft OS|2

PROGRAMMER'S REFERENCE LIBRARY

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
16011 NE 36th Way, Box 97017, Redmond, Washington 98073-9717

Library of Congress Cataloging in Publication Data
Microsoft OS/2 programmer's reference.
Includes index.
1. Microsoft OS/2 (Computer operating system) I. Microsoft Press
QA76.76.063078 1989 005.4'469 89-2817
ISBN 1-55615-220-5(Vol. 1)

Printed and bound in the United States of America.
1 2 3 4 5 6 7 8 9 FGFG 3 2 1 0 9

Distributed to the book trade in the United States by Harper & Row.
Distributed to the book trade in Canada by General Publishing Company, Ltd.
Distributed to the book trade outside the United States and Canada
by Penguin Books Ltd.
Penguin Books Ltd., Harmondsworth, Middlesex, England
Penguin Books Australia Ltd., Ringwood, Victoria, Australia
Penguin Books N.Z. Ltd., 182-190 Wairau Road, Auckland 10, New Zealand

Writers: Brad Hastings
Stan Krute
Donn Morse
Ralph Walden
Dan Weston

The character-set tables in this manual are reprinted by permission from the *IBM Operating System/2 User's Reference*, © 1987 by International Business Machines Corporation.

Document No. LN0702A-110-R00-0289

Contents

Part 1 Introducing MS OS/2 3

Chapter 1 Introduction

1.1 Overview 7
1.2 About the MS OS/2 Programmer's Reference 7
1.3 How to Use This Manual 8
1.4 MS OS/2 and the C Programming Language 8
1.5 MS OS/2 Naming Conventions 9
1.6 Notational Conventions 11

Chapter 2 MS OS/2 Overview

2.1 Introduction 15
2.2 MS OS/2 and Presentation Manager 15
2.3 The Window Manager 17
2.4 The Graphics Programming Interface 19
2.5 System Services 22
2.6 The MS OS/2 System Functions 25

Chapter 3 MS OS/2 Programming Models

3.1 Introduction 29
3.2 Full-Screen Programs 29
3.3 Presentation Manager Applications 30
3.4 The Family Application Programming Interface 32
3.5 Using the Command Line 37
3.6 Using Structures 38
3.7 Using Bit Masks 39
3.8 Sharing Resources 40
3.9 C-Language Header Files 40

Part 2 Window Manager 47

Chapter 4 Windows

4.1 Introduction 51
4.2 About Windows 51
4.3 System-Modal Windows 58
4.4 Using Windows 66
4.5 Summary 72

Chapter 5 Messages and Message Queues

5.1 Introduction 79
5.2 About Messages and Message Queues 79
5.3 Using Messages in an Application 86
5.4 Summary 89

Chapter 6 Window Classes

6.1 Introduction 93
6.2 About Window Classes 93
6.3 Using Window Classes 98
6.4 Summary 98

Chapter 7 Window Procedures

7.1 Introduction 101
7.2 About Window Procedures 101
7.3 Using a Window Procedure 103
7.4 Summary 105

Chapter 8 Mouse and Keyboard Input

8.1 Introduction 111
8.2 About Mouse and Keyboard Input 111
8.3 Using the Mouse and Keyboard in an Application 112
8.4 Summary 120

Chapter 9 Frame Windows

9.1 Introduction 127
9.2 About Frame Windows 127
9.3 Using Frame Windows 139
9.4 Summary 141

Chapter 10 Control Windows

10.1 Introduction 145
10.2 About Control Windows 145
10.3 Using Control Windows in an Application 146
10.4 Creating a Custom Control Window 146
10.5 Summary 147

Chapter 11 Title-Bar Controls

11.1 Introduction 153
11.2 About Title Bars 153
11.3 Using Title-Bar Controls in Applications 153
11.4 Default Title-Bar Behavior 154
11.5 Summary 155

Chapter 12 Button Controls

12.1 Introduction 159
12.2 About Button Controls 159
12.3 Using Button Controls in an Application 160
12.4 Default Button Behavior 164
12.5 Summary 165

Chapter 13 Entry-Field Controls

13.1 Introduction 171
13.2 About Entry-Field Controls 171
13.3 Using Entry-Field Controls in an Application 171
13.4 Default Entry-Field Behavior 174
13.5 Summary 176

Chapter 14 List-Box Controls

14.1 Introduction 181
14.2 About List Boxes 181
14.3 Using a List Box in an Application 181
14.4 Default List-Box Behavior 187
14.5 Summary 189

Chapter 15 Static Controls

15.1 Introduction 195
15.2 About Static Controls 195
15.3 Using Static Controls in an Application 195
15.4 Default Static-Control Behavior 197
15.5 Summary 198

Chapter 16 Scroll-Bar Controls

16.1 Introduction 203
16.2 About Scroll Bars 203
16.3 Using Scroll Bars 207
16.4 Summary 210

Chapter 17 Menus

17.1 Introduction 215
17.2 About Menus 215
17.3 Defining Menu Items in a Resource File 217
17.4 Menu Data Structures 219
17.5 Using Menus in your Applications 221
17.6 Summary 232

Chapter 18 Accelerator Tables

18.1 Introduction 239
18.2 About Accelerator Tables 239
18.3 Accelerator Tables in a Resource-Definition File 239
18.4 Accelerator-Table Data Structures 240
18.5 Using an Accelerator Table in an Application 241
18.6 Summary 242

Chapter 19 Dialog Windows

19.1 Introduction 247
19.2 About Dialog Windows 247
19.3 Dialog Data Structures 249
19.4 Dialog Resources 250
19.5 Using Message and Dialog Boxes 251
19.6 Summary 258

Chapter 20 Painting and Drawing

20.1 Introduction 265
20.2 About Painting and Drawing 265
20.3 Strategies for Painting and Drawing 268
20.4 Printing 275
20.5 Summary 275

Chapter 21 Drawing in Windows

21.1 Introduction 281
21.2 Window-Drawing Functions 281
21.3 Using Window-Drawing Functions 282
21.4 Summary 285

Chapter 22 Mouse Pointers and Icons

22.1 Introduction 289
22.2 About Mouse Pointers 289
22.3 Using a Mouse Pointer in an Application 290
22.4 Summary 291

Chapter 23 Cursors

23.1 Introduction 297
23.2 About Cursors 297
23.3 Using Cursors in an Application 297
23.4 Summary 298

Chapter 24 Printing

24.1 Introduction 303
24.2 About Printing 303
24.3 Printing 305
24.4 Special Printing Topics 313
24.5 Summary 317

Chapter 25 Heaps

25.1 Introduction 321
25.2 About Heaps 321
25.3 Using a Heap in an Application 322
25.4 Summary 330

Chapter 26 Clipboard

26.1 Introduction 333
26.2 About the Clipboard 333
26.3 Using the Clipboard 335
26.4 Summary 342

Chapter 27 Dynamic Data Exchange

27.1 Introduction 347
27.2 About Dynamic Data Exchange 347
27.3 Using Dynamic Data Exchange 349
27.4 Summary 360

Chapter 28 Hooks

28.1 Introduction 365
28.2 About Hooks 365
28.3 Types of Hooks 365
28.4 Using Hooks 372
28.5 Hook Example 373
28.6 Summary 374

Chapter 29 Help

29.1 Introduction 379
29.2 About Help 379
29.3 Using Help in an Application 381
29.4 Summary 386

Part 3 Graphics Programming Interface 389

Chapter 30 Presentation Spaces and Device Contexts

30.1 Introduction 393
30.2 About Presentation Spaces and Device Contexts 393
30.3 Using Presentation Spaces and Device Contexts 399
30.4 Summary 400

Chapter 31 Coordinate Spaces and Transformations

31.1 Introduction 405
31.2 About Coordinate Spaces and Transformations 405
31.3 Using Coordinate Spaces and Transformations 421
31.4 Summary 423

Chapter 32 Line and Arc Primitives

32.1 Introduction 427
32.2 About Line and Arc Primitives 427
32.3 Using Line and Arc Primitives 432
32.4 Summary 436

Chapter 33 Fonts and Character Primitives

33.1 Introduction 441
33.2 About Fonts and Character Primitives 441
33.3 Using Fonts and Character Primitives 459
33.4 Summary 462

Chapter 34 Color and Mix Modes

34.1 Introduction 469
34.2 About Color and Mix Modes 469
34.3 Using Colors and Mix Modes 474
34.4 Summary 476

Chapter 35 Paths

35.1 Introduction 481
35.2 About Paths 481
35.3 Using Paths 490
35.4 Summary 492

Chapter 36 Area Primitives

36.1 Introduction 495
36.2 About Areas and Area Primitives 495
36.3 Using Areas and Area Primitives 501
36.4 Summary 504

Chapter 37 Marker Primitives

37.1 Introduction 507
37.2 About Marker Primitives 507
37.3 Using Markers 509
37.4 Summary 510

Chapter 38 Bitmaps

38.1 Introduction 513
38.2 About Bitmaps 513
38.3 Using Bitmaps 520
38.4 Summary 524

Chapter 39 Regions

39.1 Introduction 529
39.2 About Regions 529
39.3 Using Regions 533
39.4 Summary 537

Chapter 40 Clipping

40.1 Introduction 541
40.2 About Clipping 541
40.3 Using Clipping 543
40.4 Summary 547

Chapter 41 Metafiles

41.1 Introduction 551
41.2 About Metafiles 551
41.3 Using Metafiles 554
41.4 Summary 557

Chapter 42 Segments and Retained Graphics

42.1 Introduction 563
42.2 About Segments and Retained Graphics 563
42.3 Using Segments and Retained Drawings 570
42.4 Summary 573

Part 4 System Services 583

Chapter 43 Processes, Threads, and Sessions

43.1 Introduction 587
43.2 About Processes, Threads, and Sessions 587
43.3 Using Processes 593
43.4 Using Threads 597
43.5 Summary 599

Chapter 44 The Memory Manager

44.1 Introduction 603
44.2 About the Memory Manager 603
44.3 Using the Memory Manager 604
44.4 Summary 611

Chapter 45 Dynamic Linking

45.1 Introduction 615
45.2 About Dynamic Linking 615
45.3 Building Dynamic-Link Libraries 616
45.4 Using Dynamic Linking 620
45.5 Summary 620

Chapter 46 The File System

46.1 Introduction 623
46.2 About the File System 623
46.3 Using the File System 626
46.4 Summary 634

Chapter 47 Video Input and Output

47.1 Introduction 639
47.2 About Video Input and Output 639
47.3 Using Video Input and Output 640
47.4 Summary 644

Chapter 48 Advanced Video Input and Output

48.1 Introduction 649
48.2 About Advanced Video Input and Output 649
48.3 Using Advanced Video Input and Output 653
48.4 Summary 654

Chapter 49 The Mouse

49.1 Introduction 661
49.2 About the Mouse 661
49.3 Using the Mouse 662
49.4 Summary 665

Chapter 50 The Keyboard

50.1 Introduction 669
50.2 About the Keyboard 669
50.3 Using the Keyboard 670
50.4 Summary 674

Chapter 51 Interprocess Communication

51.1 Introduction 677
51.2 About Interprocess Communication 677
51.3 Using Interprocess Communication 683
51.4 Summary 689

Chapter 52 Timers

52.1 Introduction 695
52.2 About Timers 695
52.3 Using Timers 695
52.4 Summary 696

Chapter 53 Window Timers

53.1 Introduction 699
53.2 About Window Timers 699
53.3 Using Window Timers 700
53.4 Summary 701

Chapter 54 Device Monitors

54.1 Introduction 705
54.2 About Device Monitors 705
54.3 Using Device Monitors 705
54.4 Summary 705

Chapter 55 Atom Tables

55.1 Introduction 709
55.2 About Atom Tables 709
55.3 Using Atom Tables in an Application 710
55.4 Summary 713

Chapter 56 System Information

56.1 Introduction 717
56.2 About System Information 717
56.3 Using System Information 724
56.4 Summary 729

Index 733

Figures

Figure 2.1 A Typical Presentation Manager Window 17
Figure 11.1 Frame Window with Title Bar 153
Figure 12.1 Button Types 159
Figure 14.1 Typical List-Box Control 181
Figure 17.1 Menu-Bar and Pull-Down Menus 216
Figure 17.2 Submenus 218
Figure 19.1 Sample Message Box 249
Figure 19.2 Sample Dialog Box 251
Figure 20.1 Application-to-Device Path 265
Figure 20.2 Clip Region and Visible Region 267
Figure 20.3 Update Region and Visible Region 267
Figure 20.4 Cached-Micro Presentation Space 271
Figure 20.5 Micro Presentation Space 273
Figure 20.6 Normal Presentation Space 274
Figure 21.1 Rectangle Types 282
Figure 22.1 Bit Values in the AND and XOR Masks 289
Figure 24.1 Application to Device Path 303
Figure 24.2 World-Space to Device-Space Translation 310
Figure 24.3 Portrait-Mode Pages in World Space 311
Figure 24.4 Landscape-Mode Pages in World Space 311
Figure 24.5 IBM4201 Page Setup 313
Figure 25.1 Heap in Automatic Data Segment 321
Figure 25.2 Back Pointer for Moveable Heap Object in Automatic Data Segment 326
Figure 25.3 Back Pointer for Moveable Heap Object in Separate Data Segment 327
Figure 27.1 Typical DDE Segment 351
Figure 29.1 Help Menu Item 380
Figure 31.1 Device Coordinate System 406
Figure 31.2 Video Display and Single-Coordinate-Space System... 406
Figure 31.3 Diagonal Line in a Single-Coordinate-Space System... 407
Figure 31.4 One Rectangle Displayed on Two Different Devices .. 407
Figure 31.5 Fixed Value 410
Figure 31.6 Rotating an Object 411
Figure 31.7 Subpictures in Four Chained Segments in World Space 413
Figure 31.8 Two Model Spaces 414
Figure 31.9 Viewing Limit in Model Space 414

Figure 31.10 Retained Subpicture Drawn Using Model Transformations 415
Figure 31.11 Clip Path in World Space 416
Figure 31.12 Scaling the Clipped Part of a Subpicture in Model Space 417
Figure 31.13 Presentation Page in Page Space 419
Figure 31.14 Page Viewport in Device Space 419
Figure 31.15 Determining Scaling Factors 420
Figure 31.16 Translating the Page Viewport in Device Space 420
Figure 32.1 Sample Illustrations Using Line and Arc Functions ... 427
Figure 32.2 Arcs 429
Figure 32.3 Transforming the Unit Circle Using the Arc Parameters 430
Figure 32.4 Line Styles 431
Figure 33.1 Strokes 441
Figure 33.2 Serifs 441
Figure 33.3 Bold Font 442
Figure 33.4 Italic Font 442
Figure 33.5 Normal Font 442
Figure 33.6 Character Cell 443
Figure 33.7 Em Height 443
Figure 33.8 X Height 443
Figure 33.9 Maximum Ascender 444
Figure 33.10 Maximum Descender 444
Figure 33.11 Lowercase Ascent 444
Figure 33.12 Lowercase Descent 445
Figure 33.13 Internal Leading 445
Figure 33.14 External Leading 445
Figure 33.15 Em Increment 446
Figure 33.16 Maximum Baseline Extent 446
Figure 33.17 Character Slope 446
Figure 33.18 In-line Direction 447
Figure 33.19 Character Rotation 447
Figure 33.20 Superscript 448
Figure 33.21 Subscript 448
Figure 33.22 Image Font 450
Figure 33.23 Outline Font 450
Figure 33.24 Character-Cell Alignment 450
Figure 33.25 Proportional and Fixed Fonts 450
Figure 33.26 Code Page 437 451
Figure 33.27 Character Shear 453
Figure 33.28 Image Font and Character Shear 453
Figure 33.29 Image Font and Character Angle 454

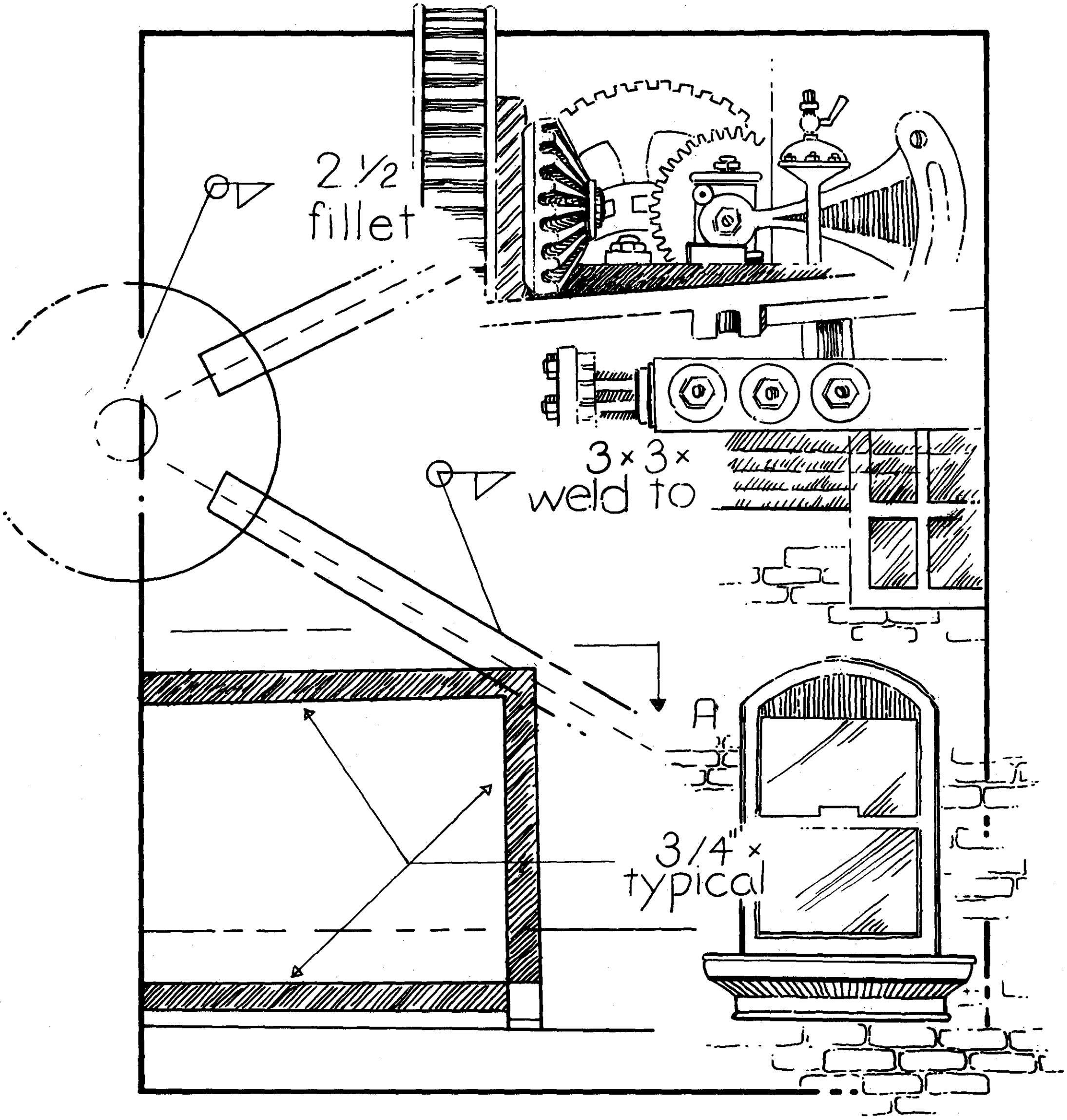
Figure 33.30 Image Font and Three Sizes of Character Box 454
Figure 33.31 Outline Font and Character Shear 455
Figure 33.32 Outline Font and Character Angle 455
Figure 33.33 Outline Font and Three Sizes of Character Box 455
Figure 33.34 Graph with Outline Font 456
Figure 35.1 Stroked and Filled Rectangles 481
Figure 35.2 Geometric Lines and Normal Lines 484
Figure 35.3 Stroked Paths 484
Figure 35.4 Filled Paths 485
Figure 35.5 Fill Modes 485
Figure 35.6 Alternate-Mode Test 486
Figure 35.7 Winding-Mode Test 486
Figure 35.8 Triangular Clip Path 487
Figure 35.9 Geometric Line Ends 488
Figure 35.10 Geometric Line Joins 488
Figure 36.1 Two Areas 495
Figure 36.2 Disjoint Area 495
Figure 36.3 Predefined Fill Patterns 500
Figure 37.1 Default Marker Set 507
Figure 37.2 Markers Used in a Graph 508
Figure 38.1 Bitmap and Image 513
Figure 38.2 Bitmap Shown on Two Displays with Different Aspect Ratios 514
Figure 38.3 Bits and Pels in a Bitmapped Image 514
Figure 38.4 A Bitcount and its Associated Color Table 516
Figure 38.5 Bits and Pels in a Special Bitmapped Image 519
Figure 39.1 Disjoint Region 529
Figure 39.2 Region of Two Intersecting Rectangles 530
Figure 39.3 Text in Client Area 531
Figure 39.4 Disjoint Clip Region 531
Figure 39.5 Combining Regions 532
Figure 39.6 Three Square Regions and Three Fill Patterns 532
Figure 40.1 Text in an Elliptical Clipping Area 541
Figure 40.2 Valid Clipping Areas 542
Figure 40.3 Clipping in Four Coordinate Spaces 543
Figure 42.1 Combining Subpictures to Create a Floor Plan 564
Figure 42.2 Chained and Called Segments 565
Figure 42.3 Correlation Operation 568
Figure 42.4 Inserting a New Element in a Segment 569
Figure 42.5 Replacing an Element with a New Element 570
Figure 48.1 2-Byte Character Format 650
Figure 48.2 4-Byte Character Format 651

Tables

Table 30.1 Presentation-Space Features and Restrictions 393
Table 38.1 Drawing Modes and Bitmapped Output..................... 518
Table 38.2 GpiBitBlt Output 524

Part 1

Introducing MS OS/2



Part 1

Introducing MS OS/2

Chapter 1 Introduction 5

Chapter 2 MS OS/2 Overview 13

Chapter 3 MS OS/2 Programming Models 27

Chapter

1

Introduction

1.1 Overview 7
1.2 About the MS OS/2 Programmer's Reference 7
1.3 How to Use This Manual.. 8
1.4 MS OS/2 and the C Programming Language.................. 8
1.5 MS OS/2 Naming Conventions 9
1.6 Notational Conventions.. 11

1.1 Overview

This manual describes the Microsoft® Operating System/2 (MS® OS/2) system functions. MS OS/2 is a single-user, multitasking operating system for personal computers. MS OS/2 system functions let programs use the operating system to carry out tasks such as reading from and writing to disk files, allocating memory, and starting other programs.

Part 1, "Introducing MS OS/2," introduces the MS OS/2 system functions. It provides a brief description of the *Microsoft Operating System/2 Programmer's Reference*, describes the role of the C programming language in the *Programmer's Reference*, and gives the calling and notational conventions used in this manual. The chapters in this part provide a general overview of the MS OS/2 system functions and describe the three MS OS/2 programming models.

Part 2, "Window Manager," describes the portion of MS OS/2 that lets applications create and manage windows. The chapters in this part provide detailed information about windows, messages, message queues, control windows, dialog windows, and other window-management topics.

Part 3, "Graphics Programming Interface," describes the portion of MS OS/2 that lets applications use device-independent graphics. The chapters in this part provide detailed information about presentation spaces, transformations, device contexts, graphics primitives, retained graphics, metafiles, and other graphics-related topics.

Part 4, "System Services," describes the portion of MS OS/2 that lets applications use the basic multitasking services of MS OS/2. The chapters in this part provide detailed information about processes and threads, memory management, the file system, dynamic linking, keyboard and mouse input, video output, device control, and other information about the system.

This manual is intended to describe the purpose of the MS OS/2 system functions and to explain the operating-system concepts behind them. It also shows how the MS OS/2 system functions work together to carry out specific tasks. It does not show how to write, compile, and link programs containing these functions. For more information on these topics, see *Microsoft Operating System/2 Programming Tools*.

1.2 About the MS OS/2 Programmer's Reference

The *Microsoft Operating System/2 Programmer's Reference*, a set of three volumes, fully describes the MS OS/2 system functions and related data types, macros, structures, messages, and file formats. The *Programmer's Reference* is the source for specific information about programming for MS OS/2.

This manual, *Volume 1* of the three-volume set, describes the purpose of the MS OS/2 system functions and the concepts and principles behind the functions. *Volume 1* is intended for programmers new to MS OS/2 or learning parts of MS OS/2 for the first time. This volume provides the basic information needed for an understanding of MS OS/2.

Volumes 2 and *3* consist of alphabetical listings of MS OS/2 system functions and related data types, macros, structures, and messages. These two volumes define the details of the syntax, parameters, and return values of each MS OS/2 system function. *Volumes 2* and *3* are intended to be used by programmers already acquainted with MS OS/2 and who need only specifics of particular functions.

1.3 How to Use This Manual

This manual describes the MS OS/2 system functions in individual-topic chapters. Each chapter describes the portion of MS OS/2 that lets an application carry out a specific task or set of related tasks. For example, the chapter on memory management defines the basic memory management terms, describes the role of the memory-management functions, and illustrates how to use those functions.

Each chapter has three parts: a general description, programming samples, and a summary. The general description contains a thorough discussion of the purpose and operation of pertinent MS OS/2 functions. The programming samples show how to use those MS OS/2 functions in applications to carry out useful tasks. The summary briefly describes each function and message described in the chapter.

In many cases, the reader must have some basic knowledge of other portions of MS OS/2 in order to understand the concepts described in the chapter. Each chapter lists the prerequisite topics.

1.4 MS OS/2 and the C Programming Language

The C programming language is the preferred development language for MS OS/2 programs. Many of the programming features of MS OS/2 were designed with C and other high-level languages in mind. MS OS/2 programs can also be developed in Pascal, FORTRAN, BASIC, and assembly language, but C is the most straightforward and easiest language to use to access MS OS/2 functions. For this reason, all syntax and program samples in this manual are written in the C programming language.

The MS OS/2 system functions use many types, macros, and structures that are not part of standard C language. These types, macros, and structures have been defined to make the task of creating MS OS/2 programs simpler and to make program sources clearer and easier to understand.

All types, macros, and structures discussed in this manual are defined in the MS OS/2 C-language include files. Programmers may wish to use these include files when developing MS OS/2 programs in other computer languages such as Pascal or assembly language. If include files for a given language are not available, a programmer can translate the definitions by following the guidelines given in *Volumes 2* and *3* of the *Programmer's Reference*.

Many chapters in this manual include program examples. These examples show how to use MS OS/2 system functions to accomplish simple tasks. In nearly all cases, the examples are code fragments, not complete programs. A code fragment is intended to show the context in which a function can be used; it often assumes that variables, structures, and constants used in the example have been defined and/or initialized. A code fragment may also use comments to represent a task instead of giving the actual statements.

Although the examples are not complete, you can use them in your programs by taking the following steps:

- Include the *os2.h* file in your program.
- Define the appropriate include constants for the functions, structures, and constants used in the example.
- Define and initialize all variables.
- Replace comments that represent tasks with appropriate statements.
- Check return values for errors and take appropriate actions.

Some examples in this manual combine both MS OS/2 and C run-time functions to carry out their tasks.

1.5 MS OS/2 Naming Conventions

In this manual, all parameter, variable, structure, field, and constant names conform to MS OS/2 naming conventions. MS OS/2 naming conventions are rules that define how to create names that indicate both the purpose and data type of an item used with MS OS/2 system functions. These naming conventions are used in this manual to help you readily identify the purpose and type of the function parameters, structure fields, and variables. These conventions are also used in most MS OS/2 sample program sources to make the sources more readable and informative.

The following list briefly describes the MS OS/2 naming conventions:

Item	Convention
Variable Parameter Field	All variable, parameter, and field names consist of up to three elements: a prefix, a base type, and a qualifier. The names always consist of at least a base type or a qualifier. In most cases, the name also includes a prefix. The base type identifies the data type of the item; the prefix specifies additional information, such as whether the item is a pointer, an array, or a count of bytes. The qualifier specifies the purpose of the item. All letters in the prefix and base type are lowercase. The letters in the qualifier are mixed-case (both uppercase and lowercase). When naming variables, the prefix and base type are optional for common integer types such as **SHORT** and **USHORT**.
Structure	All structure names consist of a word or phrase that specifies the purpose of the structure. All letters in the structure name are uppercase.

Item	Convention
Constant	All constant names consist of a prefix (derived from the name of the function associated with the constant) and a word or phrase that specifies the meaning of the constant in terms of a value, action, color, or condition. All letters in the constant name are uppercase and an underscore (_) separates the prefix from the rest of the name.
Function	All function names consist of a three-letter system prefix and a word or phrase that describes the action of the function. Each word in the function name starts with an uppercase letter. Verb and noun combinations, such as **DosGetDateTime**, are recommended.

The following examples show some of the standard prefix and base types you will see in this manual:

Prefix/Base type	Description	Example
f	Boolean flag; TRUE if successful	`BOOL fSuccess;`
ch	8-bit character	`CHAR chChar;`
s	16-bit signed integer	`SHORT sRate;`
l	32-bit signed integer	`LONG lDistance;`
uch	8-bit unsigned character	`UCHAR uchScan;`
us	16-bit unsigned integer	`USHORT usHeight;`
ul	32-bit unsigned integer	`ULONG ulWidth;`
b	8-bit unsigned integer	`BYTE bAttribute;`
sz	Zero-terminated array of characters	`CHAR szName[];`
fb	Array of flags in a byte	`BYTE fbMask;`
fs	Array of flags in a short	`USHORT fsMask;`
fl	Array of flags in a long	`ULONG flMask;`
sel	16-bit segment selector	`SEL selSegment;`
p	32-bit far pointer to a given type	`PCH pchBuffer;`
np	16-bit near pointer to a given type	`NPCH npchBuffer;`
a	Array of a given type	`CHAR achData[1];`
i	Index to an array of a given type	`USHORT ichIndex;`
c	Count of items of a given type	`USHORT cb;`
hf	Handle identifying a given object	`HFILE hf;`
off	Offset	`USHORT offSeg;`
id	Identifier for a given object	`USHORT idSession;`

1.6 Notational Conventions

The following notational conventions are used throughout this manual:

Convention	Meaning
bold	Bold type is used for keywords—for example, the names of functions, data types, structures, and macros. These names are spelled exactly as they should appear in source programs.
italics	Italic type is used to indicate the name of an argument; this name must be replaced by an actual argument. Italics are also used to show emphasis in text.
`monospace`	Monospace type is used for example program-code fragments.

Chapter

2

MS OS/2 Overview

2.1 Introduction 15

2.2 MS OS/2 and Presentation Manager 15
2.2.1 Queued Input 16
2.2.2 Device-Independent Graphics 16
2.2.3 Shared Resources 16

2.3 The Window Manager 17
2.3.1 Windows 17
2.3.2 Menus 18
2.3.3 Dialog Windows 18
2.3.4 The Message Loop 18

2.4 The Graphics Programming Interface 19
2.4.1 Presentation Spaces and Device Contexts 19
2.4.2 Graphics Primitives 19
2.4.3 Other Graphics Tools 20
2.4.4 Drawing 21
2.4.5 Retained Graphics and Segments 22
2.4.6 Metafiles 22

2.5 System Services 22
2.5.1 Multitasking 22
2.5.2 Dynamic Linking 23
2.5.3 Memory Management 23
2.5.4 The File System 23
2.5.5 Full-Screen Keyboard, Mouse, and Video Operations 24
2.5.6 Interprocess Communication 24

2.6 The MS OS/2 System Functions 25

2.1 Introduction

This chapter is an overview of the features of Microsoft Operating System/2. The most important of these features are the graphical user interface and device-independent graphics provided by the MS OS/2 Presentation Manager and the multitasking and other system services provided by the MS OS/2 base. In particular, this overview describes the following:

- MS OS/2 and Presentation Manager
- The window manager
- The graphics programming interface (GPI)
- The system services
- The MS OS/2 system functions

2.2 MS OS/2 and Presentation Manager

In a multitasking environment, it is important to give all applications some portion of the screen through which they can interact with the user. One of the principal goals of MS OS/2 is to provide visual access to most, if not all, applications at the same time. This access can be granted either by giving selected applications full use of the screen while other applications wait in the background or by letting applications share the screen. In MS OS/2, each application decides which method to use by choosing a "session" to run in. The session dictates whether the application receives complete control of the screen or must share it with other applications.

When MS OS/2 first starts, it creates the Presentation Manager session. All applications in this session share the screen. Applications that run in this session are called Presentation Manager applications, since Presentation Manager is the portion of MS OS/2 that creates and manages the Presentation Manager session. When a new application starts, it can direct the system to create a new session for it. The new session gives complete control of the screen to the application. Applications that use the full screen are called full-screen applications.

A Presentation Manager application shares the display with other applications by using a "window" for interaction with the user. Basically, a window is a rectangular portion of the system display that the system grants to an application. However, a window is also a combination of visual devices, such as menus, controls, and scroll bars, with which the user directs the actions of the application.

A Presentation Manager application must create its own window before producing any output or receiving any input. Once the application creates its window, MS OS/2 provides the application with detailed information about what the user is doing with the window and automatically carries out many of the tasks the user requests, such as moving and sizing the window.

A Presentation Manager application can create and use any number of windows to display information in a variety of ways. The system manages the screen, controlling the placement and display of windows and ensuring that no two applications attempt to access the same part of the system display at the same time. (In the latter case, the system overlaps the window of one application with the window of the other.)

2.2.1 Queued Input

In traditional programming environments, a program reads from the keyboard by making an explicit call to a function (**getchar**, for example). The function typically waits until the user presses a key before returning the character code to the program. A Presentation Manager application does not make explicit calls to read from the keyboard. Instead, MS OS/2 receives all input from the keyboard, mouse, and timer into its system queue and automatically redirects the input to the application by copying it from the system queue to the application queue. When the application is ready to retrieve input, it reads from its queue and dispatches the message to the appropriate window.

In Presentation Manager, input from the keyboard and mouse is provided automatically to every window that is created. MS OS/2 provides input in a uniform format called an input message. This message contains information about the input that far exceeds the information available in other environments. An input message specifies the system time, the position of the mouse, the state of the keyboard, the scan code of the key (if a key is pressed), the number of the mouse button (if a button is pressed), and the device that generated the message. For example, the keyboard message WM_CHAR corresponds to a press or release of a specific key. In each message, MS OS/2 provides a device-independent virtual-key code that identifies the key, as well as the device-dependent scan code generated by the keyboard. The message also specifies the status of other keys on the keyboard, such as SHIFT, CTRL, and NUMLOCK. Keyboard, mouse, and timer messages all have the same format and are processed in the same manner.

2.2.2 Device-Independent Graphics

In Presentation Manager, you have access to a rich set of device-independent graphics operations. This means that your applications can easily draw lines, rectangles, circles, and complex regions, and can use the same calls and data to draw on a high-resolution graphics display as they use to draw on a dot-matrix printer.

MS OS/2 requires device drivers to convert graphics-output requests to output for a printer, plotter, display, or other output device. A device driver is a special executable library that an application can load and use to carry out graphics operations in the "context" of the specific device—that is, the device driver, the output device, and the communications port.

2.2.3 Shared Resources

MS OS/2 is a multitasking system. This means that more than one application can run at a time. Presentation Manager applications must share the display, the keyboard, the mouse, and even the CPU with all other applications that are currently running in the same session. For this reason, MS OS/2 carefully controls these resources and requires applications to use a specific program interface that guarantees this control.

2.3 The Window Manager

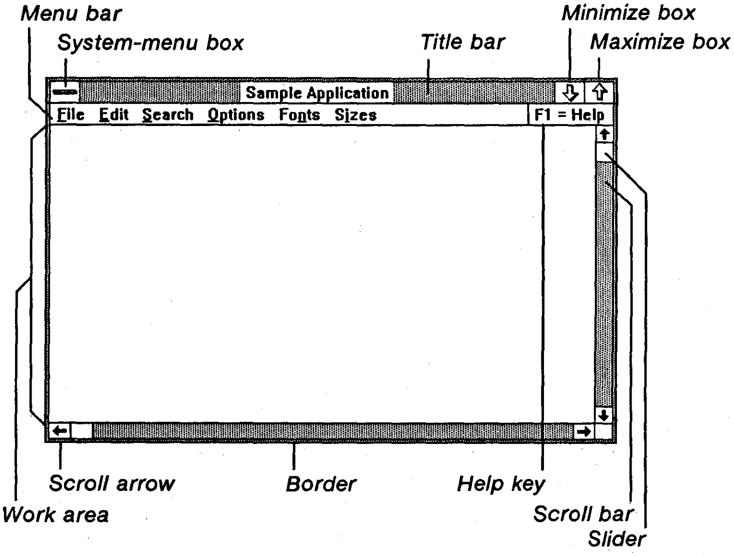
The MS OS/2 window manager consists of the MS OS/2 system functions that let applications create and manage windows and related elements. These related elements are primarily menus, dialog windows, and the message loop. The window manager provides the elements that your applications need to construct a graphical user interface.

2.3.1 Windows

A window is the primary input and output device of any Presentation Manager application. It is the application's only access to the system display, so, since nearly all Presentation Manager applications interact with the user in some way through the system display, these applications must use windows.

A window is a rectangle on the system display. A typical window is composed of a title bar, a menu bar, scroll bars, borders, and other features. You list the features you want for a window when you create the window. MS OS/2 then draws and manages the window. Figure 2.1 shows the main features of a window:

Figure 2.1
A Typical Presentation Manager Window



Interestingly, most Presentation Manager user-interface elements are also windows, including menus, title bars, buttons, entry fields, icons, and scroll bars.

Although an application creates a window and technically "owns" it, the management of the window is actually a collaborative effort between the application and the system. The system maintains the position and appearance of the window, manages the standard window features such as the border, scroll bars, and title, and carries out many tasks initiated by the user that directly affect the window.

The application maintains everything else about the window—in particular, the client window, in which the application is free to display anything it wants.

To manage this collaborative effort, MS OS/2 advises each window of changes that might affect it. Every window must have a corresponding window procedure that receives these window-management messages and responds appropriately. Window-management messages either specify actions for the function to take or request information from the function.

2.3.2 Menus

Menus are the principal means of user input for a Presentation Manager application. A menu is a list of commands that the user can view and choose from. When you create an application, you supply the menu name and the command names. MS OS/2 displays and manages the menus for you, sending a message to the window procedure when the user makes a choice. This message is the signal to carry out the command.

2.3.3 Dialog Windows

A dialog window is a temporary window that you can create to let the user supply more information for a command. A dialog window contains one or more controls. A control is a small window that has a very simple input or output function. The controls in a dialog window give the user a means of supplying filenames, choosing options, and otherwise directing the action of the command. For example, an entry-field control lets the user enter and edit text.

2.3.4 The Message Loop

Since your application receives input through a message queue, the chief feature of any Presentation Manager application is the message loop. The message loop retrieves input messages from the message queue and dispatches them to the appropriate windows.

For example, MS OS/2 collects hardware input, in the form of messages, in its system queue. It then copies this input to the appropriate message queue. The message loop in the application retrieves a message from the message queue and dispatches it, through the system, to the appropriate window procedure. The window procedure can respond to an input message by calling MS OS/2 functions to carry out work in the window.

For a more specific example, consider how the system and an application collaborate to process keyboard-input messages. The system receives keyboard input when the user presses and releases a key. The system copies the keyboard messages from the system queue to the application's message queue. The message loop retrieves the keyboard messages, translates them into ANSI-character WM_CHAR messages, and dispatches the WM_CHAR messages to the appropriate window procedure. The window procedure then uses the **GpiCharString** function to display the character in the client window.

MS OS/2 sends window-management messages directly to a window (**Win**) function. For example, after MS OS/2 carries out a request to destroy a window, it sends a WM_DESTROY message directly to the window procedure, bypassing the message queue. The window procedure must then use the **WinPostMsg** function to copy a WM_QUIT message into the message queue, signaling the main function that the window is destroyed and that the application should terminate. When the message loop retrieves the WM_QUIT message, the loop terminates and the main function exits.

2.4 The Graphics Programming Interface

The graphics programming interface (GPI) consists of the MS OS/2 system functions that let you create device-independent graphics for your applications. The **Gpi** functions are used in conjunction with the window manager to draw lines, shapes, and text in a window. Applications can also use the **Gpi** functions to draw graphics output on such devices as raster printers and vector plotters.

2.4.1 Presentation Spaces and Device Contexts

A presentation space is the key to an application's access to the system display, to printers, and to other graphics-output devices. Conceptually, a presentation space is a device-independent space in which you can create and manipulate graphics for display. The presentation space defines your drawing environment by specifying the tools you have available to create graphics. These tools include the graphics primitives granted to every presentation space, as well as the bitmaps and fonts that your application loads for its exclusive use.

Actually, a presentation space is little more than a data structure whose fields contain values that define the drawing environment. The values represent the colors, widths, styles, and other attributes of the graphics you draw. The system creates the data structure when you create the presentation space and initializes the structure to default values.

You must create a presentation space to create graphics. You must also create a device context to display those graphics on a device. A device context is a bridge from a presentation space to a specific device. You create a device context by specifying the device you want to access and the type of access you want, such as direct or queued (for printing). You begin displaying graphics on the device by associating the device context with the presentation space. Once you have associated the device context, any lines, text, and images you draw in the presentation space are also displayed on the given device.

Like a presentation space, a device context is a data structure. It contains information about the device driver that supports the specified device. The device driver interprets graphics commands sent to it from the presentation space and creates the corresponding commands for its device. It then sends the commands either directly to the device or to the spooler, depending on the type of access you gave the device context when you created it.

2.4.2 Graphics Primitives

In MS OS/2, graphics primitives are lines, arcs, markers, text, areas, and images. They are called primitives because you use them as the basic tools to create the documents, pictures, and other composite graphics that your applications display to the user.

You draw a primitive by using a **Gpi** function. For example, to draw a line, you use the **GpiLine** function and specify the ending point of the line. The function uses the current point as the starting point for the line and draws from the starting point to the ending point. The current point is simply the ending point of the last primitive, unless you explicitly set the current point by using a function such as **GpiMove**.

A line primitive is a straight line. An arc primitive is a curve. Curves can be arcs of a circle or of an ellipse, or they can be more complex curves such as splines and fillets. A marker primitive is a mark or character that you draw at a specific point. Markers are typically used to plot points in a graph. An area primitive is a closed figure that has been filled with a pattern. A common use for an area primitive is to represent a cross-section in a mechanical drawing. An image is a bitmapped image, with each bit representing the color of a pel (picture element) on the device. Images are often used for complex pictures that cannot easily be drawn.

Every primitive has a corresponding set of primitive attributes. The attributes specify the color, style, size, and orientation of the primitive when your application draws it. The primitive attributes are given default values when you create the presentation space, so you can use the primitives immediately. However, you can reset the attributes at any time. You have the choice of changing the attributes for individual primitives or changing a specific attribute for all primitives. For example, you can set the color for all primitives by using the **GpiSetColor** function, or you can set it for just the line primitive by using the **GpiSetAttrs** function.

2.4.3 Other Graphics Tools

In addition to the graphics primitives, MS OS/2 provides graphics tools that you can use to draw graphics and to affect how the graphics are drawn. These tools are paths, bitmaps, clipping areas, transformations, and color tables.

A path is a sequence of lines that you can use to create a filled area, a geometric line, or a clip path. A path is very much like an area primitive, in that you can use the path as a closed figure and fill it with a pattern. Unlike an area primitive, however, a path can be used to create geometric lines, sometimes called wide lines. Geometric lines are drawn, using a given width and pattern, so that they follow the outline specified by the path. Geometric lines give you a selection of line styles and patterns that are not available with the line primitive.

A bitmap is an array of bits that represents an image that you can display on a raster output device. Bitmaps typically represent scanned images and icons and are very much like image primitives. Unlike an image primitive, however, a bitmap can have several different formats, each format specifying color information that an image primitive cannot contain. Also, bitmaps can be used to create fill patterns that you can use to fill figures created using paths and area primitives. Finally, bitmaps can be copied from one presentation space to another or even from one location to another within the same presentation space.

A font is a collection of characters and symbols that you can use to draw text. Characters in a font belong to the same typeface and share stroke-width and serif characteristics. Some common fonts are 12-point Helvetica, 10-point Times Roman Bold, and 12-point Courier Italic. To use fonts in an application, you first create a logical font that describes the typeface and other characteristics that you want. Then you use the local identifier for the logical font to set that font as the current font for the presentation space. Subsequent text functions use the current font to draw text.

Clipping is a process that limits graphics output to a specific region on the display or on a page of printer paper. You can use clipping with a presentation space by creating a clipping area. The clipping area is the region where output can appear. If an application attempts to draw output outside a clipping area, the system will "clip" the output, preventing it from appearing on the device. You can create a clipping area for a presentation space by setting the dimensions of the graphics field and viewing limits or by creating a clip path or clip region. The final clipping area is the intersection of these four possible clip regions.

A transformation defines how the system should map the points in one coordinate space onto another coordinate space. Since all graphics primitives and other drawing tools use coordinate spaces, a transformation affects the way all graphics are drawn by your application. For example, you can use a transformation to move a figure from one place to another on the display or to rotate or adjust the size of the figure. Transformations are typically used to give the user different perspectives on a single drawing or to create rotated or sheared figures that would be time-consuming for the application to plot and draw.

A logical color table is an array of colors that an application uses when drawing graphics. Any primitive or other graphic you draw has one of the colors given in the table. You specify a color by giving a color index. The index identifies the table entry defining the color you want. Every presentation space has a default color table when it is created, but you can create a new logical color table to replace the default table if you need other colors. Creating a new table associates the color indexes with whatever color you have specified in the corresponding table entry.

2.4.4 Drawing

You draw graphics by using the MS OS/2 drawing functions. A drawing function draws a primitive or other graphic, applying the primitive attributes and whatever information you supply to the function when you call it. For example, when drawing line primitives, the system applies the current line color and style. The style determines whether the line is solid or a series of dashes, dots, or both.

Some attributes apply to all graphics primitives. For example, the foreground and background colors and mix modes affect all primitives. The foreground color defines the color of the primitive and the background color defines the color "behind" the primitive. For a line drawn using a dashed style, the dashes have the foreground color and the gaps between the dashes have the background color. The mix modes define how the foreground and background colors are combined with colors already on the display. The mix mode can cause the color to overpaint the existing color, leave it alone, or mix with it by using a binary operator such as the exclusive-OR operator.

Some attributes are specific to a particular graphics primitive. For example, the arc parameters apply only to arcs. The arc parameters specify a transformation that maps a circle to another circle, ellipse, or similar shape. When you draw an arc, the system uses the shape defined by the transformed circle as the shape for your arc. You supply a multiplier to set the final size of the arc.

A number of drawing functions use loadable resources to draw graphics. For example, the text-drawing functions, such as **GpiCharString** and **GpiCharStringPos**, can use a loaded font to draw text. To make a loadable resource available for these functions, you typically load the resource into memory and create a local identifier for the resource. For example, to use a font resource, you load it using the **GpiLoadFonts** function and then set the local identifier by using the **GpiCreateLogFont** function. Once you have a local identifier, you can set the resource to be the current resource by using a function such as **GpiSetCharSet**. Along with the text-drawing functions, the marker and area functions can use resources when they draw.

2.4.5 Retained Graphics and Segments

MS OS/2 lets you retain the graphics you draw in your application by storing them in retained segments. You create a retained segment by setting the drawing mode of the presentation space to DM_RETAIN or DM_DRAWANDRETAIN and opening the segment. All subsequent graphics are stored in the segment (and are also drawn on the device, if you specified DM_DRAWANDRETAIN). You can close the segment at any time and draw the contents by using a function such as **GpiDrawSegment**.

Retained segments are useful for storing graphics that result from user input. Once stored, the graphics can be redrawn or edited at any time. An element pointer lets the application move to a specific graphics element in a segment. The element can then be drawn or replaced, or new elements can be inserted.

2.4.6 Metafiles

A metafile, created by using a special device context, is another method of storing graphics. In this case, you associate the metafile device context with the presentation space, draw the graphics you want in the metafile, and then disassociate the device context and close it. Closing the metafile returns a handle that you can use to save the metafile in a disk file.

Metafiles are a useful way of transferring graphics images from one computer to another. An application can load a metafile from disk and play it into a presentation space. The presentation space can be associated with any device—display or printer. The graphics in the metafile are stored as graphics commands, not as a bitmap, so an application can examine and extract portions of the metafile if necessary.

2.5 System Services

The system services consist of all the MS OS/2 system functions that let you create processes and threads, access disk files and devices, allocate memory, and retrieve or set information about the system. In Presentation Manager applications, the system-service functions are typically used to carry out tasks for which no corresponding window-manager or **Gpi** function exists. In full-screen programs, system-service functions are used almost exclusively, even to interact with the user and access the devices of the computer.

2.5.1 Multitasking

Multitasking, one of the principal features of MS OS/2, is the ability of the system to manage the execution of more than one program at a time. This ability helps to optimize use of the computer, since time normally spent by a program waiting for user input is distributed to other programs that may be printing a document or recalculating a spreadsheet.

MS OS/2 provides multitasking in the traditional sense of having more than one program run at a time, and it also extends this concept to permit a single program to run more than one copy of itself at the same time.

Every program that has been loaded into memory and is running is called a process. Each copy of a process is called a thread. A process always has at least one thread, called the main thread or thread 1, and can create more threads. These additional threads are useful for carrying out tasks unrelated to the processing of the main thread. For example, a process may create a thread to write data to a disk file. This frees the main thread so that it can continue to process user input.

2.5.2 Dynamic Linking

Dynamic linking lets a program gain access at run time to functions that are not part of its executable code. These functions are contained in dynamic-link libraries—special program modules that contain executable code but cannot be run as programs. Instead, programs load the appropriate dynamic-link libraries and execute the code in the libraries by linking to them dynamically.

Dynamic-link libraries are very common in MS OS/2. In fact, most of the system is contained in dynamic-link libraries. The chief advantage of dynamic-link libraries is that they reduce the amount of memory needed by a program. A program loads a library only if it needs to execute a function in the library. Once the library is loaded, the system also shares it with any other program that needs it. This means that only one copy of the library is ever loaded at any one time.

2.5.3 Memory Management

Programs can, at any time, allocate additional memory for their own use. MS OS/2 controls access to system memory through the use of selectors. A selector is a unique number identifying a specific segment in memory. When a program allocates a segment, it specifies the size of the segment (in bytes) and receives a selector for that segment. This selector can then be used to access the memory.

MS OS/2 protects memory from unauthorized use. The process that allocates memory owns that memory, and no other process can access it. Attempting to access memory owned by another process causes a protection violation and usually terminates the process.

If two processes need to share memory, a process can create shared memory and either pass the selector to the process that is to share the memory or pass the name of the shared segment to that process. When two processes share a segment, no protection violation occurs for them, but the memory remains protected from all other processes. The sharing processes must manage the shared memory.

2.5.4 The File System

MS OS/2 programs have complete access to the disk files and devices of the computer. MS OS/2 manages its disk files and its devices in essentially the same way. For example, a program can use the same functions to open and read from a disk file as it uses to open and read from a serial port. Each open file or device is identified by a unique file handle. The program uses the handle in system functions to access the file or device.

MS OS/2 lets programs create, open, move, and delete files and directories in the file system. When a process opens a file, it specifies whether the file can be shared—that is, whether it can be accessed and possibly modified by other processes. This sharing also applies to devices that a process may open. Processes can open any device directly, including the parallel port, the serial ports, and the disk drive. MS OS/2 provides a wide range of input-and-output-control functions that a process can use to access and set the modes of the devices it has opened.

Ordinarily, the system automatically opens three files when a program starts: the standard-input, standard-output, and standard-error files. These files correspond to the keyboard and the full-screen display. The program can use these files to read from the keyboard and write to a full-screen display.

2.5.5 Full-Screen Keyboard, Mouse, and Video Operations

For full-screen programs, MS OS/2 provides access to the keyboard, the mouse, and the video display. A program can open these devices in much the same way as it opens a file. The MS OS/2 keyboard (**Kbd**) functions return much more information about a keystroke than do the standard file-system functions. Also, the keyboard functions let a program create logical keyboards and manage these keyboards for the processes in the same screen group.

Similarly, a program can open the mouse and read events from the mouse-event queue. An event is a mouse motion or button click. The program can also manage the mouse pointer, moving it, hiding it, and showing it as necessary.

Any full-screen program can write individual characters and strings directly to a character-based display. Unlike Presentation Manager applications, which must write characters to windows, a full-screen program has complete control of the system display while its session is in the foreground. The program can write both characters and attributes to the display, read characters from the display, and change modes for the display. A program that uses the video functions in a full-screen session must manage the display for that session.

The keyboard and mouse functions should not be used in Presentation Manager applications, since the system provides its own mouse and keyboard management. Many of the video functions can be used in a special type of Presentation Manager application called an advanced-video-input-and-output (AVIO) program. An AVIO program creates a window but uses the video functions to write text to the window.

2.5.6 Interprocess Communication

MS OS/2 provides several methods of interprocess communication: semaphores, pipes, signals, and queues.

A semaphore is a special variable that a process can use to signal the beginning and ending of a given operation and to prevent more than one thread within the process from accessing a specific resource at the same time. A process can create and use three types of semaphores: system, RAM, and fast-safe RAM. System semaphores are used between processes to control access to a shared resource. RAM semaphores are used between threads in the same process to control a resource or to signal the end of an operation. Fast-safe RAM semaphores are used between threads *or* processes to control a resource. The RAM semaphores are typically used when semaphore processing must be fast.

A pipe is a special file that two processes can use to transfer data. Although a pipe is like a file, it does not correspond to a device or a file on disk. Instead, the pipe is maintained by the system. Two processes use a pipe by opening the pipe and retrieving two handles: a read handle and a write handle. One process uses the write handle to write data to the pipe. The other process uses the read handle to read the data from the pipe.

A signal is a special interrupt that is sent to a process by the system or by another process. The signal temporarily stops normal execution of the process and causes the process to execute a signal handler. Signals are typically used to stop a process and exit. For example, pressing the CTRL+C key combination in a full-screen session generates a signal that usually stops the current process. The signal handler defines what a process does when it receives a signal. If a process does not want default signal handling, it can disable a signal or replace the signal handler with one of its own.

A queue is a special buffer that a process creates and shares with other processes. A queue is a convenient way for one process to channel data from two or more related processes into a single buffer. Note that this kind of queue is different from the message queue used by Presentation Manager. The queues are not related.

2.6 The MS OS/2 System Functions

The MS OS/2 system functions give applications access to all the features of MS OS/2. The MS OS/2 features, such as windows, device-independent graphics, and multitasking, let you create programs that make optimal use of the computer's memory, display, and CPU while still meeting the needs of a wide range of users through either the traditional character-based interface or the graphical user interface of Presentation Manager.

The MS OS/2 system functions are organized into several distinct groups, as described in the following list:

Function group	Usage
Dev	Use the Presentation Manager device (**Dev**) functions to open and control Presentation Manager device drivers. These functions let you create device contexts that you can associate with a presentation space and use with the **Gpi** functions to carry out device-independent graphics operations for displays, printers, and plotters.
Dos	Use the disk-operating-system (**Dos**) functions in full-screen and Presentation Manager sessions to read from and write to disk files, to allocate memory, to start threads and processes, to communicate with other processes, and to access the computer's devices directly. Most functions in this group can be used in Presentation Manager applications.

Function group	Usage
Gpi	Use the graphics-programming-interface (**Gpi**) functions to create graphics output for a display, a printer, or other output devices. The **Gpi** functions give you a full range of graphics primitives, from lines to complex curves to bitmaps. You choose the attributes for the primitives (such as color, line width, and pattern) and then draw lines, text, and shapes. The retained-graphics capability lets you save the drawings in segments and build complex pictures by drawing a chain of segments.
Kbd	Use the keyboard (**Kbd**) functions in full-screen sessions to read keystrokes from the keyboard, to manage multiple logical keyboards, and to change code pages and translation tables. Since the Presentation Manager session provides its own keyboard support, **Kbd** functions are not needed in Presentation Manager applications.
Mou	Use the mouse (**Mou**) functions in full-screen sessions to read mouse input from the mouse-event queue, to set the mouse-pointer shape, and to manage the mouse for all processes in a session. As with the keyboard, the Presentation Manager session provides its own mouse support, so **Mou** functions are not needed in Presentation Manager applications.
Vio	Use the video-input-and-output (**Vio**) functions in full-screen sessions to write characters and character attributes to the screen, to create pop-up windows for messages, to change video modes, and to access physical video memory. **Vio** functions can also be used in advanced-video-input-and-output (AVIO) applications for the Presentation Manager session, to write characters and character attributes in a window. Most Presentation Manager applications, however, use the graphics-programming-interface (**Gpi**) functions to write text in a window.
Win	Use the window-manager (**Win**) functions to create and manage windows. Presentation Manager applications use windows as the main interface with the user. The **Win** functions let you create menus, scroll bars, and dialog windows that let the user choose commands and supply input. Your application receives all mouse and keyboard input as messages from the message queue. The **Win** functions let you retrieve messages from the queue and dispatch them to the window the input is intended for.

Chapter

3

MS OS/2 Programming Models

3.1 Introduction 29

3.2 Full-Screen Programs 29

3.3 Presentation Manager Applications 30

3.4 The Family Application Programming Interface 32

3.5 Using the Command Line 37

3.6 Using Structures 38

3.7 Using Bit Masks 39

3.8 Sharing Resources 40

3.9 C-Language Header Files 40

3.1 Introduction

This chapter describes the types of programs that you can develop for MS OS/2. MS OS/2 supports the following program types:

- Full-screen programs
- Full-screen programs in a window
- Presentation Manager applications
- Advanced-video-input-and-output (AVIO) programs
- Family-application-programming-interface (FAPI) programs

3.2 Full-Screen Programs

A full-screen program is any MS OS/2 program that does not create a Presentation Manager message queue. In other words, it is a program that does not rely on the Presentation Manager mouse and keyboard processing for input. Full-screen programs typically run in a full-screen session.

Most full-screen programs use the **Dos** functions to perform input, output, memory management, and other activities. Full-screen programs also commonly use the standard-input, standard-output, and standard-error files created for them when they start.

A full-screen program uses a **main** function as its starting point and can call as many other functions as needed to complete its designated task. The following simple full-screen program copies the line "Hello, world" to the screen:

```
#include <os2.h>

main( )
{
    USHORT cbWritten;

    DosWrite(1, "Hello, world\r\n", 14, &cbWritten);
}
```

The MS OS/2 system functions use many structures, data types, and constants that are not part of the standard C language. For example, the data type **USHORT** is a special MS OS/2 data type that specifies an unsigned short integer. In order to use these items, you must include the MS OS/2 header file *os2.h* at the beginning of your program source file. For more information about the C-language header files, see Section 3.9.

The MS OS/2 system functions are not standard C functions. They use the Pascal calling convention. This means, for example, that the MS OS/2 functions expect parameters to be passed in left-to-right order instead of the standard right-to-left order of C functions. To use the MS OS/2 functions in a C-language program, you must declare them with the **pascal** keyword, which directs the C compiler to generate proper instructions for the function call. All MS OS/2 functions are declared this way within the *os2.h* file, so including the file saves you the trouble of declaring each function individually.

The *os2.h* file also declares the parameter types for each function. Without these declarations, many function parameters would require type casting to avoid compiler errors. For example, the **DosWrite** function shown in the preceding example requires the second parameter to be a complete far (32-bit) address to the given string. Since the *os2.h* file declares the second parameter with this type, the compiler does the cast for you.

Some full-screen programs can also run in a window in the Presentation Manager session. Although the program runs in a window, it does not create the window. Instead, the system creates the window and provides the input and output to the program just as if it were running in a full-screen session. A full-screen program can run in a window only if it does not use functions that directly access the devices that Presentation Manager controls. For example, a program that attempts to retrieve the address of the video buffer or to change video modes may fail.

3.3 Presentation Manager Applications

A Presentation Manager application is any MS OS/2 program that creates a message queue. A window is the only means a Presentation Manager application has to receive input and display output, so Presentation Manager applications typically create one or more windows to interact with the user.

All MS OS/2 Presentation Manager applications have essentially the following structure:

- A **main** function
- One or more window procedures
- Optional functions to support the main function and/or the window procedures

Since nearly all Presentation Manager applications create and use windows, the **main** function carries out the same basic tasks in most applications. The typical **main** function does the following, in the order shown:

1. Initializes the application for Presentation Manager.
2. Creates a message queue.
3. Creates a window class.
4. Creates a window.
5. Starts the message loop and continues to dispatch messages until the WM_QUIT message is retrieved.
6. Destroys the window when finished using it.
7. Destroys the message queue.
8. Terminates the application.

Every MS OS/2 Presentation Manager application has at least one thread of execution. Each thread that calls Presentation Manager functions must register with the system by calling the **WinInitialize** function. This function creates an anchor block and returns an anchor-block handle that the thread can use in subsequent functions.

An anchor block links a process with the system. The anchor block includes an instance data segment in which to store the process's environment and storage for error messages. The anchor-block handle is used in the call to the

WinTerminate function that ends the association with the anchor block just before the application terminates.

The application creates the message queue by using the **WinCreateMsgQueue** function. This function returns a queue handle that can be used in subsequent functions. Once the queue is created, the application can register a window class, create a window and start the message loop. After the message loop ends, the application can destroy the window and use the queue handle in the **WinDestroyMsgQueue** function to destroy the queue.

Once the application is initialized and a message queue and window are created, the application can enter the main message loop. The application waits there for messages to appear in the queue, retrieves them, and dispatches them, as appropriate, to its windows. When the user or system chooses to terminate an application, a WM_QUIT message is used to trigger an exit from the message loop.

After leaving the message loop, an application carries out various termination activities, including destroying windows, releasing memory, destroying message queues, closing files, and severing connections with the shell and other applications.

The following code fragment from a simple Presentation Manager application copies the line "Hello, world" to its window:

```
#define INCL_WIN
#define INCL_DOS
#include <os2.h>
HAB hab;              /* anchor-block handle     */
HMQ hmq;              /* message-queue handle    */
QMSG qmsg;            /* message-queue structure */

MRESULT CALLBACK MyWindowProc(HWND, USHORT, MPARAM, MPARAM);

HWND hwndFrame;       /* frame-window handle     */
HWND hwndClient;      /* client-window handle    */
ULONG flStyle = FCF_TITLEBAR | FCF_SYSMENU | FCF_SIZEBORDER |
                FCF_MINMAX | FCF_SHELLPOSITION | FCF_TASKLIST;

main()
{
    /*
     * Initialize the thread for making Presentation Manager calls and
     * create the message queue.
     */

    hab = WinInitialize(0);
    hmq = WinCreateMsgQueue(hab, DEFAULT_QUEUE_SIZE);

    /* Register the class, terminate on failure. */

    if (!WinRegisterClass(hab, "MyClass",
            MyWindowProc, CS_SIZEREDRAW, NULL))
        DosExit(EXIT_PROCESS, 0);

    /* Create the window, terminate on failure. */

    if (!(hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE,
            &flStyle, "My Window!", 0L, NULL, 0, &hwndClient)))
        DosExit(EXIT_PROCESS, 0);

    /* Get and dispatch messages. */

    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
        WinDispatchMsg(hab, &qmsg, NULL, 0, 0);
    WinDestroyWindow(hwndFrame);    /* destroy the main window   */
    WinDestroyMsgQueue(hmq);        /* destroy the message queue */
    WinTerminate(hab);              /* terminate                 */
}

MRESULT CALLBACK MyWindowProc(hwnd, usMessage, mp1, mp2)
HWND hwnd;
USHORT msg;
MPARAM mp1;
MPARAM mp2;
{
    HPS hps;        /* presentation-space handle */
    RECTL rcl;      /* rectangle structure       */
    POINTL ptl;     /* point structure           */

    switch (msg) {
        case WM_PAINT:
            hps = WinBeginPaint(hwnd,
                NULL, NULL);                       /* start painting  */
            WinQueryWindowRect(hwnd, &rcl);        /* get window size */
            WinFillRect(hps, &rcl, CLR_WHITE);     /* fill background */
            ptl.x = (rcl.xRight - rcl.xLeft) / 2;
            ptl.y = (rcl.yTop - rcl.yBottom) / 2;
            GpiMove(hps, &ptl);          /* move to center of window  */
            GpiCharString(hps, 12L,
                "Hello, world");                     /* draw string   */
            WinEndPaint(hps);                        /* end painting  */
            return (OL);

        default:
            return (WinDefWindowProc(hwnd, msg, mp1, mp2));
    }
}
```

An advanced-video-input-and-output (AVIO) program is a Presentation Manager program that uses the advanced **Vio** functions for text output. These function let a Presentation Manager application write text to a window just as if it were writing the text to a full screen. These programs must run in the Presentation Manager session and must create at least one window for input and output.

3.4 The Family Application Programming Interface

Many MS OS/2 functions can be used in programs intended to be run in real mode. These functions, collectively called the family application programming interface (family API, or FAPI), let developers create MS OS/2 programs that can run in both protected and real modes; that is, they can run under MS OS/2 and under MS-DOS versions 2.*x* and 3.*x*.

To use the family API in real-mode programs, you must use only the MS OS/2 functions that belong to the FAPI, and you must observe the restrictions that apply to these functions when running in real mode. Also, you must bind your program by using the Microsoft Operating System/2 Bind utility (**bind**). The **bind** utility supplies the code needed to link the MS OS/2 functions to the corresponding MS-DOS system calls. This code is used only when the program is run in real mode; that is, a bound program can still run in protected mode.

Not all MS OS/2 functions belong to the FAPI, and some that do belong have slightly different behavior when used in real mode than when used in protected mode. The following is a complete list of the FAPI functions. Those functions marked with a dagger (†) operate differently in real mode than in protected mode; all other FAPI functions operate identically in both protected and real modes.

DosAllocHuge †

DosAllocSeg †

DosBeep

DosBufReset

DosCaseMap †

DosChDir

DosChgFilePtr

DosClose

DosCreateCSAlias †

DosDelete

DosDevConfig

DosDevIOCtl †

DosDupHandle

DosErrClass

DosError †

DosExecPgm †

DosExit †

DosFileLocks

DosFindClose

DosFindFirst †

DosFindNext †

DosFreeSeg †

DosGetCollate †

DosGetCtryInfo †

DosGetDateTime

DosGetDBCSEv †

DosGetEnv

DosGetHugeShift

DosGetMachineMode

DosGetMessage †

DosGetVersion

DosHoldSignal †

DosInsMessage †

DosMkDir

DosMove

DosNewSize

DosOpen †

DosPutMessage †

DosQCurDir

DosQCurDisk

DosQFHandState

DosQFileInfo

DosQFileMode

DosQFSInfo

DosQVerify

DosRead †

DosReallocHuge †

DosReallocSeg †

DosRmDir

DosSelectDisk

DosSetDateTime

DosSetFHandState †

DosSetFileInfo

DosSetFileMode

DosSetFSInfo

DosSetSigHandler †

DosSetVec †

DosSetVerify

DosSleep

DosSubAlloc

DosSubFree

DosSubSet

DosWrite

KbdCharIn †

KbdFlushBuffer †

KbdGetStatus†

KbdPeek †

KbdSetStatus †

KbdStringIn†

VioGetBuf

VioGetCurPos

VioGetCurType

VioGetMode

VioGetPhysBuf

VioReadCellStr

VioReadCharStr

VioScrLock †

VioScrollDn

VioScrollLf

VioScrollRt

VioScrollUp

VioScrUnLock

VioSetCurPos

VioSetCurType

VioSetMode

VioShowBuf

VioWrtCellStr

VioWrtCharStr

VioWrtCharStrAtt

VioWrtNAttr

VioWrtNCell

VioWrtNChar

VioWrtTTY

Note The **DosGetMachineMode** function is especially useful in FAPI programs, since it specifies which environment the program is running in: MS OS/2 or MS-DOS.

Following are the real-mode restrictions and/or differences in operation for the FAPI functions marked with daggers (†) in the preceding list:

DosAllocHuge Rounds the *usPartialSeg* parameter value up to the next paragraph (16-byte) value. This function copies the actual segment address, not a selector, to the variable pointed to by the *psel* parameter.

DosAllocSeg Rounds the *usSize* parameter value up to the next paragraph (16-byte) value. This function copies the actual segment address, not a selector, to the variable pointed to by the *psel* parameter.

DosCaseMap Provides no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

DosCreateCSAlias Returns as a selector the actual segment address of the allocated memory. Freeing either the returned selector or the original selector immediately frees the block of memory.

DosDevIOCtl Restricts the input-and-output-control functions that can be used. Categories 2, 3, 4, 6, 7, 10, and 11 cannot be used. Also, some control functions in categories 1, 5, and 8 can be used with MS-DOS 3.*x* but not with MS-DOS 2.*x*. The following input-and-output-control functions can be used in FAPI programs:

ASYNC_SETBAUDRATE
ASYNC_SETLINECTRL
DSK_BLOCKREMOVABLE
DSK_GETLOGICALMAP
DSK_LOCKDRIVE †
DSK_REDETERMINEMEDIA †
DSK_SETLOGICALMAP
DSK_UNLOCKDRIVE †
PRT_GETFRAMECTL
PRT_GETINFINITERETRY
PRT_GETPRINTERSTATUS
PRT_INITPRINTER
PRT_SETFRAMECTL (for IBM Graphics Printers only)
PRT_SETINFINITERETRY (current program only)

† These input-and-output-control functions can be used only with MS-DOS versions 3.2 and later.

DosError If the *fEnable* parameter is HARDERROR_DISABLE, causes all subsequent **int 24h** requests to fail, until another call is made to the **DosError** function with *fEnable* set to HARDERROR_ENABLE.

DosExecPgm Allows only the value EXEC_SYNC for the *fExecFlags* parameter. Other values cause errors. The buffer pointed to by the *pchFailName* parameter is filled with blanks, even if the function fails. The **codeResult** field of the **RESULTCODES** structure receives the exit code for the **DosExit** function or the MS-DOS call that terminates the program.

DosExit Exits from the currently executing program, since there are no threads in the real-mode environment. If the *fTerminate* parameter is EXIT_THREAD, the entire process ends, not just a thread.

DosFindFirst Requires the *phdir* parameter to be HDIR_SYSTEM.

DosFindNext Requires the *hdir* parameter to be HDIR_SYSTEM.

DosFreeSeg Does not treat a code-segment selector (created by using the **DosCreateCSAlias** function) and the corresponding data-segment selector as unique. Freeing one frees both.

DosGetCollate Provides no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

DosGetCtryInfo Provides no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

DosGetDBCSEv Provides no method of identifying the boot drive. The system assumes that the *country.sys* file is in the root directory of the current drive.

DosGetMessage Provides no method of identifying the boot drive. The system assumes that the message file is in the root directory of the current drive.

DosHoldSignal Recognizes only the signal-interrupt (SIG_CTRLC) and signal-break (SIG_CTRLBREAK) signals.

DosInsMessage Provides no method of identifying the boot drive. The system assumes that the message file is in the root directory of the current drive.

DosOpen Restricts the values that can be used with the *fsOpenMode* parameter. The parameter can be a combination of the following values:

Value	Meaning
OPEN_ACCESS_READONLY	Read-only access mode.
OPEN_ACCESS_WRITEONLY	Write-only access mode.
OPEN_ACCESS_READWRITE	Read/write access mode.
OPEN_SHARE_DENYREADWRITE	Deny read/write share mode. Not available in MS-DOS 2.*x*. Available in MS-DOS 3.*x* only when the **share** command has been used.
OPEN_SHARE_DENYWRITE	Deny-write share mode. Not available in MS-DOS 2.*x*. Available in MS-DOS 3.*x* only when the **share** command has been used.
OPEN_SHARE_DENYREAD	Deny-read share mode. Not available in MS-DOS 2.*x*. Available in MS-DOS 3.*x* only when the **share** command has been used.
OPEN_SHARE_DENYNONE	Deny-none share mode. Not available in MS-DOS 2.*x*. Available in MS-DOS 3.*x* only when the **share** command has been used.
OPEN_FLAGS_NOINHERIT	Inheritance flag. Not available in MS-DOS 2.*x*.

Value	Meaning
OPEN_FLAGS_WRITE_THROUGH	Write-through flag. Not available in MS-DOS 2.*x*.
OPEN_FLAGS_DASD	Direct-access-storage-device (DASD) flag.

The fail-on-error flag (OPEN_FLAGS_FAIL_ON_ERROR) is not available to real-mode programs.

DosPutMessage Provides no method of identifying the boot drive. The system assumes that the message file is in the root directory of the current drive.

DosRead Uses the **KbdStringIn** function whenever the specified file handle identifies the keyboard device. In real mode, **KbdStringIn** reads only the number of characters specified in the call, then beeps to signal the user that no additional characters can be entered. (In protected mode, the user can enter characters until the keyboard buffer is full.)

DosReallocHuge Rounds the *usPartialSize* parameter value up to the next paragraph (16-byte) value.

DosReallocSeg Rounds the *usNewSize* parameter value up to the next paragraph (16-byte) value.

DosSetFHandState Requires that the OPEN_FLAGS_FAIL_ON_ERROR flag and the OPEN_FLAGS_WRITE_THROUGH flag *not* be set. Also, the OPEN_FLAGS_NOINHERIT flag must not be set in MS-DOS 2.*x*.

DosSetSigHandler Can be used to install signal handlers for only the signal-interrupt (SIG_CTRLC) and signal-break (SIG_CTRLBREAK) signals. Furthermore, the SIG_CTRLC and SIG_CTRLBREAK signals are treated as the same signal, so the function accepts only the SIG_CTRLC value when setting a signal handler.

DosSetVec Does not accept VECTOR_EXTENSION_ERROR as the *usVecNum* value, since this exception is not raised in machines using the 8088 or 8086 microprocessor.

KbdCharIn Does not copy the system time to the **KBDKEYINFO** structure and provides no interim character support. This function retrieves characters only from the default keyboard (handle 0). The **fbStatus** field can be 0x0000 or SHIFT_KEY_IN. The *hkbd* parameter is ignored.

KbdFlushBuffer Ignores the *hkbd* parameter.

KbdGetStatus Does not support the interim or turnaround character.

KbdPeek Does not copy the system time to the **KBDKEYINFO** structure and provides no interim character support. This function retrieves characters only from the default keyboard (handle 0). The **fbStatus** field can be 0x0000 or SHIFT_KEY_IN. The *hkbd* parameter is ignored.

KbdSetStatus Does not support the interim character or the turnaround character. Raw input mode with echo mode on is not supported. The *hkbd* parameter is ignored.

KbdStringIn Ignores the *hkbd* parameter.

VioScrLock Always indicates that the lock was successful.

3.5 Using the Command Line

In standard C-language programs, you can use the *argc* and *argv* parameters of the **main** function to retrieve individual copies of the command-line arguments. You can use these parameters in MS OS/2 programs, but you can also retrieve the entire command line, exactly as the user typed it, by using the **DosGetEnv** function.

When it starts a program, MS OS/2 prepares an environment segment for the program. This segment contains definitions of all environment variables, as well as the command line. The **DosGetEnv** function retrieves the segment selector for this environment segment and the address offset within that segment for the start of the command line.

You can echo the command line on the screen by using the **DosGetEnv** function to get the address of the command line in the environment segment, as shown in the following sample program:

```
#define INCL_DOS
#include <os2.h>

main( )
{
    SEL selEnvironment;
    USHORT offCommand;
    PSZ pszCommandLine;
    USHORT cbWritten;
    USHORT i, cch;

    DosGetEnv(&selEnvironment, &offCommand);
    pszCommandLine = MAKEP(selEnvironment, offCommand);

    /*
     * The first string is the program name. The command line is the
     * next null-terminated string.
     */

    for (i = 0; pszCommandLine[i]; i++);

    /* Find the length of the command-line string. */

    for (i++, cch = 0; pszCommandLine[cch + i]; cch++);

    DosWrite(1, &pszCommandLine[i], cch, &cbWritten);
}
```

The command line has two parts. The first part is the program name, terminated by a zero byte. The second part is the rest of the command line, terminated by two zero bytes. The preceding program echoes the command line by skipping over the program name and then writing everything up to the next zero byte to the screen. The first **for** statement skips over the command name; the second **for** statement computes the length of the string. The **MAKEP** macro creates the far pointer that is needed to access the command line in the environment segment.

You can examine your program's environment by using the selector retrieved by the **DosGetEnv** function. The program's environment consists of the environment variables that have been declared and passed to the program. Each program has a unique environment that is typically inherited from the program that started it—for example, from the MS OS/2 command processor, **cmd**.

You can use the **DosScanEnv** function to scan for a specific environment variable. This function takes as an argument the name of the environment variable that you are interested in and copies the current value of this variable to a buffer that you supply. The following sample program uses **DosScanEnv** to display the value of the environment variable specified in the command line:

```
#define INCL_DOSQUEUES
#include <os2.h>

main( )
{
    SEL selEnvironment;
    USHORT offCommand;
    PSZ pszCommandLine;
    PSZ pszValue;
    USHORT cbWritten;
    USHORT i, cch;

    DosGetEnv(&selEnvironment, &offCommand);
    pszCommandLine = MAKEP(selEnvironment, offCommand);

    for (i = 0; pszCommandLine[i]; i++);
    for (i++; pszCommandLine[i] == ' '; i++);

    if (!DosScanEnv(&pszCommandLine[i], &pszValue)) {
        for (cch = 0; pszValue[cch]; cch++);
        DosWrite(1, pszValue, cch, &cbWritten);
    }
}
```

3.6 Using Structures

Many MS OS/2 functions use structures for input and output. To use a structure in an MS OS/2 function, you first define the structure in your program and then pass the 32-bit far address of the structure as a parameter in the function call.

For example, the **DosGetDateTime** function copies the current date and time to a **DATETIME** structure whose address you supply. The fields of the **DATETIME** structure define the month, day, and year, as well as the time of day (to hundredths of a second). The **DATETIME** structure, defined in the *os2.h* file, has the following form:

```
typedef struct _DATETIME {    /* date */
    UCHAR   hours;
    UCHAR   minutes;
    UCHAR   seconds;
    UCHAR   hundredths;
    UCHAR   day;
    UCHAR   month;
    USHORT  year;
    SHORT   timezone;
    UCHAR   weekday;
} DATETIME;
```

To retrieve the date and time, you call the **DosGetDateTime** function, using the address operator (&) to specify the address of the **DATETIME** structure. The following sample program shows how to make the call:

```
#include <os2.h>

CHAR szDayName[] = "MonTueWedThuFriSatSun";
CHAR szMonthName[] = "JanFebMarAprMayJunJulAugSepOctNovDec";
CHAR szDate[] = "xx:xx:xx xxx xxx xx, xxxx\r\n";
```

```
main( )
{
    DATETIME date;
    SHORT offset;
    SHORT i;
    USHORT usYear;
    USHORT cbWritten;

    DosGetDateTime(&date);  /* address of DATETIME structure */

    szDate[0] = (date.hours / 10) + '0';
    szDate[1] = (date.hours % 10) + '0';
    szDate[3] = (date.minutes / 10) + '0';
    szDate[4] = (date.minutes % 10) + '0';
    szDate[6] = (date.seconds / 10) + '0';
    szDate[7] = (date.seconds % 10) + '0';
    offset = date.weekday * 3;
    for (i = 0; i < 3; i++)
        szDate[i + 9] = szDayName[i + offset];
    offset = (date.month - 1) * 3;
    for (i = 0; i < 3; i++)
        szDate[i + 13] = szMonthName[i + offset];
    szDate[17] = (date.day < 10) ? ' ' : (date.day / 10 + '0');
    szDate[18] = (date.day % 10) + '0';
    usYear = date.year;
    szDate[21] = (usYear / 1000) + '0';
    usYear = usYear % 1000;
    szDate[22] = (usYear / 100) + '0';
    usYear = usYear % 100;
    szDate[23] = (usYear / 10) + '0';
    szDate[24] = (usYear % 10) + '0';

    DosWrite(1, szDate, 27, &cbWritten);
}
```

One drawback to using MS OS/2 functions exclusively is that there are no formatted output functions, such as the C-language **printf** function. Therefore, the preceding program formats the data itself before displaying it. The program uses the integer-division operators (/ and %) to convert binary numbers to ASCII characters. The program then copies the ASCII characters to a string and displays the string by using the **DosWrite** function.

Some MS OS/2 functions require that you fill one or more fields of the structure before calling the function. For example, there are some structures whose length depends on the version of the operating system being used; MS OS/2 requires that you supply the expected length so that the function does not copy data beyond the end of your structure.

3.7 Using Bit Masks

In MS OS/2, many functions use bit masks. A bit mask (also called an array of flags) is a combination of two or more Boolean flags in a single byte, word, or double-word value. In C-language programs, you can use the bitwise AND, OR, and NOT operators to examine and set the values in a bit mask.

If a function retrieves a bit mask, you can check a specific flag in the bit mask by using the AND operator, as shown in the following code fragment:

```
USHORT fsEvent;

if (fsEvent & 0x0004)
    /* is the flag set? */
```

You can set a flag in a bit mask by using the OR operator, as shown in the following code fragment:

```
ULONG flFunctions;

flFunctions = flFunctions | KR_KBDPEEK;
```

Finally, you can clear a flag in a bit mask by using the AND and NOT operators, as shown in the following code fragment:

```
USHORT fsEvent;

fsEvent = fsEvent & ~0x0004;
```

3.8 Sharing Resources

Many MS OS/2 functions let you use the resources of the computer, such as the keyboard, screen, disk, and even the system speaker. Since MS OS/2 is a multitasking operating system and more than one program may be running at a time, MS OS/2 considers all resources of the computer to be shared resources. As a result, programs must not claim exclusive access to a given resource.

Consider a simple program that plays a short tune by using the **DosBeep** function. This function, when called by a single program, generates a tone at the system speaker, but if two programs call **DosBeep** at the same time, the result is chaotic. For example, try running two or more copies of the following program at the same time:

```
#include <os2.h>

#define CNOTES 14
USHORT ausTune[] = {
    440, 1000,
    480, 1000,
    510, 1000,
    550, 1000,
    590, 1000,
    620, 1000,
    660, 1000
    };

main( )
{
    int i;

    for (i = 0; i < CNOTES; i += 2)
        DosBeep(ausTune[i], ausTune[i + 1]);
}
```

The first parameter of the **DosBeep** function specifies the frequency of the note. The second parameter specifies the duration. The array ausTune defines frequency and duration values for each note in the tune.

DosBeep is intended to be used for signaling the user when an error occurs, such as pressing an incorrect key. Since the system speaker is a shared resource, a process should use the **DosBeep** function sparingly.

3.9 C-Language Header Files

The MS OS/2 C-language header file *os2.h* contains the definitions you need to use the functions, data types, structures, and constants described in the *Microsoft Operating System/2 Programmer's Reference*.

When you include the *os2.h* file, the C preprocessor automatically defines many, but not all, of the most commonly used MS OS/2 functions. The *os2.h* header file is the "master" file of a set of files that contain the MS OS/2 function definitions. Each file contains definitions for the functions, data types, structures, and constants associated with a specific group of MS OS/2 functions. To minimize the time required to process the many header files, each function group is conditionally processed on the basis of whether a corresponding constant is defined within the program source file. The following is a list of these constants, with descriptions of the function groups they represent:

Constant	Meaning
INCL_AVIO	Includes all MS OS/2 version 1.1 AVIO functions.
INCL_BASE	Include all MS OS/2 version 1.1 system functions (**Dos**, **Vio**, **Kbd**, **Mou**).
INCL_BITMAPFILEFORMAT	Include the bitmap file-header structure BITMAPFILEHEADER.
INCL_DEV	Include all MS OS/2 version 1.1 device functions (**Dev**).
INCL_DEVERRORS	Include the **Dev**-function error constants.
INCL_DOS	Include all MS OS/2 version 1.1 kernel functions (**Dos**).
INCL_DOSDATETIME	Include the date/time and timer functions.
INCL_DOSDEVICES	Include the device and IOPL support functions.
INCL_DOSDEVIOCTL	Include all MS OS/2 version 1.1 input-and-output control functions (IOCtls).
INCL_DOSERRORS	Include the **Dos**-function error constants.
INCL_DOSFILEMGR	Include the file-management functions.
INCL_DOSINFOSEG	Include the information-segment functions.
INCL_DOSMEMMGR	Include the memory-management functions.
INCL_DOSMISC	Include miscellaneous **Dos** functions.
INCL_DOSMODULEMGR	Include the module-manager functions.
INCL_DOSMONITORS	Include the monitor functions.
INCL_DOSNLS	Include national-language-support functions.

Constant	Meaning
INCL_DOSNMPIPES	Include named-pipe functions.
INCL_DOSPROCESS	Include the process- and thread-support functions.
INCL_DOSQUEUES	Include the queue functions and other miscellaneous functions.
INCL_DOSRESOURCES	Include the resource-support functions.
INCL_DOSSEMAPHORES	Include the semaphore functions.
INCL_DOSSESMGR	Include the session-manager functions.
INCL_DOSSIGNALS	Include the signal functions.
INCL_DOSTRACE	Include the **DosPTrace** function.
INCL_ERRORS	Include all MS OS/2 version 1.1 error constants.
INCL_FONTFILEFORMAT	Include the font-file structures.
INCL_GPI	Include all MS OS/2 version 1.1 graphics-programming-interface functions (**Gpi**).
INCL_GPIBITMAPS	Include the bitmap and pel functions.
INCL_GPICONTROL	Include the basic presentation-space-control functions.
INCL_GPICORRELATION	Include the pick-aperture, boundary, and correlation functions.
INCL_GPIERRORS	Include the **Gpi**-function error constants.
INCL_GPILCIDS	Include the physical- and logical-font functions.
INCL_GPILOGCOLORTABLE	Include the logical-color-table functions.
INCL_GPIMETAFILES	Include the metafile functions.
INCL_GPIPATHS	Include the path and clipping functions.
INCL_GPIPRIMITIVES	Include the drawing-primitive and primitive-attribute functions.
INCL_GPIREGIONS	Include the region and clipping functions.
INCL_GPISEGEDITING	Include the segment-editing functions.

Constant	Meaning
INCL_GPISEGMENTS	Include the segment-control and drawing functions.
INCL_GPITRANSFORMS	Include the transformation and transform-conversion functions.
INCL_KBD	Include all MS OS/2 version 1.1 keyboard functions (**Kbd**).
INCL_MOU	Include all MS OS/2 version 1.1 mouse functions (**Mou**).
INCL_NOCOMMON	Exclude any function group not explicitly defined.
INCL_PM	Include all MS OS/2 version 1.1 Presentation Manager functions and structures.
INCL_SHLERRORS	Include the shell error constants.
INCL_SUB	Include all MS OS/2 version 1.1 video, keyboard, and mouse functions (**Vio**, **Kbd**, and **Mou**).
INCL_VIO	Include all MS OS/2 version 1.1 video functions (**Vio**).
INCL_WIN	Include all MS OS/2 version 1.1 window functions (**Win**).
INCL_WINACCELERATORS	Include the keyboard-accelerator functions.
INCL_WINATOM	Include the atom-manager functions.
INCL_WINBUTTONS	Include the button-control functions.
INCL_WINCATCHTHROW	Include the **WinCatch** and **WinThrow** support functions.
INCL_WINCLIPBOARD	Include the clipboard-manager functions.
INCL_WINCOUNTRY	Include the country-support functions.
INCL_WINCURSORS	Include the text-cursor functions.
INCL_WINDIALOGS	Include the dialog-box functions.
INCL_WINENTRYFIELDS	Include the entry-field functions.
INCL_WINERRORS	Include the **Win**-function error constants.
INCL_WINFRAMECTLS	Include the frame-control (title bar and size border) functions.
INCL_WINFRAMEMGR	Include the frame-manager functions.

Constant	Meaning
INCL_WINHEAP	Include the heap-manager functions.
INCL_WINHOOKS	Include the hook-manager functions.
INCL_WININPUT	Include the mouse- and keyboard-input functions.
INCL_WINLISTBOXES	Include the list-box-control functions.
INCL_WINMENUS	Include the menu-control functions.
INCL_WINMESSAGEMGR	Include the message-management functions.
INCL_WINPOINTERS	Include the mouse-pointer functions.
INCL_WINPROGRAMLIST	Include the shell-program-list API functions.
INCL_WINRECTANGLES	Include the rectangle functions.
INCL_WINSCROLLBARS	Include the scroll-bar-control functions.
INCL_WINSHELLDATA	Include the shell-data functions.
INCL_WINSTATICS	Include the static-control functions.
INCL_WINSWITCHLIST	Include the shell-switch-list API functions.
INCL_WINSYS	Include the system-value and color functions.
INCL_WINTIMER	Include the timer functions.
INCL_WINTRACKRECT	Include the **WinTrackRect** function.
INCL_WINWINDOWMGR	Include the general window-management functions.

To use a function within your program, you simply define the corresponding constant by using the **#define** directive before you include the *os2.h* file. For example, the following code fragment includes definitions for the memory-management and file-management functions:

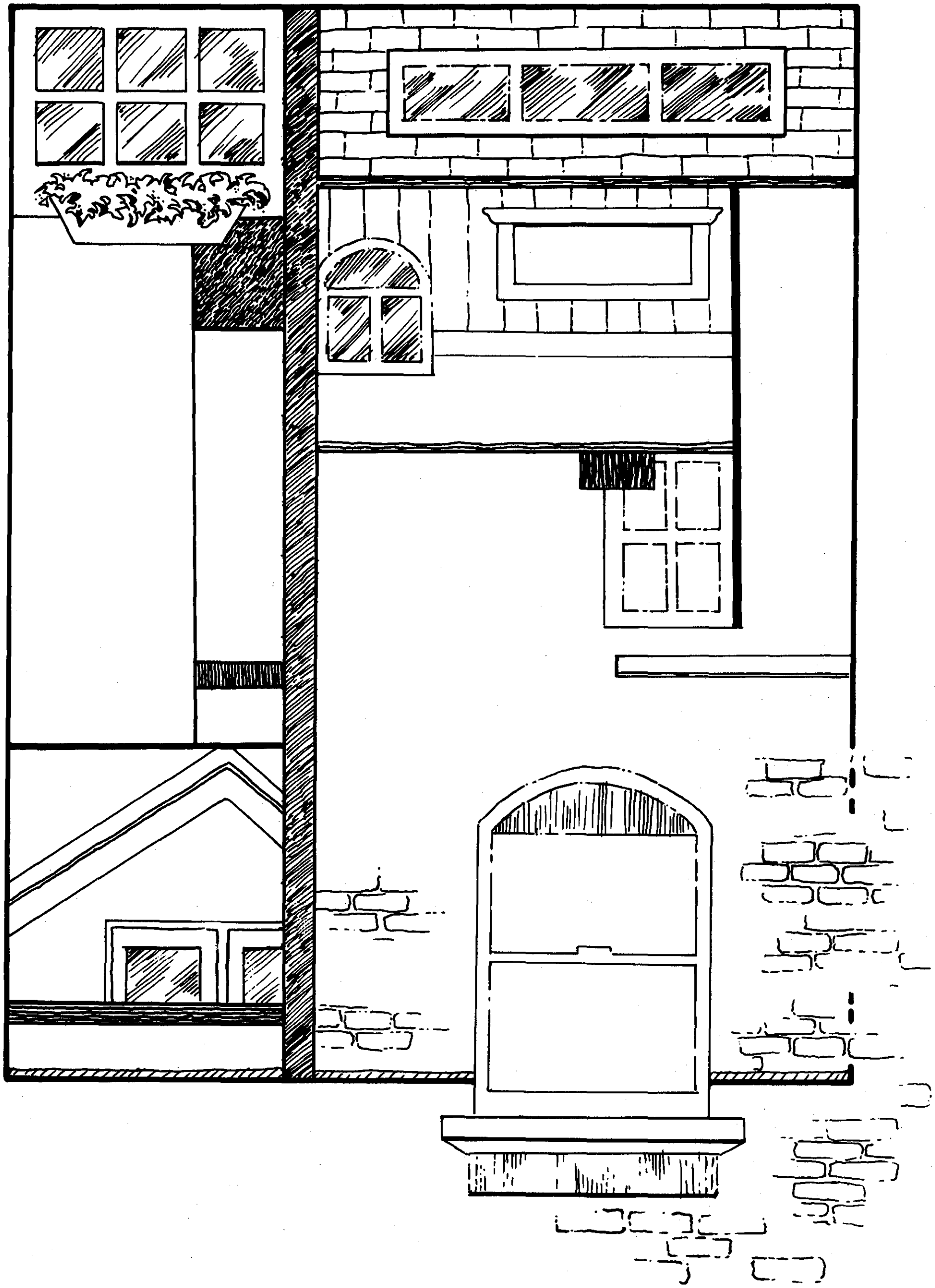
```
#define INCL_DOSMEMMGR
#define INCL_DOSFILEMGR
#include <os2.h>

main( )
{
    .
    .
    .
}
```

Once you have defined a constant, you can use any function, structure, or data type in that function group.

Part 2

Window Manager



Part 2

Window Manager

Chapter 4 Windows 49
Chapter 5 Messages and Message Queues 77
Chapter 6 Window Classes 91
Chapter 7 Window Procedures 99
Chapter 8 Mouse and Keyboard Input 109
Chapter 9 Frame Windows 125
Chapter 10 Control Windows 143
Chapter 11 Title-Bar Controls 151
Chapter 12 Button Controls 157
Chapter 13 Entry-Field Controls 169
Chapter 14 List-Box Controls 179
Chapter 15 Static Controls 193
Chapter 16 Scroll-Bar Controls 201
Chapter 17 Menus 213
Chapter 18 Accelerator Tables 237
Chapter 19 Dialog Windows 245
Chapter 20 Painting and Drawing 263
Chapter 21 Drawing in Windows 279
Chapter 22 Mouse Pointers and Icons 287
Chapter 23 Cursors 295
Chapter 24 Printing 301
Chapter 25 Heaps 319
Chapter 26 Clipboard 331
Chapter 27 Dynamic Data Exchange 345
Chapter 28 Hooks 363
Chapter 29 Help 377

Chapter

4

Windows

4.1 Introduction 51

4.2 About Windows 51

4.2.1 Desktop Windows 51

4.2.2 Application Windows 51

4.2.3 Window Creation 52

4.2.3.1 Window-Creation Functions 54

4.2.3.2 Window-Creation Messages 54

4.2.4 Window Handles 54

4.2.5 Window Size and Position 55

4.2.6 Window Styles 55

4.2.7 Window Destruction 56

4.2.8 Locked Windows 57

4.2.9 Disabled Windows 58

4.3 System-Modal Windows 58

4.3.1 Window Data 59

4.3.2 Subclassed Windows 59

4.3.3 Window Relationships 60

4.3.3.1 Parent-Child Relationship 61

4.3.3.2 Ownership 62

4.3.3.3 Object Windows 63

4.3.4 Visibility 63

4.3.5 Size 64

4.3.6 Position 64

4.3.6.1 Size and Position Messages 64

4.3.7 Z Order 64

4.3.8 Maximized and Minimized Windows 65

4.3.9 Redrawing Windows 65

4.3.10 System Commands 65

Chapter 4

4.4 Using Windows 66
4.4.1 Creating a Window 66
4.4.2 Creating a Frame Window 66
4.4.3 Destroying a Window 67
4.4.4 Setting and Querying Window Data 67
4.4.5 Creating a Top-Level Window............................. 67
4.4.6 Creating an Object Window 68
4.4.7 Changing the Parent Window............................. 68
4.4.8 Finding a Parent, Child, or Owner Window 68
4.4.9 Setting an Owner Window 69
4.4.10 Finding a Child or Owned Window....................... 69
4.4.11 Enumerating Top-Level Windows 69
4.4.12 Moving and Sizing a Window............................. 70
4.4.13 Moving a Window in a Stack of Windows............... 71
4.4.14 Showing and Hiding a Window........................... 72
4.4.15 Maximizing, Minimizing, and Restoring a Window..... 72

4.5 Summary 72
4.5.1 Window Functions.. 72
4.5.2 Standard Window Messages 73
4.5.3 Relationship Functions 74
4.5.4 Functions for Moving, Sizing, and Changing 74
4.5.5 Messages for Moving, Sizing, and Changing............ 75

4.1 Introduction

This chapter describes the portions of MS OS/2 that let you create and use windows; manage relationships between windows; and size, move, and display windows in your Presentation Manager application. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Application initialization and termination
- Messages and message queues
- Window classes and window procedures

4.2 About Windows

A window is a rectangular area of the screen where an application displays output and receives input from the user. You might think of a window as a "graphics terminal" that shares the screen with other terminals. Only one "terminal" is active at a time, and when it is, a user can use the mouse and keyboard to interact with the application that owns the terminal.

Unlike a graphics terminal, however, a window must be created by an application before it can be used. MS OS/2 does not create a window by default. This means that one of the first tasks of a Presentation Manager application is to create a window.

4.2.1 Desktop Windows

MS OS/2 automatically creates two windows: the desktop window and the desktop-object window. The desktop window is the base (bottom-most) window in the Presentation Manager session. It is the window that paints the background for this session. It serves as the base for all windows created and displayed by applications. The desktop-object window is like a desktop window that is never displayed. It serves as a base for windows that coordinate the activity of other windows that are not displayed.

4.2.2 Application Windows

Every application creates at least one window, called the main window, to serve as the "graphics terminal" for the application. The application also creates many other windows either directly or indirectly to carry out tasks related to the main window. In fact, most windows used by an application are composed of several different windows. Each window plays a part in displaying output and receiving input from the user.

Typically, an application's main window is made up of several windows acting together as one. The main window is usually a frame window that contains a client window and one or more control windows, such as a title bar and a System menu.

An application can use several types of windows: frame windows, client windows, control windows, dialog windows, message boxes, and menus.

A frame window is a special window that the application uses as the base when constructing a main window or other composite window. A frame window provides basic features, such as borders and system-command processing, that a main window needs to conform to the MS OS/2 user-interface guidelines.

A dialog window is a frame window that contains one or more control windows. Dialog windows are used almost exclusively for prompting the user for input. An application usually creates a dialog window when it needs additional information to complete a command. It then destroys the dialog window when the requested information has been entered.

A message box is a frame window that an application uses to display a note, caution, or warning to the user. Message boxes are commonly used to inform the user of problems the application encounters while carrying out a task.

A client window is the window in which the application displays the current document or data. For example, a desktop-publishing application displays the current page of a document in a client window. Most applications create at least one client window. The application must process input to the window and then display output.

A control window is any window used in conjunction with another window to carry out useful input or output tasks, such as displaying messages or reading text. MS OS/2 provides several predefined control-window classes that can be used to create control windows. Control windows include buttons, entry fields, list boxes, menus, scroll bars, static text, and title bars.

A menu is a control window that presents a list of commands and other menus to the user. The user chooses commands from the list by using a mouse or keyboard. The application then carries out the chosen task.

Many simple applications create only a main window. The application manages the client window and allows the frame and control windows to operate as defined by MS OS/2.

4.2.3 Window Creation

An application creates windows by using a window-creation function, such as **WinCreateWindow**, and supplying information about the window to be created. An application can create one or more windows in any thread for which it has created a message queue. An application creates a message queue for a thread by using the **WinCreateMsgQueue** function after initializing the application for Presentation Manager by using the **WinInitialize** function.

The following information must be supplied when creating a window:

- Window class
- Window name
- Parent window
- Window position relative to the parent window

- Window position relative to its sibling windows (Z order)
- Window width and height
- Window styles
- Owner window
- Window identifier
- Class-specific data

Every window belongs to a window class. The window class defines how the window behaves and appears when operating. The chief component of the window class is the window procedure. The window procedure is a function that receives and processes all input and requests for action sent to the window by the system. The window class also defines the class styles. These tell MS OS/2 what initial window styles to give a window created with this class.

A window can have a name. A window name is a text string that identifies the window for the user. The window name is typically displayed in the window or in a title bar within the window. How the name is used depends on the window class.

Every window created has a parent window. The parent window provides the coordinate system used for positioning the window and defines the relationship the new window has with other windows in the system. The parent window also affects the behavior and appearance of the window. For example, when the parent window is hidden, the child window is also hidden.

Every window has a position, size, and Z-order position. The position specifies the location on the screen of the window's lower-left corner. This position is relative to the lower-left corner of the parent window (in pels). A window's size is the width and height in the window (in pels). A window's Z-order position specifies the position of the window in the stack of overlapping windows. The window at the top of the Z order overlaps all sibling windows (that is, windows having the same parent window). A window at the bottom of the Z order is overlapped by all sibling windows. An application sets a window's Z-order position by placing it behind a given sibling window.

Every window can have a style. The window style specifies how the window behaves or appears. For example, a window style can specify whether the window is visible or invisible when first created. A few window styles apply to all windows, but most apply to windows of specific window classes. The window procedure for that class interprets the style.

A window can be owned by another window. An owner window is similar to a parent window, but it does not affect the behavior or appearance of the window in the same way. The owner window usually coordinates the activity of a window so that it can operate in conjunction with other windows. The window sends messages about its state to its owner window; the owner window sends messages about what action to carry out next.

A window can have a window identifier. A window identifier uniquely identifies a window that operates in conjunction with other windows. A window identifier is especially useful if a window sends information to the owner window.

A window can have class-specific data. This data further defines how the window behaves and appears when first created. The system passes class-specific data to the window procedure. The window procedure then applies the data to the new window.

4.2.3.1 Window-Creation Functions

The basic window-creation function is **WinCreateWindow**. The **WinCreateWindow** function takes window class, style, size, and position information and creates a new window. All other window-creation functions, such as **WinCreateStdWindow** and **WinCreateDlg**, supply some of this information by default and create windows of a specific class or style.

Although **WinCreateWindow** provides the most direct means of creating a window, most applications do not use it. Instead, they typically use the **WinCreateStdWindow** function to create a main window and use the **WinDlgBox** or **WinCreateDlg** function to create dialog windows.

The **WinCreateMenu**, **WinLoadMenu**, **WinLoadDlg**, **WinMessageBox**, and **WinCreateFrameControls** functions also create windows. Each of these functions substitutes for one or more calls to the **WinCreateWindow** function required to create a given window. For example, you can create a frame window, one or more control windows, and a client window, all in a single call to **WinCreateStdWindow**.

4.2.3.2 Window-Creation Messages

The system sends messages to the window procedure as it creates a window. Each window procedure receives a WM_CREATE message, specifying that the window is being created.

The system also sends a WM_ADJUSTWINDOWPOS message, specifying the size and position for the window. This message allows the window procedure to adjust the size and position before they are actually applied to the window.

The system also sends other messages. The number and order of these messages depend on the window class and style and on the function used to create the window.

4.2.4 Window Handles

When a window is created, the creation function returns a window handle. A window handle uniquely identifies the window and can be used in functions to direct the action of the function to the window. Window handles have the data type **HWND**; applications must use this type when declaring the variables that hold window handles.

There are several special constants that can be used in place of a window handle in certain functions. For example, HWND_DESKTOP can be used in the **WinCreateWindow** function to specify the desktop window as the new window's parent window. Similarly, HWND_OBJECT represents the desktop-object window. HWND_TOP and HWND_BOTTOM represent the top and bottom positions when setting the Z-order position for a window.

Although the NULL constant is not a window handle, it can be used in some functions to specify that no window is affected. For example, NULL can be used in the **WinCreateWindow** function to specify that there is no owner window. Some functions may return NULL, indicating that the given action applies to no window.

4.2.5 Window Size and Position

A window's size and position can be expressed as a bounding rectangle, given in coordinates relative to its parent window. The window's size and position can be explicitly specified when it is created, or the system can use default values. The window's size and position can be changed at any time.

The default coordinate system for a window specifies that the point (0,0) is at the window's lower-left corner; coordinates increase upward and to the right.

When two sibling windows overlap, the system must specify which window is displayed in front. This ordering of sibling windows is known as the Z order. For more information about Z order, see Section 4.3.7.

4.2.6 Window Styles

A window style is a value that specifies how a window behaves or appears in a given situation. Window styles let applications adapt windows of a given class for special circumstances. For example, an application can give a window the style WS_SYNCPAINT to cause it to paint immediately whenever any portion of the window becomes invalid. A window normally paints only if there are no messages waiting in the message queue.

An application usually sets the window style when it creates the window. It can also set the window style after creation by using the **WinShowWindow** and **WinSetWindowULong** functions. MS OS/2 provides several standard window styles that apply to all windows. It also provides many styles for the predefined frame and control windows. The frame and control styles are unique to each predefined window class and can be used only for windows belonging to the corresponding class.

Initially, the class styles of the window class used to create the window determine the window styles of the new window. If the window class has the style CS_SYNCPAINT, all windows created using that class have the style WS_SYNCPAINT by default.

MS OS/2 has the following standard window styles:

Style	Description
WS_VISIBLE	Makes the window visible. MS OS/2 draws the window on the screen unless overlapping windows completely obscure it. Windows without this style are hidden. If overlapping windows completely obscure the window, the window is still considered to be visible. Visibility simply means that MS OS/2 draws the window if it can.

Style	Description
WS_DISABLED	Disables mouse and keyboard input to the window. This style is used to temporarily prevent the user from using the window.
WS_CLIPCHILDREN	Prevents a window from painting over its child windows.
WS_CLIPSIBLINGS	Prevents a window from painting over its sibling windows.
WS_PARENTCLIP	Prevents a window from painting over its parent window.
WS_SAVEBITS	Saves the image under the window as a bitmap. When the window is moved or hidden, the system restores the image by copying the bits.
WS_SYNCPAINT	Causes the window to immediately receive WM_PAINT messages after a part of the window becomes invalid. Without this style, the window receives WM_PAINT messages only if no other message is waiting to be processed.
WS_MINIMIZED	Reduces the window to the minimum size.
WS_MAXIMIZED	Enlarges the window to the maximum size.
WS_GROUP	Identifies the window as the first dialog item in a group of dialog items. This style is used with controls in dialog windows to permit the user to move among the controls by pressing the direction keys.
WS_TABSTOP	Identifies the window as a tabstop window. This style is used with controls in dialog windows to permit the user to move to the control by pressing the TAB key.

4.2.7 Window Destruction

An application can destroy the windows it has created. When a window is destroyed, the system hides the window if it is visible, and then removes any internal data associated with the window. This invalidates the window handle; it can no longer be used in functions. An application destroys a window by using the **WinDestroyWindow** function.

Most applications destroy the windows they create soon after creating them. For example, an application usually destroys any dialog windows as soon as the application has sufficient input from the user to continue its task. An application eventually destroys the main window of the application (before terminating). In general, an application must destroy all the windows it creates.

Destroying a window does not affect the window class from which the window is created. New windows can still be created using that class and any existing windows of that class continue to operate.

Destroying a window also destroys that window's child and owned windows. The **WinDestroyWindow** function sends a WM_DESTROY message to the window, which in turn sends the same message to all its child and owned windows. Each child and owned window passes the message on to other child and owned windows. In this way, all descendant windows of the window being destroyed are also destroyed.

Before destroying a window, an application should save or remove any data associated with the window and release any resources. For example, a presentation space created for the window by the **WinGetPS** function must be released by calling the **WinReleasePS** function. This must be done before calling the **WinDestroyWindow** function. If a presentation space is associated with the device context for the window, the application should disassociate or destroy the presentation space by using the **GpiAssociate** or **GpiDestroyPS** function before calling **WinDestroyWindow**. Failing to release a resource can cause an error.

The **WinDestroyWindow** function may send several messages to a window. The following is a list of possible messages sent by **WinDestroyWindow**:

Message	Description
WM_DESTROY	Always sent to the window being destroyed after it has been hidden, but before its child windows have been destroyed.
WM_ACTIVATE	Sent with the first message parameter equal to FALSE if the window being destroyed is the active window.
WM_OTHERWINDOWDESTROYED	Sent to all main windows of the window being destroyed and to its descendant windows, if the window being destroyed has been registered with the **WinRegisterWindowDestroy** function.
WM_RENDERALLFMTS	Sent if the clipboard owner is being destroyed and there are unrendered formats on the clipboard.

If the window being destroyed is the active window, both the active and focus states are transferred to another window. The window that becomes the active window is the next window (as defined by the ALT+ESC key combination). The new active window determines which window has the input focus.

4.2.8 Locked Windows

A window can be locked. An application typically locks a window to prevent it from being destroyed. This is useful whenever a window needs to access data that may be lost if the associated window is destroyed. An application can lock a window by using the **WinLockWindow** function.

Each window has a lock count. When a window is created, its lock count is set to zero, meaning that the window is unlocked. An application can use the **WinLockWindow** function to increment or decrement the lock count. If the lock count is greater than zero, the window is locked. If the lock count is zero, the window is unlocked. The lock count can never be less than zero. An application can retrieve the current lock count by using the **WinQueryWindowLockCount** function.

The **WinQueryWindow**, **WinQueryActiveWindow**, and **WinQuerySysModalWindow** functions also lock a window if specified.

4.2.9 Disabled Windows

A window can be disabled. A disabled window temporarily receives no keyboard or mouse input. An application typically disables a window to prevent the user from using the window. For example, the application may disable a push button in a dialog window to prevent the user from choosing it. An application can enable a disabled window at any time. Enabling a window restores normal input. An application enables or disables a window by using the **WinEnableWindow** function.

By default, a window is enabled when created. The WS_DISABLED style can be specified, however, to disable a new window. If an application uses the **WinEnableWindow** function to disable an existing window, that window also loses the keyboard focus. The keyboard focus is set to NULL, meaning no window has the focus. If a child window or other descendant window has the keyboard focus, the descendant window loses it when the window is disabled.

An application can determine whether a window is disabled by using the **WinIsWindowEnabled** function.

4.3 System-Modal Windows

System-modal windows require the user to respond immediately to warnings about the state of the system. Because the system-modal window receives all keyboard and mouse input, all other windows are effectively disabled when the system-modal window is set; the user cannot continue working in other windows until the system-modal window has been cleared. An application sets and clears the system-modal window by using the **WinSetSysModalWindow** function.

Due to its absolute control of input, applications must use care when setting a system-modal window. Ideally, an application uses a system-modal window only when there is danger of losing data if the user does not respond to the problem immediately.

Although an application can destroy a system-modal window, the new active window will also be the new system-modal window. An application can also make another window active while the system-modal window exists. Again, the new active window is also the new system-modal window. In general, once a system-modal window is set, a system-modal window will continue to exist in the Presentation Manager session until explicitly cleared.

4.3.1 Window Data

Every window has an associated data structure. The window data structure contains all the information specified for the window when it was created and any additional information supplied for the window since creation. Although the exact number and meaning of fields in the window data structure is private to the system, an application can directly access any of the following fields:

- Pointer to window-class data structure
- Pointer to window procedure
- Parent-window handle
- Owner-window handle
- Handle of first child window
- Handle of next sibling window
- Window size and position (expressed as a rectangle)
- Lock count
- Window style
- Window identifier
- Update-region handle
- Message-queue handle

An application can examine and modify these fields by using functions such as **WinQueryWindowUShort** and **WinSetWindowUShort**. These functions let an application access fields, such as the lock count, which are stored as 16-bit integers. Other functions let an application access fields containing 32-bits integers and pointers. There are several fields that indirectly affect the fields in the window data structure. For example, the **WinLockWindow** function modifies the lock-count field; the **WinSubclassWindow** function replaces the window-procedure pointer.

An application can extend the number of available fields in the window data structure by specifying a count of extra bytes when it registers the corresponding window class. The window procedure can then use these bytes to store information about the window. Functions such as **WinQueryWindowUShort** and **WinSetWindowUShort** give direct access to the extra bytes.

If a window needs more than a few bytes of storage added to the window data structure, using extra bytes alone is not the best solution. One common alternative is to dynamically allocate some memory and then store a pointer to that dynamic memory in the extra bytes of the window data structure.

4.3.2 Subclassed Windows

A subclassed window is any window whose original window procedure has been replaced with another window procedure. The original window procedure is specified by the window class used to create the window. An application typically subclasses a window (replaces the window procedure) so that it can support additional capabilities in a window created with a given class. For example, an application may subclass a push-button control so that it can add sound when the user chooses the button. An application subclasses a window by using the **WinSubclassWindow** function.

Typically, a window procedure used to subclass a window will pass most (if not all) messages on to the original window procedure. The usual goal of subclassing is to add capability. The **WinSubclassWindow** function returns the address of the original window procedure, making it easy to call the original function from the new window procedure. The following code fragment shows the general format of a window procedure used for subclassing:

```
PFNWP pfnwp;

main() {

    /* Subclass in main function or other window procedure. */

    pfnwp = WinSubclassWindow(hwnd, MySubClass);
}

MRESULT CALLBACK MySubClass(hwnd, usMessage, mp1, mp2)
HWND hwnd;
USHORT usMessage;
MPARAM mp1;
MPARAM mp2;
{
    switch (usMessage) {
      .
      . /* Process messages. */
      .
    }
    return (pfnwp(hwnd, usMessage, mp1, mp2));
}
```

Note that the replacement window procedure calls the original window procedure instead of the **WinDefWindowProc** function.

An application can subclass only one window at a time. It cannot subclass an entire class.

4.3.3 Window Relationships

Window relationships define how windows interact with each other on the screen and through messages. There are parent-child window relationships and ownership relationships.

The parent-child relationship determines how a window looks when drawn on the screen. It also determines what happens to a window when a related window is destroyed or hidden. The parent-child rules apply to all windows at all times and cannot be modified.

Ownership determines how windows communicate using messages. Cooperative windows define the rules of ownership and then carry them out. Although some windows, such as windows belonging to the preregistered, public window class WC_FRAME, have quite complex rules of ownership, the application ordinarily defines the ownership rules.

4.3.3.1 Parent-Child Relationship

Most windows have a parent window. (The exceptions to this rule are the desktop and the desktop-object windows. These windows, created by the system when it first starts, have no parent windows.) An application sets the parent window when it creates the window; the system uses the parent window to determine where and how to draw the new window, as well as when to destroy the window.

A window is drawn relative to its parent window. The coordinates given to specify the position of a window's lower-left corner are relative to the lower-left corner of its parent window. For example, a window whose coordinates are (10,10) is placed 10 pels left and 10 pels up from the lower-left corner of its parent window. A window is a top-level window if its parent window is the desktop window. Top-level windows are drawn relative to the lower-left corner of the screen (the desktop window's lower-left corner).

Windows with the same parent window are called sibling windows. All top-level windows are sibling windows since they share a common parent window, the desktop window. Sibling windows can overlap; an application or a user can arrange the windows so that some appear on top of others. Every sibling window has a Z-order position that specifies where it lies in the stack of overlapping windows. The parent window for the sibling windows is always at the bottom of the stack.

A window is clipped to its parent window. This means that no part of a child window is ever drawn outside of its parent window. If an application creates a child window that is larger than the parent window or positions a child window so that some or all of the window extends beyond the edges of the parent window, the system automatically clips (does not draw) the portion of the child window that extends beyond the edges. Depending on the window styles for a window, a window may also be clipped to its child and its sibling windows. When a window has the style WS_CLIPCHILDREN or WS_CLIPSIBLINGS, the system clips the window.

A window is destroyed when its parent window is destroyed. When the parent window is destroyed, the system sends WM_DESTROY messages to each child window. This is convenient for composite windows (for example, the application's main window) since an application needs to destroy only the parent window; all the related windows, including the client window, are destroyed automatically. The parent window is always the last window to be destroyed. This allows the parent window to use any data saved or left behind by its child windows.

While every window has only one parent window, a window can have any number of child windows. Any child window can have child windows. Each child window in this chain of windows is a called a descendant window of the original parent window. Immediate child windows are child windows directly related to the parent window, not just descendant windows.

An application can change a window's parent window at any time. Changing the parent window changes where and how the child window is drawn.

4.3.3.2 Ownership

Any window can have an owner window. An owner is a window, not necessarily a parent window, that controls some aspect of another window. Applications typically use ownership to establish a connection between windows so that together they can carry out useful tasks. For example, the title bar in an application's main window is owned by the frame window. Together they let the user move the entire main window by clicking the mouse in the title bar. An application can set the owner window when it creates the window, or it can set the owner window at a later time.

Ownership establishes a relationship between windows that is independent of the parent-child relationship. Unlike parent and child windows, there are no predefined rules for how the owner and owned windows interact. The window procedures for the owner and owned windows must carry out any special interactions specified.

The preregistered, public window classes provided by MS OS/2 recognize ownership. Control windows, created with classes such as WC_TITLEBAR and WC_SCROLLBAR, notify their owners of events; frame windows, created using the WC_FRAME class, receive and process notification messages from the control windows they own. For example, a title-bar control sends a notification message to its owner when it receives a mouse click. If the owner is a frame window, the frame window receives the notification message and prepares to move the frame window and its child windows.

Owner and owned windows must be created by the same thread; that is, they must belong to the same message queue. Since ownership is independent of the parent-child relationship, the owner and owned windows do not have to be descendants of the same parent window. This means one window can be a descendant of the desktop window and the other a descendant of the desktop-object window. This can affect how windows are destroyed. Destroying the owner window does not necessarily destroy the owned window. An application must explicitly destroy any owned window that is not a descendant window of the owner.

Frame windows often have owned windows that are not descendants (they are sibling windows instead). A frame window has the following special properties:

- Destroys all owned windows, even if they are not descendants, when the frame window is destroyed.
- Moves owned windows when the frame window moves. The owned windows that are not descendants maintain their position relative to the upper-left (not the usual lower-left) corner of the owner window. Any owned window with the style FS_NOMOVEWITHOWNER does not move.
- Changes the Z-order position of all owned windows when the frame window changes.
- Hides all owned windows when the frame window is minimized or hidden. Owned windows hidden in this way are restored when the frame window is restored.

If an application needs the same special processing for its own window classes, it must provide that support in the window procedures for those classes.

4.3.3.3 Object Windows

Any descendant of the desktop-object window is called an object window. An object window is like any other window but it is not displayed. Applications typically use object windows to provide services for windows. For example, an application might use an object window to manage a shared database. The advantage of using an object window in this way is that a window can request information from the database by sending a message to the object window and receive a reply as a message.

Because object windows are not displayed, the window procedure for an object window does not have to process input and paint messages. This means that an application can use object windows just as it would other objects in object-oriented environments. The object window processes messages that affect the data belonging to the object.

The rules for parent-child relationship and ownership also apply to object windows. In particular, changing the parent window of an object window to the desktop window or to a descendant of the desktop window causes the system to display the window if it is visible.

4.3.4 Visibility

A window can be visible or invisible. The system displays visible windows on the screen. It hides invisible windows by not drawing them. If a window is visible, the user can supply input to the window and view output. If a window is invisible, the window is effectively disabled. An application sets a window's visibility state when it creates the window. Later, a user or the application can change these initial values.

A window is visible if the style WS_VISIBLE is set for the window. An application can set this style when it creates the window. By default, the **WinCreateWindow** function creates invisible windows unless the WS_VISIBLE style is given. After a window is created, an application typically hides a window to hide the details of operation from the user. For example, an application may keep a new window invisible while it customizes the window's appearance.

Even if a window is visible, the user may not be able to see the window on the screen. Other windows may completely overlap the window or the window may have been moved beyond the edge of the screen. The window is considered visible but it cannot be seen.

A visible window is subject to the clipping rules established by its parent-child relationship. If the window's parent window is not visible, the window will not be visible. Since a child window is drawn relative to the parent's lower-left corner, if the parent window is moved beyond the edge of the screen, the child window will also move.

A user may move only part of the parent window containing the child window off the edge of the screen, so although the window and its parent window are visible, the user may not be able to see them. An application determines whether the user can actually see a visible window by checking the window's current position.

4.3.5 Size

Every window has a size (width and height) given in pels. The size can be any integer value in the range 0 through 65,535. A window can have zero width and/or height. A window with zero width or height is not drawn on the screen even though it may be visible.

Although an application can create very large windows, it should consider the size of the screen when choosing a window size. One way to choose an appropriate size is to use the **WinGetMaxPosition** function to retrieve the size of the maximized window. A window that is larger than its maximized size will also be larger than the screen.

An application can retrieve the current size of the window by using the **WinQueryWindowRect** function.

4.3.6 Position

Every window has a position. The position is specified as the coordinates of the window's lower-left corner. The coordinates, sometimes called window coordinates, are always relative to the lower-left corner of the parent window.

To improve drawing performance, a frame window may adjust its horizontal position so that it is a multiple of 8, relative to the screen origin (the lower-left corner of the screen). Coordinates that are multiples of 8 correspond to byte boundaries in the screen-memory bitmap. It is usually faster to draw starting at a byte boundary. An application can override this action by using the FCF_NOBYTEALIGN style when creating the window.

4.3.6.1 Size and Position Messages

A window receives messages when it changes size or position. Before a change is actually made, the system may send a WM_ADJUSTWINDOWPOS message to allow the window procedure to make final adjustments to the window's size and position. This message includes an **SWP** structure that contains the width, height, and position requested. If the window procedure adjusts these values in the structure, the system uses the adjusted values to draw the new window. The WM_ADJUSTWINDOWPOS message is not sent if the change is a result of a call to the **WinSetWindowPos** function and the SWP_NOADJUST constant is specified.

After a change has been made to a window, the system sends a WM_SIZE message to specify the new size of the window. If the window has the class style CS_MOVENOTIFY, the system also sends a WM_MOVE message. The WM_MOVE message includes the new position for the window. The system sends a WM_SHOW message if the visibility of the window has changed.

4.3.7 Z Order

Every window has a Z-order position. Imagine an axis extends outward from the screen toward the viewer. A window at the top of the Z order is displayed in front of its sibling windows when the windows overlap. A window at the bottom of the Z order is displayed behind its sibling windows when the windows overlap.

4.3.8 Maximized and Minimized Windows

A maximized window is a window that has been enlarged so it fills the screen. Although a window's size can be set so it exactly fills the screen, a maximized window is slightly different—the system automatically moves the window's title bar to the top of the screen and sets the WS_MAXIMIZED window style.

A minimized window is a window whose size has been reduced so that it is exactly the size of an icon. Like a maximized window, a minimized window is more than just a window of a given size. The system typically moves the minimized window to the lower part of the screen and sets the WS_MINIMIZED style for that window. The lower part of the screen is sometimes call the icon area. The system moves a minimized window into the first available icon position in the icon area if no other position is specified.

If a window is created with the styles WS_MAXIMIZED or WS_MINIMIZED, the system draws the window as a maximized or minimized window.

An application can restore a maximized or minimized window to its previous size and position.

4.3.9 Redrawing Windows

After the system moves or changes the size of a window, it may invalidate all or part of the window. If at all possible, the system tries to preserve the contents of the window and simply copy them to the new position. But if a window's size has increased, the window must fill the area exposed by the size change. If a window has moved from behind an overlapping window, any area that was formerly obscured by the other window must be drawn. In these cases, the system invalidates the exposed areas and the window receives a WM_PAINT message.

An application can require that the system invalidate the entire window for each move or size change by setting the CS_SIZEREDRAW class style in the corresponding window class. This class style is typically used for applications that use the window's current size and position to determine how to draw the window. For example, a clock application may always draw the face of the clock so that it exactly fills the window.

An application can also explicitly specify which parts of the window to preserve during a move or size change. Before any change is made, the system sends a WM_CALCVALIDRECTS message to windows that do not have the style CS_SIZEREDRAW. This allows the window procedure to specify what part of the window to save and where to align it after the move or size change.

4.3.10 System Commands

An application that has a window with a System menu can change the size and position of that window by sending system commands. The system commands are usually generated by the user choosing commands from the System menu. An application can emulate the user action by sending a WM_SYSCOMMAND message to the window.

Some of the system commands are listed here:

Command	Description
SC_SIZE	Starts a size command. The user can change the size of the window by using the mouse or keyboard.
SC_MOVE	Starts a move command. The user can move the window by using the mouse and keyboard.
SC_MINIMIZE	Minimizes the window.
SC_MAXIMIZE	Maximizes the window.
SC_RESTORE	Restores a minimized or maximized window to its previous size and position.
SC_CLOSE	Closes the window. This command sends a WM_CLOSE message to the window. The window carries out any steps needed to clean up and destroy itself.

4.4 Using Windows

The following sections explain how to create and use windows in an application, how to manage ownership and parent-child window relationships, and how to move and size windows.

4.4.1 Creating a Window

You create windows by using the **WinCreateWindow** function.

For all windows, the parent-child relationship is set when you create the window using the **WinCreateWindow** function or other window-creation function. You can set the ownership for a window at any time. (Note that a window does not need an owner window unless you want to establish a relationship other than the standard parent-child relationship for the window.)

You can specify the initial size and position for a window when you create it. You can change these settings at any time.

4.4.2 Creating a Frame Window

Although **WinCreateWindow** can be used to create all windows, most applications do not call this function. Instead, they use the **WinCreateStdWindow** function to create frame windows and the **WinDlgBox** or **WinCreateDlg** function to create dialog windows.

4.4.3 Destroying a Window

You can destroy a window by using the **WinDestroyWindow** function. The following code fragment shows how to create and then destroy an entry-field control:

```
HWND hwndMain;   /* application's main window */
HWND hwnd;

hwnd = WinCreateWindow(...);

/* Read from the control. */

WinDestroyWindow(hwnd);
```

4.4.4 Setting and Querying Window Data

You can examine the data associated with a window by using the **WinQueryWindowUShort** and **WinQueryWindowULong** functions.

Each of these functions specifies a field to examine. The index value can be an integer representing a zero-based index or a constant (QW_) that specifies a specific field.

4.4.5 Creating a Top-Level Window

You can create a top-level window by setting the desktop window as the window's parent window. Almost all main windows for applications are top-level windows; the desktop window is frequently given in calls to the **WinCreateStdWindow** function.

The following code fragment creates a top-level window for an application:

```
/* Set the creation flags. */

ULONG flCreationFlags =
    FCF_TITLEBAR |          /* title bar                      */
    FCF_SIZEBORDER |        /* size border                    */
    FCF_MINMAX |            /* minimize and maximize buttons  */
    FCF_MENU |              /* menu                           */
    FCF_SYSMENU |           /* System menu                    */
    FCF_HORZSCROLL |        /* horizontal scroll bar          */
    FCF_VERTSCROLL; |       /* vertical scroll bar            */

/*
 * Create a frame window with a client window that belongs to the
 * window class "MyPrivateClass".
 */

hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,           /* owner is desktop window    */
    OL,                     /* no styles for frame window */
    &flCreationFlags,       /* frame controls             */
    "MyPrivateClass",       /* window class for client    */
    "Sample Window",        /* window title               */
    OL,                     /* no styles for client       */
    NULL,                   /* use application's module   */
    1,                      /* resource ID                */
    &hwndClient);           /* client handle              */
```

4.4.6 Creating an Object Window

You can create an object window by using the **WinCreateWindow** function and setting the desktop-object window as the parent window.

The following code fragment creates an object window:

```
hwndObject = WinCreateWindow(
    HWND_OBJECT,         /* parent is object window      */
    "MyPrivateClass",    /* window class for client      */
    "Sample Window",     /* window title                 */
    OL,                  /* no styles for object window  */
    O, O,                /* lower-left corner            */
    O, O,                /* width and height             */
    NULL,                /* no owner                     */
    HWND_TOP,            /* insert at top of Z order     */
    1,                   /* window ID                    */
    NULL,                /* no class-specific data       */
    NULL);               /* no presentation data         */
```

4.4.7 Changing the Parent Window

You can change a window's parent window by using the **WinSetParent** function. For example, an application that uses child windows to display documents may want only the active-document window to show a System menu. One way to do this is to change that menu's parent window back and forth between the document window and the object window when WM_ACTIVATE messages are received. This is shown in the following code fragment:

```
case WM_ACTIVATE:
    hwndMenu = WinWindowFromID(hwnd, FID_MENU);
    if (SHORT1FROMMP(mp1) == TRUE)
        WinSetParent(hwndMenu, hwnd, TRUE);
    else
        WinSetParent(hwndMenu, HWND_OBJECT, TRUE);
```

4.4.8 Finding a Parent, Child, or Owner Window

You can determine the parent, child, and owner windows for any window by using the **WinQueryWindow** function. The function returns the window handle of the requested window. It can also lock that window. If a window is locked, it must be unlocked by using the **WinLockWindow** function.

The following code fragment determines the parent window of the given window (it does not lock the parent window):

```
HWND hwndParent;

hwndParent = WinQueryWindow(hwnd, QW_PARENT, FALSE);
```

The following code fragment determines the topmost child window and locks it:

```
HWND hwndChild;

if (hwndChild = WinQueryWindow(hwnd, QW_TOP, TRUE)) {

    /* Lock the child window. */

    WinLockWindow(hwndChild, FALSE);
}
```

If a given window does not have an owner or child window, the function returns NULL.

4.4.9 Setting an Owner Window

You can set the owner for a window by using the **WinSetOwner** function. After setting the owner, a window typically notifies the owner window of the new relationship by sending a message.

The following code fragment shows how to set the owner window and send it a message:

```
#define NEW_OWNER   1
HWND hwnd;                                   /* window to get new owner    */
HWND hwndOwner;                              /* window to become new owner */

if (WinSetOwner(hwnd, hwndOwner)) {

    /* Send a notification message. */

    WinSendMsg(hwndOwner,          /* send to owner                     */
        WM_CONTROL,                /* control message for notification */
        MAKELONG(NEW_OWNER, 1),    /* notification code and ID          */
        NULL);                     /* no extra data                     */
}
```

A window can have only one owner, so **WinSetOwner** removes any previous owner.

4.4.10 Finding a Child or Owned Window

A parent or owner window can retrieve the handle of a child or owned window by using the **WinWindowFromID** function and supplying the identifier of the child or owned window. **WinWindowFromID** searches all child and owned windows to locate the window having the given identifier. The window identifier is set when the application creates the child or owned window.

An owner window typically uses the **WinWindowFromID** function to respond to a notification message from an owned window.

The following code fragment retrieves the window handle of the owned window having the window identifier 1:

```
hwndOwned = WinWindowFromID(hwndOwner, 1);
WinSendMsg(hwndOwned, WM_ENABLE, MPFROM2SHORT(0, TRUE), NULL);
```

You can also retrieve the handle of a child window by using the **WinWindowFromPoint** function and supplying a point in the corresponding parent window.

4.4.11 Enumerating Top-Level Windows

You can enumerate all top-level windows by using the **WinBeginEnumWindows** and **WinGetNextWindow** functions. An application can create a list of all child windows for a given parent window by using the **WinBeginEnumWindows** function. This list contains the window handles of immediate child windows. An application can retrieve, one at a time, the window handles from the list using the **WinGetNextWindow** function. When the application has finished using the list, it must release it by using the **WinEndEnumWindows** function.

The following code fragment shows how to enumerate all top-level windows (all immediate child windows of the desktop window):

```
/* Enumerate all top-level windows. */

henum = WinBeginEnumWindows(HWND_DESKTOP);

/*
 * Loop through all enumerated windows, performing the desired task
 * on each one.
 */

while (hwnd = WinGetNextWindow(henum)) {
    .
    . /* Lock the window. */
    .
    WinLockWindow(hwnd, FALSE); /* unlock window when done     */
}

/* Return memory required for enumeration back to the system. */

WinEndEnumWindows(henum);
```

4.4.12 Moving and Sizing a Window

You can move a window by using the **WinSetWindowPos** function and specifying the SWP_MOVE constant. The function changes the position of the window to the specified position. The position is always given as coordinates relative to the parent window.

The following code fragment moves the window to the position (10,10):

```
WinSetWindowPos(
    hwnd,       /* window handle                */
    NULL,       /* not used for moving and sizing */
    10, 10      /* new position                 */
    0, 0,       /* not used for moving          */
    SWP_MOVE);  /* move and size                */
```

You can set the size of a window by using the **WinSetWindowPos** function and specifying the SWP_SIZE constant. The function changes the width and height of the window to the specified width and height.

You can combine moving and sizing in a single function call, as shown in the following code fragment:

```
WinSetWindowPos(
    hwnd,                 /* window handle                */
    NULL,                 /* not used for moving and sizing */
    10, 10                /* new position                 */
    200, 200,             /* width and height             */
    SWP_MOVE | SWP_SIZE); /* move and size                */
```

You can retrieve the current size and position of a window by using the **WinQueryWindowPos** function. This function copies the current information to an **SWP** structure.

The following code fragment uses the current size and position to change the height of the window, but leaves the width and position unchanged:

```
SWP swpCurrent;

WinQueryWindowPos(hwnd, &swpCurrent);
WinSetWindowPos(
    hwnd,                    /* window handle                */
    NULL,                    /* not used for moving and sizing */
    0, 0,                    /* not used for sizing          */
    swpCurrent.cx,           /* current width                */
    swpCurrent.cy + 200,     /* new height                   */
    SWP_SIZE);               /* change the size              */
```

You can also move and change the size of several windows at once by using the **WinSetMultWindowPos** function. This function takes an array of **SWP** structures. Each structure specifies the window to be moved or changed.

4.4.13 Moving a Window in a Stack of Windows

You can move a window to the top or bottom of the Z order by passing the SWP_ZORDER constant to the **WinSetWindowPos** function. You specify where to move the window by specifying HWND_TOP or HWND_BOTTOM.

The following code fragment uses **WinSetWindowPos** to reorder a stack of child windows:

```
HENUM henum;
HWND hwndParent;
HWND hwndNext;

henum = WinBeginEnumWindows(hwndParent);

while (hwndNext = WinGetNextWindow(henum)) {
    WinSetWindowPos(
        hwndNext,       /* next window to move  */
        HWND_TOP,       /* put window on top    */
        0, 0, 0, 0,     /* not used for Z order */
        SWP_ZORDER);    /* change Z order       */

    WinLockWindow(hwndNext, FALSE);     /* unlock window */
    .
    . /* Wait a little before doing the next window. */
    .
}
WinEndEnumWindows(henum);
```

You can also specify the window you want the given window to move behind. In this case, you specify the window handle instead of the HWND_TOP or HWND_BOTTOM constant.

If you enumerate windows as shown in the previous code fragment, the following code fragment will reverse the order of every other pair of windows:

```
hwndExchange = WinGetNextWindow(henum);

/* hwndNext has top window, hwndExchange has window under the top */

WinSetWindowPos(
    hwndNext,       /* next window to move     */
    hwndExchange,   /* put lower window on top */
    0, 0, 0, 0,     /* not used for Z order    */
    SWP_ZORDER);    /* change Z order          */
```

4.4.14 Showing and Hiding a Window

Moving and sizing a window still applies if a window is not visible. The effects of moving and sizing cannot be seen until the window is visible. You can show and hide a window by using the **WinShowWindow** function. This function changes the WS_VISIBLE style for a window to the specified setting. You can also use the **WinIsWindowVisible** function to check the visibility of a window. The function returns TRUE if the window is visible.

4.4.15 Maximizing, Minimizing, and Restoring a Window

You can maximize, minimize, or restore a frame window by using the **WinSetWindowPos** function and specifying the constant SWP_MAXIMIZE, SWP_MINIMIZE, or SWP_RESTORE. Only a frame window can maximize and minimize by default. For any other window, you must provide support for these actions in the corresponding window procedure.

The following code fragment shows how to maximize a frame window:

```
SWP swpCurrent;

WinQueryWindowPos(hwnd, &swpCurrent);
WinSetWindowPos(
    hwnd,               /* window handle             */
    NULL,               /* not used to maximize      */
    swpCurrent.x,
    swpCurrent.y,       /* stored for restoring window */
    swpCurrent.cx,
    swpCurrent.cy,      /* stored for restoring window */
    SWP_MAXIMIZE | SWP_SIZE | SWP_MOVE);    /* maximize */
```

4.5 Summary

The following sections list all the functions and messages an application can use to create, maintain, and destroy windows; to manage window relationships; and to set, query, and initialize the size, position, and visibility of windows.

4.5.1 Window Functions

The following functions are used by an application to create, maintain, and destroy windows:

WinCreateWindow Creates a window. This is the most flexible, general purpose window-creation function. It can be used to create windows of any class. The function's parameters let you specify the window class, the parent window, the owner window, the window size and position, the Z-order position, the window identifier, general and class-specific window styles, and additional class-specific data. Other window-creation functions make one or more calls to this function to create their window(s).

WinDestroyWindow Destroys a window. Related child windows and owned windows will also be destroyed.

WinEnableWindow Enables or disables a window. A disabled window loses the focus and ignores input. This function clears or sets the WS_DISABLED style.

WinIsWindow Determines if a window handle is valid.

WinIsWindowEnabled Determines whether a window is enabled or disabled. The function tests the WS_DISABLED style.

WinLockWindow Increments or decrements a window lock count. The lock count is initialized to zero and must be zero for the window to be destroyed.

WinQuerySysModalWindow Determines the system-modal window. This function returns the system-modal window handle if successful or NULL if there is no system-modal window.

WinQueryWindowLockCount Retrieves the window lock count.

WinQueryWindowPtr Examines a pointer in a window data structure.

WinQueryWindowULong Examines a 32-bit field in a window data structure.

WinQueryWindowUShort Examines a 16-bit field in a window data structure.

WinRegisterWindowDestroy Notifies other applications when the specified window is destroyed.

WinSetSysModalWindow Sets a window as the system-modal window or ends the system-modal state. This function should be called only while processing keyboard or mouse input.

WinSetWindowBits Sets bits in a 32-bit field in a window data structure.

WinSetWindowPtr Sets a pointer in a window data structure.

WinSetWindowULong Sets a 32-bit field in a window data structure.

WinSetWindowUShort Sets a 16-bit field a window data structure.

WinSubclassWindow Changes a window procedure. If successful, the function returns a pointer to the previous window procedure.

4.5.2 Standard Window Messages

The following are standard window messages:

WM_CREATE Sent to a window during processing of the **WinCreateWindow** function, before the window is sized, positioned, or shown.

WM_DESTROY Sent to the window being destroyed. This message is sent after the window has been hidden on the device but before its child windows have been destroyed.

WM_ENABLE This message is sent when a window is being enabled.

WM_OTHERWINDOWDESTROYED Sent to all top-level windows when a window is destroyed.

WM_QUERYWINDOWPARAMS Sent to obtain certain window data. The data is specified and returned by using a **WNDPARAMS** data structure.

WM_SETWINDOWPARAMS Sent to set window data.

4.5.3 Relationship Functions

The following functions can be used to manage window relationships:

WinBeginEnumWindows Begins the window-enumeration process. This function creates an enumeration list of the immediate child windows of a window and returns the list handle.

WinEndEnumWindows Ends a window enumeration process. This function destroys the enumeration list (just the list, not the windows) created by the **WinBeginEnumWindows** function.

WinGetNextWindow Obtains a window handle from a window list created by a call to the **WinBeginEnumWindows** function. Each call returns the next window in the list or NULL at the end of the list. The function locks the window that has the returned handle. The application must unlock the window after processing. Calling this function after it has returned NULL causes it to wrap around to the beginning of the list.

WinIsChild Determines if one window is the child of another window.

WinQueryDesktopWindow Obtains a handle of the desktop window HWND_DESKTOP.

WinQueryObjectWindow Obtains a handle of the desktop-object window HWND_OBJECT.

WinQueryWindow Retrieves a window's parent, owner, or child windows. This function returns the handle of the specified window or NULL if no such window exists.

WinSetOwner Sets a window's owner window. Note that you can set the handle of the owner window to NULL, meaning it has no owner.

WinSetParent Sets a window's parent window. This allows you to change object windows to regular windows, descendant windows of top-level windows to top-level windows, and vice versa.

4.5.4 Functions for Moving, Sizing, and Changing

The following functions can be used to move and change the position of a window:

WinGetMaxPosition Obtains a window's maximized size and position. A window's maximized size is the size of its parent window plus an adjustment outward in all four directions equal to the size of its border. The adjustment is made because maximized windows do not show their border.

WinGetMinPosition Obtains an icon location for a minimized window. The function searches for an icon area, starting at the given point and continuing with subsequent positions until the next available icon position is found.

WinIsWindowVisible Determines whether a window is visible.

WinQueryWindowPos Determines a window's current size and position.

WinQueryWindowRect Determines a window's bounding rectangle, relative to its parent window. You can determine the window's size and position from the bounding rectangle.

WinSetWindowPos Sets a window's size, position, and Z order. Position is specified in window coordinates relative to the parent window's lower-left corner. Size is specified in device units. Z order is relative to a window's sibling windows.

WinSetMultWindowPos Sets the size, position, and Z order for an array of windows. This function and the **WinSetWindowPos** function are the same except for the number of windows each can affect.

WinShowWindow Makes a window visible or invisible.

4.5.5 Messages for Moving, Sizing, and Changing

The following messages are received by a window procedure when the corresponding window is moved or changes size or visibility:

WM_ADJUSTWINDOWPOS Sent to a window about to be moved or sized. This message allows the window to adjust the new size and position before they take effect. This message is sent, by default, during calls to the **WinCreateWindow**, **WinSetWindowPos**, and **WinSetMultWindowPos** functions.

WM_CALCVALIDRECTS Sent from the **WinSetWindowPos** and **WinSetMultWindowPos** functions when a window is about to be resized. Both these functions determine if there is a rectangular area of the window that can be preserved across the size change. If such a valid rectangle exists, its bitmap data can be simply and quickly copied to the new window image. Handlers of this message can specify the coordinates of the valid rectangle to be preserved, as well as determine where the valid rectangle will be placed within the resized window.

WM_MOVE Sent by the system when a window with CS_MOVENOTIFY style changes its absolute (relative to the screen) position. The window's new position can be obtained by calling the **WinQueryWindowPos** function.

WM_SHOW Sent by the system after a window's WS_VISIBLE style bit has changed. An *mp1* value of TRUE indicates an invisible window has become visible. An *mp1* value of FALSE indicates a visible window has become invisible. The default window procedure takes no action on this message.

WM_SIZE Sent after a window has changed size, but before any repainting has been performed. Resizing or repositioning of child windows resulting from the size change usually occurs during the processing of this message. This message is not sent when a window is created.

Chapter 5

Messages and Message Queues

5.1 Introduction 79

5.2 About Messages and Message Queues 79
- 5.2.1 Messages 79
- 5.2.2 Message Queues 80
- 5.2.3 Message Loop 81
- 5.2.4 Messages and Window Procedures 82
- 5.2.5 Application Messages 83
- 5.2.6 System-Defined Messages 84
- 5.2.7 Application-Defined Messages 85
- 5.2.8 Semaphore Messages 85
- 5.2.9 Message Priorities 85
- 5.2.10 Message Filtering 86

5.3 Using Messages in an Application 86
- 5.3.1 Creating a Message Queue and Message Loop 86
- 5.3.2 Examining the Message Queue 87
- 5.3.3 Posting a Message to a Window 88
- 5.3.4 Sending a Message to a Window 88
- 5.3.5 Broadcasting a Message 88
- 5.3.6 Using Message Macros 89

5.4 Summary 89

5.1 Introduction

This chapter describes creating and using messages and message queues in MS OS/2 Presentation Manager applications. You should also be familiar with the following topics:

- Windows
- Window procedures
- Threads, processes, and sessions

5.2 About Messages and Message Queues

Unlike traditional applications that take complete control of the computer's keyboard, mouse, and screen, Presentation Manager applications must share these resources with other applications running at the same time. Because all applications run independently, Presentation Manager applications rely on MS OS/2 to help them manage shared resources. The system manages shared resources by controlling the operation of each application, communicating with each application when there is keyboard and mouse input or when an application must move and size its windows. The system uses messages to communicate with an application and the windows belonging to that application.

A message is information, a request for information, or a request for an action to be carried out by the application. The system communicates a message to an application so that the application can use the information or respond to the request. The system communicates in two ways: posting and sending.

The system posts a message to an application's message queue if the message represents information or a request that does not need immediate action. The message queue is an application-created storage area used to hold messages. The application can then retrieve and process a message at the appropriate time. The system posts a message by copying the message data to the message queue.

The system sends a message to an application when it needs an immediate response from the application. It sends a message by passing the message data as arguments to the window procedure. The window procedure carries out the request or lets the system carry out default processing for the message.

The following sections describe messages and message queues in detail.

5.2.1 Messages

All messages contain information that an application uses to carry out tasks. There are two types of messages: queue messages and window messages. Queue messages are messages stored in a message queue. Window messages are messages sent to a window procedure. Although these message types have very different formats, the information they contain is nearly identical.

Every message contains a message identifier. The message identifier is an integer value that determines whether the message is information or a request. When an application processes a message, it uses the message identifier to determine what to do.

Every message contains a window handle. The window handle identifies the window for which the message is intended. The window handle is important because most message queues and window procedures serve more than one window. The window handle ensures that the application processes the message for the appropriate window.

Messages contain two message parameters. A message parameter is a 32-bit value that specifies data or the location of data to be used in processing the message. The meaning and value of a message parameter depends on the message. Message parameters can be pointers to structures containing additional data, integer values, packed bit flags, and so on. Some messages do not use message parameters and typically set the parameters to zero. An application always checks the message identifier to determine how to interpret the message parameters.

Queue messages also contain the message time and mouse position. The message time specifies the system time, in milliseconds, when the message was created. The mouse position specifies the location of the mouse pointer, in screen coordinates, when the message was created.

A queue message is a **QMSG** data structure that contains six fields representing the window handle, message identifier, two message parameters, message time, and mouse position. The time and position are provided because most queue messages are input messages, representing keyboard or mouse input from the user. The time and position help the application identify the context of the message. The system posts a queue message by filling the **QMSG** structure and copying it to a message queue.

A window message consists of the window handle, the message identifier, and two message parameters. A window message does not include the message time and mouse position because most window messages are requests to carry out a task that is not related to the current time or mouse-pointer position. The system sends a window procedure by passing these values as individual arguments to a window procedure.

5.2.2 Message Queues

Every Presentation Manager application needs a message queue. A message queue is the only means an application has to receive input from the keyboard or mouse. Only applications that create message queues can create windows.

A message queue is internal storage reserved by the application for receiving and holding posted messages. An application creates a message queue by using the **WinCreateMsgQueue** function. This function returns a handle the application can use to access the message queue. After an application creates a message queue, the system posts messages intended for windows in the application to that queue. The application can retrieve queue messages by specifying the message-queue handle in a call to the **WinGetMsg** function. It can examine messages without retrieving them by using the **WinPeekMsg** function. When an application no longer needs the message queue, it can destroy it by using the **WinDestroyMsgQueue** function.

Message queues serve all windows created by the application. This means a queue may hold messages for several windows. Most messages specify the windows to which they belong, so the application can easily apply a message to the appropriate window. Messages that do not specify a window apply to the entire application.

An application that has more than one thread can create more than one message queue. The system allows one message queue for each thread. A message queue created by an application thread belongs to that thread and has no connection to other queues in that application. When an application creates a window in a given thread, the system associates the window with the message queue in that thread. The system then posts all subsequent messages intended for the window to that queue.

Although multiple messages queues are possible, most Presentation Manager applications use threads sparingly and so use only one message queue.

Since several windows typically use a message queue, it is important that the message queue be large enough to hold all possible messages that may be posted to it. An application can set the size of the message queue when it creates the queue by specifying the maximum number of messages the queue can hold.

To minimize the queue size, several types of posted messages are not actually stored in a message queue. Instead, the system keeps a record in the queue of the message being posted and combines any information contained in the message with information from previous messages. Timer, semaphore, and paint messages are handled in this way. For example, if more than one WM_PAINT message is posted, the system combines the update regions for each into a single update region. Although there is no actual message in the queue, the system constructs one WM_PAINT message with the single update region when an application uses the **WinGetMsg** function.

Mouse and keyboard input messages are also not stored in the message queue. These are stored in the system message queue. The system message queue is a system-owned queue that receives and holds messages for all mouse and keyboard input. The system does not copy these messages to application message queues. Instead, the **WinGetMsg** function searches the system queue for input messages belonging to the application when there are no other higher-priority messages in the application's message queue. The system message queue is usually large enough to hold all input messages, even if the user is typing or moving the mouse very quickly. If the system queue does run out of room, the system ignores the most recent keyboard input (usually beeping to indicate it is ignored) and collects mouse motions into a single motion.

Every message queue has a corresponding data structure. The data structure specifies the identifiers of the process and thread that own the message queue and gives a count of the maximum number of messages the queue can receive. An application can retrieve the data structure by using the **WinQueryQueueInfo** function.

A message queue also has a current status. The status specifies whether any messages are available in the queue. An application can retrieve the queue status by using the **WinQueryQueueStatus** function. Since this function is very fast, applications typically use it to check for messages rather than using the **WinPeekMsg** function, which inspects the thread's message queue.

5.2.3 Message Loop

Every application with a message queue is responsible for retrieving the messages from that queue. An application can do this by using a message loop. A message loop is a program loop, usually in the application's main function, that retrieves messages from the message queue and dispatches them to the appropriate windows. The message loop consists of two function calls: one to the **WinGetMsg** function, the other to the **WinDispatchMsg** function. The message loop has the following form:

```
while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
    WinDispatchMsg(hab, &qmsg);
```

An application starts the message loop after creating the message queue and at least one application window. Once started, the message loop continues to retrieve messages from the message queue and to dispatch (send) them to the appropriate windows. The **WinDispatchMsg** function sends each message to the window specified by the window handle in the message.

Only one message loop is needed for a message queue, even if the queue contains messages for more than one window. Each queue message is a **QMSG** structure that contains the handle of the window to which the message belongs, so the **WinDispatchMsg** function always dispatches the message to the proper window. The **WinGetMsg** function retrieves messages from the queue in first-in, first-out (FIFO) order, so the messages are dispatched to windows in the same order they were put in the queue.

If there are no messages in the queue, the system temporarily stops processing the **WinGetMsg** function until a message arrives. This means that CPU time slices that would otherwise be spent waiting for a message can be given to the applications (or threads) that do have messages in their queues.

The message loop continues to retrieve and send messages until the **WinGetMsg** function retrieves a WM_QUIT message. This message causes the function to return FALSE, terminating the loop. In most cases, terminating the message loop is the first step in terminating the application. An application can terminate its own loop by posting the WM_QUIT message in its own queue.

An application can modify its message loop in a variety of ways. For example, it can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages that do not specify a window (these messages apply to the application rather than to a specific window; they have NULL window handles). An application can also direct the **WinGetMsg** function to search for specific messages, leaving other messages in the queue. This is useful for applications that temporarily need to bypass the usual first-in, first-out order of the message queue.

5.2.4 Messages and Window Procedures

When the system needs an immediate response from an application, it sends a message to a window procedure. A window procedure is a function that receives and processes all input and requests for action sent to the window by the system. Every window class has a window procedure and every window created using that class uses the window procedure to respond to messages.

The system sends a message to the window procedure by passing the message data as arguments to the window procedure. The window procedure carries out an appropriate action for the given request. Most window procedures check the message identifier, then use the information specified by the message parameters to carry out the request. When it has completed processing the message, the window procedure returns a message result. Each message has a particular set of possible return values. The window procedure must return the appropriate value for the processing it carried out.

A window procedure cannot ignore a message. If it does not process a message, it must pass the message back to the system for default processing. The window procedure can do this by calling the **WinDefWindowProc** function. This function carries out a default action and returns the message result. The window procedure must return this value as its own message result.

A window procedure commonly processes messages for several windows. It uses the window handle specified in the message to identify the appropriate window. Most window procedures process a few types of messages and pass the others on to the system by calling the **WinDefWindowProc** function.

5.2.5 Application Messages

Any application can post and send messages. Like the system, an application posts a message by copying it to a message queue. It sends a message by passing the message data as arguments to a window procedure. An application can post a message by using the **WinPostMsg** function. It can send a message by using the **WinSendMsg** function.

Typically, an application posts a message to notify a specific window to carry out a task. The **WinPostMsg** function creates a **QMSG** structure for each message and copies the message to the message queue corresponding to the given window. The application's message loop eventually retrieves the message and dispatches it to the appropriate window procedure. One message commonly posted is WM_QUIT. This message terminates the application by terminating the message loop.

Typically, an application sends a message to notify a specific window procedure to immediately carry out a task. The **WinSendMsg** function passes the message to the window procedure corresponding to the given window. The function waits until the window procedure completes processing and then returns the message result. It is very common for parent and child windows to communicate by sending messages to each other. For example, a parent window that has an entry-field control (as its child window) can set the text of the control by sending the child window a message. The control can notify the parent window of changes to the text (carried out by the user) by sending messages back to the parent window.

Occasionally, an application may need to send or post a message to all windows in the system. For example, if the application changes a system value, it must notify all windows about the change by sending a WM_SYSVALUECHANGED message. An application can send or post messages to any number of windows by using the **WinBroadcastMsg** function. The options in **WinBroadcastMsg** determine whether the message is sent or posted and specify the number of windows to receive the message.

If an application has more than one thread, any thread in the application can post messages to a message queue, even if that thread has no message queue of its own. However, only threads that have a message queue can send messages. Posting or sending messages between threads is relatively uncommon. One reason for this is that it is costly in terms of system performance to send a message. If you do post messages between threads, it is likely to be for semaphore messages. Semaphore messages permit window procedures to jointly manage a shared resource.

An application can post a message without specifying a window. If the application supplies a NULL window handle when it calls the **WinPostMsg** function, the function posts the message that is in the message queue of the thread calling the function. Because the message has no window handle, the message loop processes the message. This is one way to create messages that apply to the entire application instead of to a specific window.

A window procedure can determine whether it is processing a message sent by another thread by using the **WinInSendMsg** function. This is useful when message processing depends on the origin of the message.

A common programming error is to assume that the **WinPostMsg** function always posts a message. This is not true when the message queue is full. An application should check the return value of the function to see if the message has been posted. In general, if an application intends to post many messages to the queue, it should set the message queue to an appropriate size when it creates the queue. The default message-queue size is ten messages.

5.2.6 System-Defined Messages

There are many system-defined messages. The system uses these messages to control the operation of applications and to provide input and other information for applications to process. The system sends or posts a system-defined message when it communicates with an application. An application can also send or post system-defined messages. Applications typically use these messages to control the operation of control windows created using the preregistered window classes.

Each system message has a unique message identifier and a corresponding symbolic constant. The symbolic constant, defined in MS OS/2 header files, typically states the purpose of the message. For example, the WM_PAINT constant represents the paint message. The paint message requests a window to paint its contents.

The symbolic constants also specify the message category. System-defined messages can belong to several categories; the prefix identifies the type of window that can interpret and process the messages. The following list gives the prefixes and related message categories:

Prefix	Message category
BM	Button-control
EM	Entry-field control
LM	List-box control
MM	Menu
SBM	Scroll-bar control
SM	Static control
TBM	Title-bar control
WM	General window

General window messages cover a wide range of information and requests and include mouse and keyboard-input messages, menu and dialog-input messages, window-creation and window-management messages, and dynamic-data-exchange (DDE) messages.

5.2.7 Application-Defined Messages

An application can create its own messages to use in its own windows. If an application creates messages, the window procedure that receives the message must interpret the message and provide appropriate processing.

MS OS/2 reserves the message-identifier values in the range 0x0000 through (WM_USER–1) for system-defined messages. Applications cannot use these values for private messages. Values in the range WM_USER through 0xBFFF are available for message identifiers defined by an application for use in that application. Values in the range 0xC000 through 0xFFFF are reserved for message identifiers defined by an application using the atom-manager registration for use in any application.

5.2.8 Semaphore Messages

The semaphore messages are a way of signaling the end of an event through the message queue. Applications use these messages like they use MS OS/2 semaphore functions to coordinate events by passing signals. Semaphore messages are often used in conjunction with MS OS/2 semaphores.

There are four semaphore messages: WM_SEM1, WM_SEM2, WM_SEM3, and WM_SEM4. An application posts one of the semaphore messages to signal the end of the given event. The window that is waiting for the given event receives the semaphore message when the message loop retrieves and dispatches the message.

Each semaphore message includes a bit flag that can be used to uniquely identify 32 possible semaphores for each semaphore message. The application passes the bit flag (with the appropriate bit set) as a message parameter with the message. The window procedure that receives the message then uses the bit flag to identify the semaphore.

To save space in a message queue, the system does not store semaphore messages in the message queue. Instead, it sets a record in the queue, indicating the semaphore message has been received, and then combines the bit flag for the message with the bit flags from previous messages. When the window procedure eventually receives the message, the bit flag specifies each semaphore message posted since the last message was retrieved.

5.2.9 Message Priorities

The **WinGetMsg** function retrieves messages from the message queue based on message priority. The function retrieves messages with higher priority first. If it finds more than one message at a particular priority level, it retrieves the oldest message first. Messages have the following priority:

Priority	Message
1	WM_SEM1
2	Messages posted using **WinPostMsg**
3	Input messages from the keyboard or mouse

Priority	Message
4	WM_SEM2
5	WM_PAINT
6	WM_SEM3
7	WM_TIMER
8	WM_SEM4

5.2.10 Message Filtering

Applications can choose specific messages to retrieve from the message queue (ignoring other messages) by specifying a message filter with the **WinGetMsg** or **WinPeekMsg** function. The message filter is a range of message identifiers (specified by a first and last identifier), a window handle, or both. The functions use the message filter to select the messages to retrieve from the queue. Message filtering is useful if an application needs to search ahead in the message queue for messages that have a lower priority or that arrived in the queue later than other less important messages.

Any application that filters messages must ensure that a message satisfying the message filter can be posted. For example, filtering for a WM_CHAR message in a window that does not have the input focus prevents the **WinGetMsg** function from returning. Some messages, such as WM_COMMAND, are generated from other messages; filtering for them may also prevent **WinGetMsg** from returning.

The WM_BUTTONCLICKFIRST and WM_BUTTONCLICKLAST, WM_MOUSEFIRST and WM_MOUSELAST, and WM_DDE_FIRST and WM_DDE_LAST constants can be used to filter button, mouse, and DDE messages.

5.3 Using Messages in an Application

This section explains how to use the message and message-queue functions to create and manage message queues and to post and send messages between windows.

5.3.1 Creating a Message Queue and Message Loop

Your application needs a message queue and message loop to process messages for windows. You create a message queue by using the **WinCreateMsgQueue** function. You create a message loop by using the **WinGetMsg** and **WinDispatchMsg** functions. You must create and show at least one window after creating the queue but before starting the message loop. The window is required because it is the only way the user can supply input to the message queue.

The following code fragment shows how to create a message queue and a message loop:

```
HAB hab;        /* anchor-block handle     */
HMQ hmq;        /* message-queue handle    */
QMSG qmsg;      /* queue-message structure */

VOID cdecl main()
{
    hab = WinInitialize(NULL);
    hmq = WinCreateMsgQueue(hab, DEFAULT_QUEUE_SIZE);

    /*
     * Use WinRegisterClass to register your window class.
     * Use WinCreateStdWindow to create your window.
     */

    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))
        WinDispatchMsg(hab, &qmsg);

    /* Use WinDestroyWindow to destroy your window. */

    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
}
```

Both the **WinGetMsg** and **WinDispatchMsg** functions take a pointer to a **QMSG** structure as a parameter. If a message is available, **WinGetMsg** copies it to the **QMSG** structure; **WinDispatchMsg** then uses the fields of the structure as arguments for the window procedure.

Occasionally, you may need to process the message before dispatching it. This can occur, for example, if a window procedure posts a message to the queue for which a NULL window handle has been specified. Because the **WinDispatchMsg** function needs a window handle to dispatch the message, the message loop must process the message before dispatching it. The following code fragment shows how the message loop might process messages that have NULL window handles:

```
while (WinGetMsg(hab, &qmsg, NULL, 0, 0)) {
    If (qmsg.hwnd == NULL) {
        .
        . /* Process the message. */
        .
    }
    else
        WinDispatchMsg(hab, &qmsg);
}
```

5.3.2 Examining the Message Queue

You can examine the contents of the message queue by using the **WinPeekMsg** or **WinQueryQueueStatus** function. It is useful to examine the queue if you start a lengthy operation that additional user input can affect, or if you need to look ahead in the queue to anticipate a response to user input.

You can use the **WinPeekMsg** function to check for specific messages in the message queue. The function is useful for extracting messages for a specific window from the queue. The function returns immediately if there is no message in the queue. This function can be used in a loop without requiring the loop to wait for a message to arrive. The following code segment checks the queue for WM_CHAR messages:

```
if (WinPeekMsg(hab, &qmsg, NULL, WM_CHAR, WM_CHAR, PM_NOREMOVE))
```

You can also use the **WinQueryQueueStatus** function to check for messages in the queue. This function is very fast and returns information about the kinds of messages available in the queue and which messages have been recently posted. Most applications use this function in program loops that need to be as fast as possible.

If you have a very long operation to carry out, you should consider creating a separate thread for the operation. Despite the MS OS/2 multitasking features, any application thread having a message queue that does not periodically relinquish control by calling the **WinGetMsg** or **WinWaitMsg** function risks monopolizing the CPU and seriously degrading system performance.

5.3.3 Posting a Message to a Window

You can use the **WinPostMsg** function to post a message to a window. The message goes to the window's message queue. For example, the following code fragment posts the WM_QUIT message:

```
if (!WinPostMsg(hwnd, WM_QUIT, OL, OL))

    /* Message was not posted. */
```

The function returns FALSE if the queue was full and the message could not be posted.

5.3.4 Sending a Message to a Window

You can use the **WinSendMsg** function to send a message directly to a window. Applications typically use this function to send messages to child windows. For example, the following code fragment directs a button control to draw a check mark by sending the BM_SETCHECK message to the control:

```
WinSendMsg(hwndButton, BM_SETCHECK, MPFROMSHORT(1), OL);
```

WinSendMsg calls the window's window procedure and waits for that procedure to handle the message and return a result. A message can be sent to any window in the system; all that is required is a handle to the window. The message is not stored in the message queue. The thread making the call must have a message queue.

5.3.5 Broadcasting a Message

You can send messages to multiple windows by using the **WinBroadcastMsg** function. This function is useful after an application changes a system value, for broadcasting the WM_SYSVALUECHANGED message. The following code fragment shows how to broadcast this message to all frame windows in all applications:

```
WinBroadcastMsg(
    hwnd,                                /* window handle         */
    WM_SYSVALUECHANGED,                  /* message ID            */
    OL,                                  /* no message parameters */
    OL,
    BMSG_FRAMEONLY | BMSG_POSTQUEUE);    /* all frame windows     */
```

You can broadcast messages to all windows, to just frame windows, or to just the windows in your application.

5.3.6 Using Message Macros

The MS OS/2 include files define several macros that help create and interpret message parameters.

One set of macros helps you construct message parameters. Macros are useful for sending and posting messages. For example, the following code fragment uses the **MPFROMSHORT** macro to convert a 16-bit integer into the 32-bit message parameter:

```
WinSendMsg(hwndButton, BM_SETCHECK, MPFROMSHORT(1), OL);
```

A second set of macros helps you extract values from a message parameter. They are useful for handling messages in a window procedure. The following code fragment illustrates this:

```
case WM_FOCUSCHANGE:
    fsFocusChange = SHORT2FROMMP(mp2);
    if (SHORT1FROMMP(mp2))
         hwndLoseFocus = HWNDFROMMP(mp1);
```

A third set of macros helps you construct a message result. They are useful for returning message results in a window procedure. This is illustrated by the following code fragment:

```
return (MRFROM2SHORT(1, 2));
```

5.4 Summary

The following are the functions and the message you can use to create and use message queues and messages:

WinBroadcastMsg Sends or posts messages to one or more windows or posts messages to all message queues.

WinCallMsgFilter Calls a message-filter hook procedure, passing it a message and a message-filter control code.

WinCreateMsgQueue Creates a message queue for the current thread and returns a handle to the queue.

WinDefDlgProc Carries out default processing for messages sent to a dialog window.

WinDefWindowProc Carries out default processing for messages sent to a window.

WinDestroyMsgQueue Destroys a particular message queue. This function must be called before terminating a thread that has a message queue.

WinDispatchMsg Sends a message to a specified window.

WinGetMsg Retrieves the next message from a message queue, removing that message from the queue. This function does not return until a message is available.

WinInSendMsg Determines whether the current thread is processing a message sent from another thread.

WinMsgMuxSemWait Waits for one of a list of semaphores to clear. This function is similar to the **DosMuxSemWait** function, except that the thread can continue to process messages sent to it from other threads.

WinMsgSemWait Waits for a semaphore to clear. This function is similar to the **DosSemWait** function, except that a thread can continue to process messages sent to it from other threads.

WinPeekMsg Copies the next message from a message queue without removing it from the queue. This function returns immediately, whether or not a message is available.

WinPostMsg Posts a message to the message queue for a specified window. After placing the message in the queue, the function returns immediately.

WinPostQueueMsg Posts a message in a specified message queue in the system. The function returns immediately after placing the message in the queue.

WinQueryMsgPos Retrieves the mouse-pointer position (in screen coordinates) that was stored with a posted message.

WinQueryMsgTime Retrieves the system time (milliseconds since the system was booted) that the message was posted.

WinQueryQueueInfo Retrieves information about a message queue.

WinQueryQueueStatus Retrieves status information about a message queue.

WinSendDlgItemMsg Sends a message to a control window that belongs to the specified dialog window.

WinSendMsg Sends a message directly to the window procedure of a specified window. The function does not return until the message has been processed by the window procedure.

WinTranslateAccel Translates a WM_CHAR message to a WM_COMMAND message if there is an entry for the specific character in the accelerator table.

WinWaitMsg Waits until a particular type of message appears in a message queue.

WM_QUIT Marks the end of message processing for a message queue. This message causes the **WinGetMsg** function to return FALSE.

Chapter
6

Window Classes

6.1 Introduction 93

6.2 About Window Classes 93
6.2.1 Custom Window Classes 93
6.2.2 Class Styles 94
6.2.3 Window Procedures 95
6.2.4 Public Window Classes 95
6.2.5 Preregistered Window Classes............................. 96
6.2.6 Custom Public Classes...................................... 97
6.2.7 Class Data 97

6.3 Using Window Classes 98
6.3.1 Registering a Private Window Class 98
6.3.2 Registering an Imported Window Procedure 98

6.4 Summary 98

6.1 Introduction

This chapter describes how applications create and use window classes. You should also be familiar with the following topics:

- Windows
- Window procedures
- Messages and message queues
- The *os2.ini* file

6.2 About Window Classes

A window class determines the window styles and window procedure given to windows of that class when they are created. Every window created by an application is a member of a window class. Each window class has an associated window procedure that it shares with all windows of that same class. The window procedure handles messages for all windows of that class and therefore defines the behavior and appearance of the window.

When an application creates a window, it must specify a window class. The **WinCreateWindow** function requires that the class be given explicitly. Other window-creation functions use specific classes by default. In all cases, a window class must be registered before it can be used to create windows. An application can register its own (private) window classes or use pregistered, public window classes.

6.2.1 Custom Window Classes

A custom (or private) window class is any window class registered by an application. The application defines the window procedure, class style, and window data size for the class and then registers the class by using the **WinRegisterClass** function. The window class is available to the application, but only to that application. Classes created in this way are private and cannot be shared by other applications. When the application terminates, the system removes any data associated with the private window class and invalidates the class name.

An application can register its own window classes at any time. Typically, an application registers window classes as part of its initialization, but this is not required. The only restriction is that no window of a particular class can be created until that class is registered by the application.

When an application registers a window class, it must supply the following information:

- Class name
- Class styles
- Window procedure
- Window data size

The class name identifies the window class. The application uses the class name when creating a window, specifying the class to use. The class name can be a character string or an integer value. The class name must be unique. The system checks to see if a public class or a class already registered by the application has the same name. If the class name is not unique, an error is returned.

The class style is one or more values that tell the system what initial window styles to give a window created with this class. Some class styles (for example, CS_SYNCPAINT) cause a new window to be given the corresponding window style when it is created. Styles such as CS_MOVENOTIFY direct the system to send messages to the window procedure when it ordinarily would not.

The window procedure is a function that receives and processes all messages sent to the window by the system. It is the chief component of the window class because it explicitly defines the appearance and behavior of each window created with the class. The window procedure can be part of the application or part of a dynamic-link library. In either case, it must be an exported function. When a window procedure is in a dynamic-link library rather than in the application, the application must import the window procedure by using an import library when linking, using the **IMPORTS** statement in the application's module-definition file, or using the **DosLoadModule** and **DosGetProcAddr** functions to retrieve the function address.

The window data size is a value that specifies the number of extra bytes to allocate for each window data structure. The system creates a window data structure for each window. The extra bytes can be used by an application to store additional information about a particular window.

Once created, a window-class data structure cannot be changed. However, it is relatively easy to change the window styles and window procedure of a window created with that class. An application cannot deregister a window class. Window classes remain registered and available until the application terminates.

An application that registers a window class can also support its own set of window styles for windows of that class. Standard window styles—for example, WS_VISIBLE and WS_SYNCPAINT—still apply to these windows. However, since a window style is a 32-bit integer and only the high 16 bits are used for the standard window styles, an application can use the low 16 bits for styles unique to the custom window.

MS OS/2 has unique window styles for all preregistered window classes. Styles such as FS_BORDER and BS_PUSHBUTTON are processed by the window procedure for the corresponding class and not by the system. This means that an application can build the support for its own window styles into the window procedure for its custom class. Using a window style designed for one window class will not work with another window class.

If more than one instance of an application is running at the same time, the window classes in one instance are not available to any other instance. This means a second instance must register the classes for itself. If an instance of an application terminates, the window classes for any other instance of that application remain unchanged.

6.2.2 Class Styles

Each window class has one or more class styles. A class style tells the system what initial window style to give a window created with that class. An application sets the class styles for a window class when it registers the class. The styles cannot be changed.

When you register a window class, you can specify one or more class styles, combining them as necessary by using the bitwise OR operator.

An application can examine the class style for any window class by using the **WinQueryClassInfo** function. There are ten class styles, as listed below:

Style	Description
CS_CLIPCHILDREN	Sets the WS_CLIPCHILDREN style for windows created with this class.
CS_CLIPSIBLINGS	Sets the WS_CLIPSIBLINGS style for windows created with this class.
CS_PARENTCLIP	Sets the WS_PARENTCLIP style for windows created with this class.
CS_SAVEBITS	Sets the WS_SAVEBITS style for windows created with this class.
CS_SYNCPAINT	Sets the WS_SAVEBITS style for windows created with this class.
CS_FRAME	Identifies windows created with this class as frame windows.
CS_PUBLIC	Creates a public window class.
CS_HITTEST	Directs the system to send WM_HITTEST messages to windows of this class whenever the mouse pointer moves in the window.
CS_SIZEREDRAW	Directs the system to invalidate the entire window whenever the size of the window changes.
CS_MOVENOTIFY	Directs the system to send WM_MOVE messages to the window whenever the window is moved.

6.2.3 Window Procedures

The window procedure for a window class handles the messages sent to windows of that class. One window procedure is shared by all windows of a class, so applications must ensure that no conflicts arise when two windows of the same class attempt to access the same global data. In other words, the window procedure must protect global data and other shared resources.

6.2.4 Public Window Classes

Although MS OS/2 allows an application to register its own window classes, most applications rarely register more than one window class. This window class supports the client window in the application's main window. For all other windows, the application generally uses public window classes. Public window classes support frame windows, controls, menus, and dialog windows.

Public window classes are available to all applications. An application does not need to register a public window class to use it. The window procedure for a public window class always resides in a dynamic-link library and is accessible to all applications. An application does not need to import the window procedure to use a public window class.

6.2.5 Preregistered Window Classes

MS OS/2 provides several preregistered, public window classes. Applications can use these public window classes to create frame windows, dialog windows, menus, push buttons, entry fields, and other controls. The window procedures for these classes are predefined so the application does not register the class before using it.

An application uses a preregistered, public window class by specifying its class in a call to the **WinCreateWindow** function. The class names for the preregistered, public window classes are integer values represented by the following constant names:

Class name	Description
WC_FRAME	Creates a frame window. Has class styles CS_FRAME, CS_HITTEST, CS_SYNCPAINT, CS_PUBLIC, and CS_CLIPSIBLINGS.
WC_BUTTON	Creates a button control. Has class styles CS_PARENTCLIP, CS_SYNCPAINT, CS_SIZEREDRAW, and CS_PUBLIC.
WC_ENTRYFIELD	Creates a text-entry control field. Has class styles CS_PARENTCLIP, CS_SYNCPAINT, CS_SIZEREDRAW, and CS_PUBLIC.
WC_LISTBOX	Creates a list box. Has class styles CS_PARENTCLIP, CS_SYNCPAINT, and CS_PUBLIC.
WC_MENU	Creates a menu. Has class styles CS_SYNCPAINT, CS_SIZEREDRAW, and CS_PUBLIC.
WC_SCROLLBAR	Creates a scroll bar. Has class styles CS_HITTEST, CS_PARENTCLIP, CS_SYNCPAINT, CS_SIZEREDRAW, and CS_PUBLIC.
WC_STATIC	Creates a static control. Has class styles CS_PARENTCLIP, CS_SYNCPAINT, CS_SIZEREDRAW, CS_HITTEST, and CS_PUBLIC.
WC_TITLEBAR	Creates a title bar. Has class styles CS_HITTEST, CS_PARENTCLIP, CS_SYNCPAINT, CS_SIZEREDRAW, and CS_PUBLIC.

Each preregistered, public window class also supports several window styles that an application can use to customize a window. For example, a window created with the WC_BUTTON class can have any one of four different behaviors and appearances. The application specifies the style (and the behavior and appearance of the window) in the call to the **WinCreateWindow** function. For a list of the available styles and more detailed information on each of the preregistered, public window classes, see Chapter 10, "Control Windows."

An application must not use the preregistered, public window-class names when it registers its own window classes.

6.2.6 Custom Public Classes

An application can create its own public window class, but this must be done in a special manner at system initialization. Only the shell can register a public window class and only when the system starts. Registering a public window class requires a special **load** entry in the *os2.ini* file that instructs the shell to load a dynamic-link library whose initialization routine registers the window class. Public window classes must be registered by using the **WinRegisterClass** function and must have the class style CS_PUBLIC. A public window class registered in this way can have the same name as an existing public window class, but this replaces the original window class.

If a dynamic-link library replaces a public window class, it can also save the previous window-procedure address and use it to subclass the original window class. The dynamic-link library retrieves the original window-procedure address by using the **WinQueryClassInfo** function. The new window procedure then passes unprocessed messages to the original window procedure. All windows created using this revised public window class will automatically be subclassed.

When subclassing a public window class, the window data size cannot be smaller than the original window data size. All public window classes defined by MS OS/2 use four extra bytes for storing a pointer to custom window data. This size is not guaranteed for public window classes defined by dynamic-link modules not belonging to MS OS/2.

6.2.7 Class Data

An application can examine a registered window class by using the **WinQueryClassInfo** function. This is useful for checking the class styles of a public window class. An application can also retrieve the name of the class for a given window by using the **WinQueryClassName** function. Using the window class name, you can then call the **WinQueryClassInfo** function to retrieve the window class data.

The **WinQueryClassName** function retrieves the name of the window class. If the window is one of the preregistered, public window classes, the window class name returned is in the form *#nnnnn*, where *nnnnn* is up to five digits representing the value of the window class name constant.

The **WinQueryClassInfo** function retrieves information about a window class. It copies the class style, window-procedure address, and window data size to a **CLASSINFO** data structure. The **CLASSINFO** structure has the following form:

```
typedef struct _CLASSINFO {     /* clsi */
    ULONG  flClassStyle;
    PFNWP  pfnWindowProc;
    USHORT cbWindowData;
} CLASSINFO;
```

6.3 Using Window Classes

The following sections explain how to register and use window classes in your applications.

6.3.1 Registering a Private Window Class

You can register a private window class at any time by using the **WinRegisterClass** function. You must define the window procedure in your application, choose a unique name, and set the window class styles for the class. The following code fragment shows how to register the window class name "MyPrivateName".

```
WinRegisterClass(hab,       /* anchor-block handle      */
    "MyPrivateName",        /* class name               */
    MyWindowProc,           /* far pointer to procedure */
    CS_SIZEREDRAW,          /* class style              */
    0);                     /* window data              */
```

6.3.2 Registering an Imported Window Procedure

You do not have to limit window procedures to your application's code segments. You can also register a window procedure that is imported from a dynamic-link library. You can do this in several ways. The easiest way is to import the window-procedure name by using either the **IMPORTS** statement in the application's module-definition file or by linking with an import library that contains an import record for the function.

6.4 Summary

An application can use the following functions to register and use window classes:

WinQueryClassInfo Obtains information about a window class. The class is specified by name. The function fills a **CLASSINFO** data structure. This gives you the long word of the window-class style, a pointer to its window procedure, and the number of additional words stored as part of the class.

WinQueryClassName Obtains a window class name for a given window. If the window class is one of the preregistered, public classes, the class name returned is in the form *#nnnnn*, where *nnnnn* is up to five digits representing the value of the window class name constant.

WinRegisterClass Registers a window class.

Chapter

7

Window Procedures

7.1 Introduction 101

7.2 About Window Procedures 101
7.2.1 Structure of a Window Procedure 101
7.2.1.1 Calling Convention 101
7.2.1.2 Arguments 102
7.2.1.3 Return Value... 103
7.2.2 Default Window Procedure 103

7.3 Using a Window Procedure.. 103
7.3.1 Associating a Window Procedure and Classes 104
7.3.2 Processing a Default Window Procedure 104

7.4 Summary 105
7.4.1 Window-Procedure Syntax 105
7.4.2 Messages Processed by the
Default Window Procedure 105

7.1 Introduction

This chapter describes window procedures and the default window procedure. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window classes
- Window messages and message queues
- Dialog windows and dialog procedures

7.2 About Window Procedures

Every window in MS OS/2 is associated with a window procedure that controls all aspects of the window: its appearance, how it responds to state changes, and how it processes user input.

Each window class has an associated window procedure, and all windows of that class use the same window procedure. For example, the system defines a window procedure for the frame window class, and all frame windows use that window procedure.

Applications typically define at least one new window class and an associated window procedure. The application can then create many instances of windows with that class, all of which use the same window procedure. Note that this means that the same piece of code may be called from several sources simultaneously. Therefore, care must be used when modifying shared resources from a window procedure.

Dialog procedures have the same structure and function as window procedures. All material referring to window procedures in this chapter also applies to dialog procedures. The only real difference between a dialog procedure and a window procedure is the recommended default procedure and what messages are handled.

The rest of this chapter shows how to write a window procedure and associate it with a window class.

7.2.1 Structure of a Window Procedure

All window procedures share a common syntax and structure. Because all messages must use the same calling syntax, the arguments and return value for window procedures are interpreted differently depending on the message being handled. The following sections describe the syntax and structure of a window procedure.

7.2.1.1 Calling Convention

From a programmer's point of view, a window procedure is a function that takes four arguments and returns a long word. The function must use Pascal calling conventions; that is, the arguments are pushed on the stack from left to right in the function definition and the function must clear the arguments from the stack before returning. This is different from the normal C-language calling convention, so the Microsoft C Optimizing Compiler provides the **pascal** keyword for defining functions that use the Pascal calling convention.

You should use the EXPENTRY macro when defining window procedures to ensure that the functions are declared appropriately.

Finally, you must ensure that the data-segment register is properly set up on entry to the window procedure. When the system calls the window procedure, it passes the proper data segment in the **ax** register. If you are programming in Microsoft C, use the **_loadds** keyword in your window-procedure definitions. This causes the compiler to insert the proper prolog and epilog code in your window procedures so that the data segment is initialized and restored properly. If you are programming in assembly language, you should load the **ds** register from **ax** on entry and restore the **ds** register on exit. If you are using another development environment, consult the relevant documentation for the appropriate compiler switch.

Because a window procedure can be called recursively, it is probably best to minimize the number of local variables used in the window procedure. To avoid overusing local variables and possible stack overflow in deep recursion, you should call other functions outside your window procedure to process individual messages.

7.2.1.2 Arguments

A window procedure takes four arguments: The first is a window handle; the second is a **USHORT** message designator; and the last two arguments are declared with the **MPARAM** data type, which is defined in the include files as a far pointer to the **VOID** data type (a generic pointer). The message arguments often contain information in both the low and high words of the long word. There are several macros defined in the *pmwin.h* include file that make it easier to extract bytes or integers from the **MPARAM** value. These macros include **SHORT1FROMMP**, which extracts the low-order word from an **MPARAM** value.

The window-procedure arguments are described in the following list:

Argument	Description
hwnd	Window handle of the window receiving the message.
msg	Message identifier. The message will generally correspond to one of the predefined constants (for example, WM_CREATE) defined in the system include files. This argument can also be equal to an application-defined message identifier. Application-defined messages must be greater than WM_USER. If a window procedure does not process a message, it is strongly recommended that it pass the message to the **WinDefWindowProc** function. This allows the default processing for the message to occur.
mp1	Message parameter. Its interpretation depends on the particular message.
mp2	Message parameter. Its interpretation depends on the particular message.

7.2.1.3 Return Value

The return value of a window procedure is defined as an **MRESULT** type. This is defined in the include files as a far pointer to a **VOID** data type. The actual interpretation of the return value depends on the particular message. Consult the description of each message to determine the appropriate return value.

7.2.2 Default Window Procedure

All windows in MS OS/2 share certain fundamental behavior. This basic behavior is encapsulated in the **WinDefWindowProc** function, the default window procedure. The default window procedure is provided so you can get the minimal functionality for a window by calling **WinDefWindowProc**. Control windows defined by the system can also call **WinDefWindowProc** for default processing.

7.3 Using a Window Procedure

The following code fragments show a sample window procedure. It shows how to use the message argument in a switch statement with individual messages handled by each case statement. Notice that each case returns a value specifically for that message. Consult the description of each message to determine the appropriate return value.

The window procedure calls the **WinDefWindowProc** function for any messages that it does not handle itself. **WinDefWindowProc** performs default processing for essential messages sent to windows.

```
MRESULT CALLBACK MyWindowProc(hwnd, msg, mp1, mp2)
HWND   hwnd;
USHORT msg;
MPARAM mp1;
MPARAM mp2;
{
    /* local variables */

    switch (msg) {

        case WM_CREATE:
        /* Initialize private window data. */

            return 0L;

        case WM_PAINT:
        /* Paint the window. */

            return 0L;

        case WM_DESTROY:
        /* Clean up private window data. */

            return 0L;

        default:
            break;
    }
    return (WinDefWindowProc(hwnd, msg, mp1, mp2));
}
```

A dialog procedure is exactly like a window procedure except that it receives a WM_INITDLG message instead of the WM_CREATE message. A dialog procedure should pass all unprocessed messages to the **WinDefDlgProc** function instead of passing them to the **WinDefWindowProc** function.

It is possible to write a window procedure that passes all messages to **WinDefWindowProc**, but the window will have no personality of its own. At the very least, a window procedure should handle the WM_PAINT message to draw itself. Typically, it should handle mouse and keyboard messages as well. Consult the descriptions of individual messages to determine if your window procedure should handle them.

There are times when you may want to create a window that does not change the default window behavior. That window's sole purpose is to keep track of its child windows. Object windows are often used this way. An easy way to create a window that does not change the default window behavior is to specify the **WinDefWindowProc** function as your window procedure when registering the window class. The application then does not write a separate window procedure for windows of this class.

7.3.1 Associating a Window Procedure and Classes

A window procedure is associated with a window class by passing a far pointer to the window procedure to the **WinRegisterClass** function. Once registered this way, the window procedure will be associated with each new window created with that class.

The following code fragment shows how to associate a window procedure with a window class:

```
WinRegisterClass(hab,     /* anchor-block handle      */
    szClassName,          /* class name               */
    MyWindowProc,         /* far pointer to procedure */
    CS_SIZEREDRAW,        /* class style              */
    0);                   /* window data              */
```

Another useful option is to subclass a window of an existing class. This is most often used to add functionality or to alter the behavior of frame windows.

To subclass a window, call the **WinSubclassWindow** function. This function returns a pointer to the window procedure for the window. Subclassing allows you to process messages using your own window procedure before passing unprocessed messages to the original window procedure. In this way, you can use the original window procedure instead of **WinDefWindowProc** for default window processing.

7.3.2 Processing a Default Window Procedure

Typically, you call the **WinDefWindowProc** function for any messages that are not handled in your window procedure. For each message handled you should return an explicit value that depends on the particular message. For all other messages you should return the **WinDefWindowProc** function. The following code fragment shows how to structure a window procedure to call the default window procedure for any unused messages:

```
switch (usMessage) {
    case WM_PAINT:
        hps = WinBeginPaint(hwnd, NULL, &rect);
        WinFillRect(hps, &rect, CLR_WHITE);
        WinEndPaint(hps);
        return 0L;
    default:
        break;
}
return (WinDefWindowProc(hwnd, usMessage, mp1, mp2));
```

You can also call **WinDefWindowProc** as part of your own processing of a window message. In these cases, you may want to modify the parameters to the message before passing it to **WinDefWindowProc**, or you may want to continue with the default processing after performing your own operations.

7.4 Summary

This section gives the window-procedure syntax and lists the messages processed by the default window procedure.

7.4.1 Window-Procedure Syntax

The following shows the syntax for a window procedure:

MRESULT CALLBACK WindowProc(HWND *hwnd*, **USHORT msg,**
MPARAM *mp1*, **MPARAM** *mp2*)

7.4.2 Messages Processed by the Default Window Procedure

The following messages are handled by the **WinDefWindowProc** function. For each message, the default processing is described; typical reasons for overriding the default behavior are also given:

WM_BUTTON1DBLCLK The default window procedure passes this message to the owner window. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON1DOWN The default window procedure activates the window by calling the **WinSetActiveWindow** function. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON1UP The default window procedure passes this message to the owner window. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON2DBLCLK The default window procedure passes this message to the owner window. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON2DOWN The default window procedure activates the window by calling the **WinSetActiveWindow** function. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON2UP The default window procedure passes this message to the owner window. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON3DBLCLK The default window procedure passes this message to the owner window. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON3DOWN The default window procedure activates the window by calling the **WinSetActiveWindow** function. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_BUTTON3UP The default window procedure passes this message to the owner window. Processing this message allows you to add functionality to mouse clicks and to differentiate the three possible mouse buttons.

WM_CALCVALIDRECTS The default window procedure returns the long-word value. Processing this message allows you to specify the portion of the window that is preserved when the window is resized and to specify where the preserved area is aligned when the window is redrawn.

WM_CHAR The default window procedure passes this message to the window owner. You can process this message to evaluate incoming keyboard events. In the case of standard control windows, unused WM_CHAR messages are passed to the **WinDefWindowProc** function and passed up the parent- and child-window hierarchy until reaching a frame or dialog window where default dialog effects, such as pressing the TAB key to move from control to control, are implemented.

WM_CLOSE The default window procedure posts a WM_QUIT message to the queue, which causes the message loop to terminate. Processing this message allows you to prevent the Close menu item in the System menu from terminating the program. This is particularly useful with child frame windows in a multiple-document application.

WM_CONTROLHEAP The default window procedure returns the heap handle for the heap maintained by the system for the window message queue. You process this message if the window maintains a separate heap.

WM_CONTROLPOINTER The default window procedure returns the mouse pointer passed in the *mp2* parameter, thus allowing the default pointer shape. This message is sent to the owner of a control window to allow it to change the shape of the mouse pointer when the pointer is over the control window. You can return a different mouse-pointer handle to override the mouse pointer chosen by the control window. For example, a special control that handles its mouse-movement message by sending a WM_CONTROLPOINTER message to its owner with its special pointer. The owner window then determines what pointer to use. The default window procedure would use the control's special pointer.

WM_DDE_INITIATE The default window procedure frees the selector in the *mp2* parameter and returns FALSE. You should process this message if your application supports the dynamic-data-exchange (DDE) protocol.

WM_DDE_INITIATEACK The default window procedure frees the selector in the *mp2* parameter and returns FALSE. You should process this message if your application supports the dynamic-data-exchange (DDE) protocol.

WM_FOCUSCHANGE The default window procedure passes the message to the owner window (if one exists) or to the parent window if no owner exists. If no owner or parent window exists, the default window procedure does nothing. Generally, this message is passed up the parent- and child-window hierarchy until it reaches a frame window, where the appropriate WM_ACTIVATE, WM_SETSELECTION, and WM_SETFOCUS messages are generated. This message is the first indication of a focus change.

WM_HELP The default window procedure passes this message to the parent window (if one exists). You process this message to provide context-sensitive help.

WM_HITTEST The default window procedure returns HT_ERROR if the window is disabled; otherwise, it returns HT_NORMAL. Processing this message to return HT_NORMAL for a disabled window allows the disabled window to receive mouse messages.

WM_MENUSELECT The default window procedure returns TRUE, which means the menu selection should be processed normally. You can process this message if you want to perform context-sensitive actions—for example, processing explanatory messages, each time a menu item is selected.

WM_MOUSEMOVE The default window procedure sets the mouse pointer to the arrow pointer. You normally process this message for mouse tracking after a mouse button-down message. This message is also useful for setting the mouse-pointer shape, depending on where the mouse is in the window.

WM_PAINT The default window procedure calls the **WinBeginPaint** and **WinEndPaint** functions to empty the update region for the window. If you do not process this message, the window will not be drawn.

WM_QUERYCONVERTPOS The default window procedure returns the cursor size and positional data pointed to by the *mp1* parameter and returns QCP_CONVERT to signify that the **RECTL** structure passed was properly converted.

WM_QUERYFOCUSCHAIN The default window procedure performs the default processing of this message, which defines the focus chain for that window.

WM_QUERYFRAMECTLCOUNT The default window procedure passes this message to the parent window until it finds a frame window, where it will be processed.

WM_QUERYWINDOWPARAMS The default window procedure sets all window parameters in the passed structure to zero and returns FALSE to indicate that the operation was not successful.

WM_SETWINDOWPARAMS The default window procedure does nothing except return FALSE.

WM_TIMER The default window procedure blinks the cursor for the window if the timer message has the TID_CURSOR timer ID. You process this message only if they create their own timers. Applications should pass a WM_TIMER (TID_CURSOR) message to the **WinDefWindowProc** function even if the application processes its own timers.

WM_TRANSLATEACCEL The default window procedure passes this message to the parent window (if one exists). This message is usually passed from a child window to its parent window until it reaches a frame window, where the default frame-window procedure calls the **WinTranslateAccel** function to determine if the key pressed is a valid accelerator key.

Chapter

8

Mouse and Keyboard Input

8.1 Introduction 111

8.2 About Mouse and Keyboard Input 111
8.2.1 The System Message Queue 111
8.2.2 Mouse Capture 112
8.2.3 Keyboard Focus 112
8.2.4 Window Activation 112

8.3 Using the Mouse and Keyboard in an Application.......... 112
8.3.1 Responding to Activation Events 113
8.3.2 Responding to Mouse Messages 113
8.3.2.1 Responding to Button Clicks 114
8.3.2.2 Responding to Mouse Movement 114
8.3.3 Changing the Mouse Capture 115
8.3.4 Keyboard Messages 115
8.3.5 Responding to Keyboard Messages 116
8.3.5.1 Key-Down or Key-Up Events 116
8.3.5.2 Repeat-Count Events 117
8.3.5.3 Character Codes 117
8.3.5.4 Virtual-Key Codes 117
8.3.5.5 Scan Codes 118
8.3.5.6 Accelerator-Table Entries 119
8.3.6 Changing the Keyboard Focus 119

8.4 Summary 120
8.4.1 Functions 120
8.4.2 Messages 121
8.4.2.1 Focus Change and Activation Messages 121
8.4.2.2 Mouse Messages 122
8.4.2.3 Keyboard Message 123

8.1 Introduction

This chapter describes how to use mouse and keyboard input in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window messages and message queues
- Accelerator tables

8.2 About Mouse and Keyboard Input

MS OS/2 Presentation Manager applications should support input from both the mouse and keyboard. The mouse can have either one, two, or three buttons. Only one window at a time can receive mouse input and only one window at a time can receive keyboard input. These windows are not necessarily the same, although they can be.

8.2.1 The System Message Queue

All mouse and keyboard input is routed through a single system message queue. The system takes input events, such as keys being pressed or mouse movements, out of the system queue and routes them to the appropriate application. Generally, mouse input is directed to the application that owns the window in which the mouse-pointer event occurs. An application can route all mouse input, regardless of mouse position, to a particular window by setting the mouse capture window. Keyboard input is sent to the application that owns the "keyboard focus" window. Keyboard focus and mouse capture are discussed more fully later in this chapter.

The system takes messages out of the system message queue and places them in the appropriate application message queue. An application receives input messages from its own queue by calling the **WinGetMsg** or **WinPeekMsg** function.

Mouse and keyboard events in the system queue are strictly ordered so that an input event cannot be processed until all previous input events have been fully processed. This is because the destination window of an input event is not known until all previous input has been processed.

For example, if a user types a command in one window, uses the mouse to activate another window, and then types another command in the second window, the destination of the second keyboard command depends on how the mouse click is handled. An application might not activate its window in response to the mouse click. The second keyboard event goes to the second window only if that window becomes active as a result of the mouse click, which also depends on how the application processes the mouse click.

Because the input queue is strictly ordered and events cannot be processed until all previous input has been processed, it is very important that applications process input events quickly to avoid slowing user interactions with the system.

8.2.2 Mouse Capture

Generally, a mouse message goes to the window under the mouse pointer at the time the event is read from the system queue. Applications can change this by setting the mouse capture window, in which case all mouse input is sent to the mouse capture window until the mouse capture is released or set to another window. Mouse capture is useful if a window should receive all mouse input even when the mouse pointer moves outside the window. For example, it is common to track the mouse-pointer position after a mouse button-down event, following the mouse pointer until a mouse button-up event is received. If you do not set the mouse capture to a particular window and the user moves the mouse pointer outside the window and releases the mouse button, your application will not receive the mouse button-up message. If you set the mouse capture to a particular window while tracking the mouse pointer, your application will receive the mouse button-up message even if the mouse moves outside the window.

8.2.3 Keyboard Focus

Only one window at a time receives keyboard input. The window receiving the keyboard input is called the keyboard-focus window. Applications can allow the system to set the keyboard-focus window by default as windows are activated and deactivated, or an application can specifically set the keyboard focus to a specific window. If there is no keyboard-focus window specified, the system sends keyboard input to the currently active frame window.

8.2.4 Window Activation

Because there can be many windows belonging to many applications on the screen at the same time, MS OS/2 provides a way to arbitrate input among windows and applications. There is at most one active application at any time in the Presentation Manager screen group. The active application usually has only one active frame window, although it is possible to have more than one active frame window. For example, an application with a multiple-document interface can have several child frame windows.

The activation state of a window is important when an application responds to mouse clicks. It is also important because activation and keyboard focus are closely related window attributes.

The **WinQueryActiveWindow** function returns the currently active frame window. Note that a client window is never returned by this function. Activation of the client window is an attribute of frame windows.

8.3 Using the Mouse and Keyboard in an Application

An application that uses the mouse and keyboard for input must respond to activation, mouse, and keyboard events. The following sections describe how to handle these three related topics.

8.3.1 Responding to Activation Events

A client window receives a WM_ACTIVATE message when its parent frame window is being activated or deactivated. The activation or deactivation message is usually accompanied by messages to set or lose the keyboard focus. Therefore, applications should not use the WM_ACTIVATE message to change the keyboard focus.

The low word of the first message parameter is TRUE if the window is activated, and FALSE if the window is deactivated.

One use for the WM_ACTIVATE message is to toggle the state of an application's private variable that tells whether a window is active or not, as shown in the following code fragment:

```
case WM_ACTIVATE:
     fActivated = (BOOL) mp1;
     return(OL);
```

It is important to know the activation state of a window in order to correctly handle mouse-button clicks.

8.3.2 Responding to Mouse Messages

Mouse messages occur when a user presses or releases one of the mouse buttons (a click) and when the mouse is moved. All mouse messages contain the *x*- and *y*-coordinates of the mouse-pointer hot spot (relative to the window coordinates of the window receiving the message) at the time the event occurs.

The system sends a WM_HITTEST message to the window that is about to receive a mouse message. The window can determine if it should actually receive the mouse message or not. The default processing of this message in the **WinDefWindowProc** function is to return HT_NORMAL if the window is enabled and HT_ERROR if the window is disabled. If the return value is HT_ERROR, the system does not send the mouse message to the window. Most applications pass WM_HITTEST messages on to the **WinDefWindowProc** function by default so disabled windows do not receive mouse messages. Windows that specifically respond to WM_HITTEST messages can change this default behavior.

Because windows process WM_HITTEST and mouse messages, an application can ignore hit-test code in a mouse message unless the application returns special values for hit-test code. One possible use for hit-test code is to react differently to a mouse click in a disabled window.

The contents of the mouse-message arguments (*mp1* and *mp2*) are listed below:

- The *x*-position is in low word of *mp1.*
- The *y*-position is in high word of *mp1.*
- The hit-test code is in low word of *mp2.*

8.3.2.1 Responding to Button Clicks

Applications typically respond to mouse button-down events differently depending on whether the window is currently active. The first button-down event in an inactive window should activate a window. A subsequent button-down event in an active window produces an application-specific action.

Typically, an application processes mouse clicks in the client window of a standard frame window. Because the activated/deactivated status of a window is a frame-window characteristic, the system does not provide an easy way to determine if the client window is active. That is, the window handle returned by the **WinQueryActiveWindow** function is the active frame-window handle rather than the client window owned by the frame.

The following are two typical methods for determining if a client is an active frame window:

- Call the **WinQueryActiveWindow** function and compare the window handle it returns with the frame window that contains the client window, as shown in the following code fragment:

  ```
  fActivated = (WinQueryWindow(hwndClient, QW_PARENT, FALSE) ==
      WinQueryActiveWindow(HWND_DESKTOP, FALSE))
  ```

- Maintain a private variable for the client window that is set and cleared when processing WM_ACTIVATE messages. Each time the frame window is activated, the client window receives a WM_ACTIVATE message with the low word of the first parameter equal to TRUE. When the frame window is deactivated, the client window receives a WM_ACTIVATE message with a FALSE activation indicator. The following code fragment shows how to use activation messages to toggle a private-status variable:

  ```
  case WM_ACTIVATE:
       fActivated = (BOOL) mp1;
       return (0L);
  ```

Depending on the method used to determine if a client window is active, a mouse button-down message is passed to the **WinDefWindowProc** function if the window is not active at the time of the message. The default processing activates the window and its frame.

A common problem for an application processing WM_BUTTON1DOWN or similar messages is the failure to activate or set the window focus. If the window processes character messages, the window procedure should call the **WinSetFocus** function to make sure the window receives the input focus and is activated. If the window does not need the keyboard focus, an application should call the **WinSetActiveWindow** function.

8.3.2.2 Responding to Mouse Movement

The system sends mouse-move messages to the window under the mouse pointer or the current mouse-capture window, if any, whenever the mouse pointer moves. This is useful for tracking the mouse pointer and changing its shape based on its location in a window. For example, the mouse pointer changes shape when it passes over the size border of a standard frame window.

All standard control windows use mouse-move messages to set the mouse-pointer shape. If your application handles WM_MOUSEMOVE messages in some situations but not others, unused messages should be passed to the **WinDefWindowProc** function to change the shape of the mouse pointer.

8.3.3 Changing the Mouse Capture

Mouse messages are usually routed to the window under the mouse pointer. Applications can call the **WinSetCapture** function to process all mouse messages by a specified window. This is particularly useful when an application is tracking the mouse pointer after a button-down message.

For example, in a paint application that uses a button-down message to start a drawing operation, the application tracks the mouse using mouse-move messages until a button-up message is received. If a user drags the mouse pointer outside the window and releases the button, the button-up message will not go to the original window unless the application has called the **WinSetCapture** function for that window.

Some applications must receive a button-up message to match a button-down message. When processing a button-down message, these applications call the **WinSetCapture** function to set the capture to their own window, and then they call the **WinSetCapture** function with a NULL window handle to release the mouse capture when processing a matching button-up message.

8.3.4 Keyboard Messages

All keyboard messages come to a window as WM_CHAR messages. The system reads the keyboard and collects keyboard events in the system queue. It then routes these messages to the appropriate windows depending on the current keyboard-focus window at the time the message is sent. WM_CHAR messages are sent to the window that has the keyboard focus. If no window has the keyboard focus, then WM_CHAR messages are posted to the active frame-window queue. The following are two typical situations where applications receive WM_CHAR messages:

- An application has a client window or custom control window, each of which can have the keyboard focus. If a window procedure for the client or control window does not process characters, it should pass them to its owner window, which can be accomplished by passing them through to the **WinDefWindowProc** function. This is especially true for dialog-control items, as this is how the TAB and direction-key control processing is implemented in the user interface.
- An application window owns a control window that handles some, but not all WM_CHAR messages. This is common in dialog windows. If a control window that has the focus in a dialog window cannot process a WM_CHAR message, it can call the **WinDefWindowProc** function to send the message to its owner, which is usually a dialog-frame window. The application dialog procedure then receives the WM_CHAR message. This is also the case when an application client window owns a control window.

8.3.5 Responding to Keyboard Messages

A WM_CHAR message may represent a key-down or key-up transition. It may contain a character code, a virtual-key code, or a scan code. This message also contains information about the state of the SHIFT, CONTROL, and ALT keys.

Each time a user presses a key, at least two WM_CHAR messages are generated: one when the key is pressed down, and one when the key is released. If the key is held down long enough to trigger the keyboard repeat, multiple WM_CHAR key-down messages are generated.

If the keyboard repeats faster than the application can retrieve the events from the event queue, the system combines repeating character events into one WM_CHAR event representing multiple-key events for the same key. WM_CHAR messages contain a count byte indicating the number of keystrokes represented by the message. Generally, this byte is set to 1, but it should be checked every time a WM_CHAR message is processed to avoid missing keystrokes.

A control may ignore the repeat count; for example, it may ignore the count on direction keys. If the system is slow, it may be more aesthetic to have a cursor move slowly than to see it jump 40 characters.

Applications decode WM_CHAR messages by examing individual bits in the flag word contained in the low word of the first argument passed with every WM_CHAR message. These bits may be set in various combinations. For example, a WM_CHAR message can have the KC_KEYDOWN, KC_CHAR, KC_SCANCODE, and KC_SHIFT attribute bits all set at the same time.

The *mp1* and *mp2* parameters that are part of the message contain different information depending on the nature of the keyboard event, as follows:

- The flag word is in low word of *mp1*.
- The repeat-key count is in low byte of high word of *mp1*.
- The scan code is in high byte of high word of *mp1*.
- The character code (ASCII) is in low word of *mp2*.
- The virtual-key code is in high word of *mp2*.

An application window procedure should return TRUE if it processes a particular WM_CHAR message, or FALSE otherwise. Typically, applications respond to key-down events and ignore key-up events.

The following sections describe the different types of WM_CHAR messages. Generally, decoding these messages consists of layers of conditional statements to eliminate and discriminate the different combinations of attributes that can occur in a keyboard message.

8.3.5.1 Key-Down or Key-Up Events

Generally, the first attribute that an application checks in a WM_CHAR message is the key-down or key-up events. The distinction between a key-down and a key-up event is found by examining the KC_KEYUP bit of the low word of the first message parameter. If this flag bit is set, then the message is from a key-up event. If the bit is clear, then the message is from a key-down event. The following code fragment shows how to decode a message for this information:

```
case WM_CHAR:
    fs = SHORT1FROMMP(mp1);

    if ((fs & KC_KEYUP))

        /* this is a key-up event   */

    else

        /* this is a key-down event */

    return TRUE;
```

8.3.5.2 Repeat-Count Events

Applications should always check the key repeat-count part of a WM_CHAR message to see if the message represents more than one keystroke. The count is greater than one if the keyboard is sending characters to the system queue faster than the application can retrieve them. If the system queue fills up, the system combines consecutive keyboard events for each key in a single WM_CHAR message with the repeat count set to the number of combined events. The repeat count is in the low byte of the high word of the first message parameter.

8.3.5.3 Character Codes

The most typical use of WM_CHAR messages is to extract a character code from the message and display the character on the screen. When the KC_CHAR bit is set in the WM_CHAR message, the low word of the second message parameter contains a character code based on the current code page. Generally, this value is a glyph code (typically an ASCII code) for the character for the key that was pressed.

The following code fragment shows how to respond to a character message:

```
    fs = SHORT1FROMMP(mp1);

    if (fs & KC_CHAR) {

        /* CHAR is in SHORT1FROMMP(mp2) */

        /* handle the key character */

        return(TRUE);
}
```

Note that if the KC_CHAR bit is not set, the **SHORT1FROMMP(*mp2*)** parameter may still contain useful information. If either the ALT or CTRL key, or both, are down, the KC_CHAR bit will not be set when the user presses another key. For example, pressing the *a* key when the ALT key is down, the low word of *mp2* will contain a 0x0041, the KC_ALT bit will be set, and the KC_CHAR bit will be clear. If the translation does not generate any valid characters, the **char** field is set to zero.

8.3.5.4 Virtual-Key Codes

WM_CHAR messages often contain virtual-key codes that correspond to various function keys and direction keys on a typical keyboard. These keys do not correspond to any particular glyph code but are used to initiate operations. When the KC_VIRTUALKEY bit is set in flag word of a WM_CHAR message, the high word of the second message parameter contains a virtual-key code for the key.

Note that some keys, such as the ENTER key, have both a valid character code and a virtual-key code. WM_CHAR messages for these keys will contain character codes for newline characters (ASCII 11) and virtual-key codes (VK_ENTER).

The following code fragment shows how to decode a WM_CHAR message containing a valid virtual-key code:

```
fs = SHORT1FROMMP(mp1);

if (fs & KC_VIRTUALKEY) {

    /* virtual key is in SHORT2FROMMP(mp2) */

    switch (SHORT2FROMMP(mp2)) {
        case VK_TAB:

            /* handle the TAB key   */

            return (TRUE);

        case VK_LEFT:

            /* handle the LEFT key  */

            return (TRUE);

        case VK_UP:

            /* handle the UP key    */

            return (TRUE);

        case VK_RIGHT:

            /* handle the RIGHT key */

            return (TRUE);

        case VK_DOWN:

            /* handle the DOWN key  */

            return (TRUE);
        .
        .
        .
        default:
            return (FALSE);
    }
}
```

8.3.5.5 Scan Codes

A third possible value in a WM_CHAR message is the scan code for the key pressed. The scan code represents the value generated by the keyboard hardware when a key is pressed. An application can use the scan code to identify the physical key pressed, as opposed to the character code represented by the same key. The byte-length value for the scan code is in the high byte of the high word of the first message parameter.

All WM_CHAR messages that are generated by the keyboard have valid scan codes. WM_CHAR messages that are posted by other applications may or may not have valid scan codes. The following code fragment shows how to extract a scan code from a WM_CHAR message:

```
fs = SHORT1FROMMP(mp1);
if (fs & KC_SCANCODE) {
    /* scan code is in HIBYTE(HIWORD(mp1)) */
    return (TRUE);
}
```

8.3.5.6 Accelerator-Table Entries

The system checks all incoming keyboard messages to see if they match any existing accelerator-table entries, either in the system queue or in the application-message queue. The translation first checks the accelerator table associated with the active frame window, and if no match is found, it uses the accelerator table associated with the message queues. If the keyboard event corresponds to an accelerator-table entry, the WM_CHAR message changes to a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, depending on the attributes of the accelerator table. The original WM_CHAR message is not processed by the application.

Accelerator tables should be used to implement keyboard shortcuts in applications rather than translating command keystrokes. For example, if an application uses the F2 key to save a document, a keyboard accelerator entry for the F2 virtual key should be created so that it generates a WM_COMMAND message rather than a WM_CHAR message.

8.3.6 Changing the Keyboard Focus

Applications can change the keyboard focus window by calling the **WinSetFocus** function for the new focus window.

The **WinSetFocus** function causes the following events to occur:

- If a window currently has the focus, it receives a WM_SETFOCUS message indicating the loss of focus.
- If a window currently has the focus, it receives a WM_SETSELECTION message indicating that it should deselect the current selection.
- If changing focus causes a change in the active window and there is a currently active window, a WM_ACTIVATE message is sent to the active window indicating the loss of active status.
- A new active window, new focus window, and the active application are established.
- If the active window is changing, a WM_ACTIVATE message is sent to the new main window indicating the acquisition of active status.

- The new focus window is sent a WM_SETSELECTION message indicating that it should select the current selection.
- The new focus window is sent a WM_SETFOCUS message indicating the acquisition of focus.

Using the **WinQueryActiveWindow** or **WinQueryFocus** function while processing the **WinSetFocus** function causes the previous active and focus windows to be returned until new active and focus windows are established. In other words, even though WM_SETFOCUS and WM_ACTIVATE messages with the *fFocus* parameter equal to FALSE may have been sent to the previous windows, those windows are considered active and have the focus until the system establishes new active and focus windows.

If the **WinSetFocus** function is called during the WM_ACTIVATE message processing, a WM_SETFOCUS message with the *fFocus* parameter equal to FALSE is not sent because no window has the focus.

8.4 Summary

The following sections describe the functions and messages associated with activation and keyboard/mouse input.

8.4.1 Functions

The following are the functions associated with activation, keyboard, and mouse input:

WinEnablePhysInput Enables or disables mouse and keyboard input, depending on the *fEnable* argument. Because this call affects the system queue, it is important that any application that disables input should enable it again as soon as possible.

WinFocusChange A version of the **WinSetFocus** function that allows more control over messages generated for the old and new focus windows. For example, if an application sets the focus to a new window without deselecting text in the old focus window, this function should be used.

WinGetKeyState Used to determine whether a specified virtual key is up, down, or toggled. A key, such as the CAPSLOCK key, is toggled if it has been pressed an odd number of times. This function can also be used to obtain the state of the mouse buttons that use the VK_BUTTON1, VK_BUTTON2, and VK_BUTTON3 virtual key codes.

WinGetPhysKeyState Returns information about the asynchronous (interrupt level) state of a specified virtual key. This function returns the physical state of the key; it is not synchronized to the processing of input and is not affected by calls to the **WinSetKeyboardStateTable** function.

WinIsPhysInputEnabled Returns the status, on or off, of mouse and keyboard input.

WinQueryCapture Returns the window handle of the window currently holding the mouse capture. If the *fLock* argument is TRUE, the window is returned locked and remains locked until it is unlocked by calling the **WinLockWindow** function with *fLock* set to FALSE.

WinQueryFocus Returns the keyboard focus window, or NULL if no focus window exists. If the *fLock* argument is TRUE, the window is returned locked and remains locked until it is unlocked by calling the **WinLockWindow** function with *fLock* set to FALSE.

WinSetCapture Sends all mouse messages to a specified window. Specifying a NULL window handle releases the mouse capture so that mouse input is sent to the window beneath the mouse pointer.

WinSetFocus Sets the focus window to the specified window, or to no window if a NULL window is specified for the *hwndSetFocus* argument. This function can cause activation and deactivation messages to go to the current and new focus windows. The window losing focus receives WM_SETFOCUS(FALSE) and WM_SETSELECTION(FALSE) messages. The frame window losing the focus receives a WM_ACTIVATE(FALSE) message, and by default passes it to its FID_CLIENT window. The frame window receiving the focus receives a WM_ACTIVATE(TRUE) message, which it passes to its FID_CLIENT window by default. The window receiving the focus receives the WM_SETSELECTION(TRUE), and WM_SETFOCUS(TRUE) messages.

WinSetKeyboardStateTable This function receives or sets the keyboard-state table. To change the state of one virtual key, call the **WinSetKeyboardStateTable** function with the *fSet* argument set to FALSE to copy the current state table into a 256-byte table (pointed to by the *pKeyStateTable* argument). It is then possible to modify a virtual-key entry in the table and call the **WinSetKeyboardStateTable** function using the same table and *fSet* argument set to TRUE. This call does not change the physical state of the keyboard. It affects the result of subsequent calls to the **WinGetKeyState** function, but not the result of calls to the **WinGetPhysKeyState** function.

8.4.2 Messages

The following sections describe the messages associated with focus change, activation, the mouse, and the keyboard.

8.4.2.1 Focus Change and Activation Messages

The following messages are associated with focus change and activation:

WM_ACTIVATE Sent to a window when it is activated or deactivated. This function can be used for tracking the activation state of a client window.

WM_FOCUSCHANGE Sent to a window when the focus is changing. Most applications pass this message to the **WinDefWindowProc** function, which sends it to the parent frame window. The frame window uses this message to generate appropriate WM_ACTIVATE, WM_SETFOCUS, and WM_SETSELECTION messages for the old and new focus windows.

WM_QUERYFOCUSCHAIN Used to define the focus chain (that is, to keep from hard wiring the focus chain to the parent-window relationship).

WM_SETFOCUS Sent to a window when it is losing or receiving the keyboard focus. A typical response is to display a text-insertion cursor when receiving the focus and hide the cursor when losing the focus.

WM_SETSELECTION Sent to a window when it is receiving or losing the keyboard focus. A typical response is to highlight the currently selected text when receiving the focus and unhighlight the selection when losing the focus.

8.4.2.2 Mouse Messages

The following messages are associated with mouse events:

WM_BUTTON1DBLCLK Sent to the window under the mouse pointer or the current mouse capture window, if any, when the user clicks the first mouse button twice in a system-specified time limit. The amount of time between clicks necessary to make the action a double-click is a system parameter that a user can set using Control Panel. The application receives a WM_BUTTON1DOWN and WM_BUTTON1UP message for the first click of a double-click.

WM_BUTTON1DOWN Sent to the window under the mouse pointer or the current mouse capture window, if any, when the user presses the first mouse button.

WM_BUTTON1UP Sent to the window under the mouse pointer or the current mouse capture window, if any, when the user releases the first mouse button.

WM_BUTTON2DBLCLK Like WM_BUTTON1DBLCLK but for the second mouse button.

WM_BUTTON2DOWN Like WM_BUTTON1DOWN but for the second mouse button.

WM_BUTTON2UP Like WM_BUTTON1UP but for the second mouse button.

WM_BUTTON3DBLCLK Like WM_BUTTON1DBLCLK but for the third mouse button.

WM_BUTTON3DOWN Like WM_BUTTON1DOWN but for the third mouse button.

WM_BUTTON3UP Like WM_BUTTON1UP but for the third mouse button.

WM_HITTEST This message occurs when an application requests a message by calling the **WinPeekMsg** or **WinGetMsg** function. If the message represents a mouse event, it is sent to the the window under the mouse pointer or to the current capture window, if any, to determine whether the message is destined for the window. The default window procedure returns HT_ERROR if the window is disabled; otherwise, it returns HT_NORMAL. The handling of this message determines whether a disabled window can process mouse clicks.

WM_MOUSEMOVE Sent to the window under the mouse pointer or the window with the mouse capture, if any, when the mouse pointer moves. The system generates this message only as often as the application requests new messages. The distance the mouse pointer moves before this message is posted depends on how fast the application executes its message loops.

8.4.2.3 Keyboard Message

The following message is associated with keyboard events:

WM_CHAR Posted to the current focus window whenever there is a keyboard event. The message contains a flag word that indicates the composition of the message. The following list of flag values can be used to test the flag word to determine the nature of a message:

Flag	Meaning
KC_ALT	The ALT key was down when this message was generated.
KC_CHAR	The message contains a valid character code for a key. Typically, this code is an ASCII code.
KC_COMPOSITE	In combination with the KC_CHAR flag this flag bit means that the character code is a combination of the key that was pressed and the previous dead key. This is used to create characters with diacritical marks.
KC_CTRL	The CONTROL key was down when this message was generated.
KC_DEADKEY	In combination with a KC_CHAR flag this flag bit means that the character code represents a dead-key glyph (such as an accent). An application displays the dead-key glyph and does not advance the cursor. Typically, the next WM_CHAR message is a KC_COMPOSITE message containing the character associated with the dead-key character.
KC_INVALIDCHAR	The current character is not valid for the current translation tables.
KC_INVALIDCOMP	The current character is not valid in combination with the previous dead key.
KC_KEYUP	The message was generated when the user released the key. If this flag bit is clear, the message was generated when the user pressed the key. Use this bit to determine key-down and key-up events.
KC_LONEKEY	No other key was pressed while this key was down. Typically used to indicate that the user pressed the ALT key by itself.
KC_PREVDOWN	In combination with a KC_VIRTUALKEY flag this flag bit means that the virtual key was previously down. If this flag bit is clear, the virtual key was previously up.

Flag	Meaning
KC_SCANCODE	The message contains a valid raw scan code generated by the keyboard when the key is pressed. The scan code is used by the system to identify the character code in the current code page, therefore, most applications do not need the scan code unless they cannot identify the key that was pressed. WM_CHAR messages generated by actual user keyboard input generally have a valid scan code, but WM_CHAR messages posted to the queue by other applications might not contain a scan code.
KC_SHIFT	The SHIFT key was down when this message was generated.
KC_TOGGLE	The KC_TOGGLE bit toggles "on" and "off" every time the key is pressed. For example, it is set "on" every odd number of presses and "off" every even number of presses. This is important for keys like NUMLOCK, which have an on or off state.
KC_VIRTUALKEY	The message contains a valid virtual-key code for a key. Virtual keys typically correspond to function keys.

Chapter

9

Frame Windows

9.1 Introduction 127

9.2 About Frame Windows 127
9.2.1 Main Window 127
9.2.2 Frame Controls 128
9.2.3 Client Window 128
9.2.4 Sizing Border and Minimize and Maximize Buttons 129
9.2.5 Frame-Control Identifiers 129
9.2.6 Frame-Window Creation 129
9.2.7 Frame-Control Flags 130
9.2.8 Frame-Window Styles 132
9.2.9 Frame-Window Resources 133
9.2.10 Frame-Window Class Data 134
9.2.11 Frame-Window Data 135
9.2.12 Frame-Window Operation 135
9.2.13 Nonstandard Frame Windows 136
9.2.14 Default Frame-Window Behavior 137

9.3 Using Frame Windows 139
9.3.1 Creating a Main Window 139
9.3.2 Retrieving Frame Handles 141

9.4 Summary 141
9.4.1 Functions 141
9.4.2 Messages 141

9.1 Introduction

This chapter describes creating and using frame windows in Presentation Manager applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Windows
- Window relationships
- Control windows
- Messages and message queues
- Resources and using the MS OS/2 Resource Compiler (**rc**)

9.2 About Frame Windows

A frame window is the basic window used by most Presentation Manager applications. A frame window provides a base for the application's main window, dialog windows, and message boxes. Although applications rarely use frame windows alone, applications nearly always start with a frame window to create a composite window that consists of the frame window, several frame controls, and a client window. The frame window coordinates the actions of the other windows, allowing the composite window to act as if it were a single unit.

A frame window is a window of the preregistered, public-window class WC_FRAME. The frame-window class, like the preregistered control classes, defines the appearance and behavior of the frame window. The appearance and behavior of a frame window are designed to match the standard user-interface guidelines for MS OS/2 Presentation Manager applications, including applications that use the multiple-document interface. This means that applications that use frame windows have a quick and efficient way to create the "standard" windows recommended by the user-interface guidelines.

Although frame windows are an important part of dialog windows, dialog windows are not described in this chapter. For a complete description of dialog windows, see Chapter 19, "Dialog Windows."

9.2.1 Main Window

An application's main window is typically made up of a frame window with control windows such as a title bar, System menu, menu bar, and scroll bar. The main window also typically includes a client window.

The frame window is sometimes *under* other windows. Although it is not visible, it provides the standard services the user expects from the window—for example, moving, sizing, minimizing, and maximizing. The frame window receives input from the control windows (called frame controls). It sends messages to the frame controls and to the client window to tell what action is needed next.

9.2.2 Frame Controls

When an application creates a frame window, it can specify that one or more control windows be created as child windows of the frame window. A frame window can have a title-bar, System-menu, menu, and scroll-bar controls. Each is a unique window created from a preregistered control-window class.

These frame controls provide a particular aspect of the user interface for a "standard" application window. A title bar appears at the top of the window and displays the application and/or window title. A System menu appears at the left end of the title bar. It contains the commands used to move, size, and close the window. A menu appears below the title bar and contains the commands the user can choose to carry out work with the application. The scroll bars appear at the right edge and bottom of the frame window. These let the user scroll the contents of the client window.

Although all frame controls are optional, most, if not all, application main windows use the title-bar and System-menu controls. These provide the minimum functionality for a window that meets the user-interface guidelines.

Each frame control is a child window of the frame window. Each frame control is owned by the frame window. That is, the frame window is the owner as well as the parent window for each frame control. Because the main role of a frame window is to coordinate the activities of other windows, ownership of the frame controls is very important. Ownership gives the frame controls a way to send notification messages to the frame window. Notification messages tell the frame window what the user does with the frame control.

For example, a user can move a window by clicking the title bar and then dragging the window to a new position using a mouse. The title bar responds to the click by sending a message to the frame window notifying it of the user's request to move the window. The frame window can then track the mouse motion and move the frame window and all its child windows to the new position.

An application can add frame controls to a frame window by specifying the FCF_TITLEBAR, FCF_SYSMENU, FCF_MENU, FCF_VERTSCROLL, and FCF_HORZSCROLL styles. Frame controls are described in separate chapters. For a general discussion of controls, see Chapter 10, "Control Windows."

9.2.3 Client Window

Every main window has a client window. The client window is the part of the main window where the application displays output and receives mouse and keyboard input. What an application displays in the client window, how it displays it, and how it interprets input to the window is controlled entirely by the application.

An application creates the client window when it creates the frame window. The client window is specific to the application; it is nearly always created by using a private window class (a class registered by the application). Like frame controls, the client window is a child window and an owned window of the frame window. This means, for example, that the client window moves when the frame window moves, that the client window is clipped to the frame-window size, and that the client window is destroyed when the frame window is destroyed.

The relationship between the frame window and the client window allows the frame window to pass messages from other frame controls to the client window and vice versa. For example, a scroll-bar control notifies the frame window when the user requests scrolling; the frame window then sends a message to the client window. The client window requests that the frame window change the window title; the frame window sends a message to the title-bar control.

9.2.4 Sizing Border and Minimize and Maximize Buttons

Although the sizing border and minimize and maximize buttons are not frame controls, they act very much like controls for the frame window. However, they are different than frame controls because the frame window draws and maintains these items; frame controls draw and maintain themselves.

The sizing border, enclosing the frame window, lets the user change the size of the window by using a mouse. The minimize button, at the right end of the title bar, lets the user shrink the frame window to an icon. The maximize button, next to the minimize button, lets the user enlarge the window so that it fills the screen. An application can add these items to a frame window by using the FCF_SIZEBORDER, FCF_MAXBUTTON, and FCF_MINBUTTON (or the FCF_MINMAX) styles. (The FCF_MINMAX style adds both a minimize and a maximize button.)

9.2.5 Frame-Control Identifiers

A frame window uses a set of standard constants to identify the frame controls and the client window. The frame-control identifiers all begin with the prefix FID_ and can be used in functions such as **WinWindowFromID** to uniquely identify a given control or the client window. The frame controls also use these identifiers in notification messages they send to the frame window. The following are the frame-control identifiers:

- FID_CLIENT
- FID_HORZSCROLL
- FID_MENU
- FID_MINMAX
- FID_SYSMENU
- FID_TITLEBAR
- FID_VERTSCROLL

9.2.6 Frame-Window Creation

An application can create a frame window by using the **WinCreateWindow** function and specifying the WC_FRAME window class. This creates the frame window but does not add the frame controls and client window that accompany most frame windows in applications. To add these additional windows, the application can continue to call the **WinCreateWindow** function, specifying the original frame window as the parent and owner window for each frame control and for the client window. Or the application can call **WinCreateStdWindow**, which automatically carries out the individual calls to **WinCreateWindow**.

Frame windows are also used to create dialog windows. In this case, the frame window contains control windows but no client window. An application can create a dialog window by using the **WinLoadDlg** or **WinCreateDlg** function. These functions require an appropriate dialog template from the application's resources on disk or from memory. The dialog template specifies the styles and dimensions for the frame window and the control windows that make up the dialog window.

9.2.7 Frame-Control Flags

An application can specify both the frame-window style and the frame controls for a frame window by using the frame-control flags with the **WinCreateStdWindow** function. The following are the frame-control flags:

Flag	Description
FCF_TITLEBAR	Creates a title bar.
FCF_SYSMENU	Creates a System menu.
FCF_MENU	Creates a menu. This flag loads a menu from the application's resources on disk.
FCF_MINBUTTON	Creates a minimize button.
FCF_MAXBUTTON	Creates a maximize button.
FCF_MINMAX	Creates both a minimize and a maximize button.
FCF_VERTSCROLL	Creates a vertical scroll bar.
FCF_HORZSCROLL	Creates a horizontal scroll bar.
FCF_SIZEBORDER	Creates a sizing border. A sizing border lets the user adjust the size of the window.
FCF_BORDER	Creates a border. Use this flag for windows that must not change size.
FCF_DLGBORDER	Creates a dialog border. Use this flag for dialog windows.
FCF_SHELLPOSITION	Directs the frame window to request an initial size and position from Start Programs.
FCF_TASKLIST	Adds the window title to the switch list of Task Manager. If the process creating a frame window already has an entry in the switch list, the window title is appended to the previous entry.

Flag	Description
FCF_NOBYTEALIGN	Enables the frame window to be moved to any position on the screen. If this flag is not given, a frame window always adjusts its position so that the *x*-coordinate of its left edge is a multiple of 8. Using this flag affects how quickly the system can draw the frame window.
FCF_NOMOVEWITHOWNER	Enables the frame window to maintain its position even if its owner window moves. This applies only to frame windows that are not child windows of the owner. If this flag is not given, the frame window moves when the owner window moves.
FCF_ICON	Loads an icon from the application's resources on disk. The icon is used whenever the frame window is minimized.
FCF_ACCELTABLE	Loads an accelerator table from the application's resources on disk. The accelerator table is used for all keyboard input to the frame window.
FCF_SYSMODAL	Creates a system-modal frame window. Setting this flag is the same as using the **WinSetSysModalWindow** function.
FCF_SCREENALIGN	Aligns the initial position of the frame window relative to the screen origin instead of to the parent window.
FCF_MOUSEALIGN	Aligns the initial position of the frame window relative to the mouse position instead of to the parent window. An application can use this flag to position the default button in a dialog window under the mouse pointer.
FCF_STANDARD	Combines the FCF_TITLEBAR, FCF_SYSMENU, FCF_MENU, FCF_SIZEBORDER, FCF_MINMAX, FCF_ICON, FCF_ACCELTABLE, FCF_SHELLPOSITION, and FCF_TASKLIST styles.

When the **WinCreateStdWindow** function is called without any of these flags set, the standard window is created invisible, behind all its sibling windows, in Z order, with a width and height of zero, positioned at the lower-left of its parent window. When **WinCreateStdWindow** returns, you can call **WinSetWindowPos** to change the window's size, *x*- and *y*-positions, Z-order position, and visibility.

When **WinCreateStdWindow** is called with the FCF_SHELLPOSITION frame-control flag, the window is created in front of its sibling windows, in Z order, with a standard size and *x*- and *y*-positions obtained from the shell program.

9.2.8 Frame-Window Styles

The frame-window class, like other preregistered window classes, provides many class-specific window styles that applications can use to adapt the appearance and behavior of a frame window. The frame-window styles, specified as constants starting with the FS_ prefix, can be combined with the standard window styles when creating a frame window. The following are the frame-window styles:

Style	Description
FS_ACCELTABLE	Loads an accelerator table from the application's resources on disk. The frame window uses the accelerator table to translate keyboard input.
FS_BORDER	Creates a single-line border. Use this style when the window must not change size.
FS_SIZEBORDER	Creates a sizing border. Use this style to let the user adjust the size of the window.
FS_DLGBORDER	Creates a double-line dialog border. Use this style for dialog windows.
FS_ICON	Loads an icon from the application's resources on disk. The frame window draws the icon when the window is minimized.
FS_SCREENALIGN	Aligns the initial position of the frame window relative to the screen origin instead of to the parent window.
FS_MOUSEALIGN	Aligns the initial position of the frame window relative to the mouse position instead of to the parent window. An application can use this style to position the default button in a dialog window under the mouse pointer.

Style	Description
FS_NOBYTEALIGN	Enables the frame window to be moved to any position on the screen. If this style is not given, a frame window always adjusts its position so that the *x*-coordinate of its left edge is a multiple of 8. Using this style affects how quickly the system can draw the frame window.
FS_NOMOVEWITHOWNER	Enables the frame window to keep its position even if its owner window moves. This applies only to frame windows that are not child windows of the owner window. If this style is not given, the frame window moves when the owner window moves.
FS_SHELLPOSITION	Directs the frame window to request an initial size and position from Start Programs.
FS_SYSMODAL	Creates a system-modal window. Using this style is the same as calling the **WinSetSysModalWindow** function for the frame window.
FS_TASKLIST	Adds the window title to the switch list of Task Manager. If the process creating a frame window already has an entry in the switch list, the window title is appended to the previous entry.
FS_STANDARD	Combines the FS_ICON, FS_ACCELTABLE, FS_SHELLPOSITION, and FS_TASKLIST styles.

The FS_ window styles are rarely used. Although the constants are useful for creating a frame window without also creating frame controls, most applications use frame controls and therefore use the FCF_ constants to specify the frame-window styles. For each FS_ constant there is an equivalent FCF_ constant. For more information, see the following section.

9.2.9 Frame-Window Resources

If the FCF_MENU, FCF_ICON, FCF_ACCELTABLE, FCF_STANDARD, FS_ICON, FS_ACCELTABLE, or FS_STANDARD style is specified when creating the frame window, the application must provide the appropriate resources to support these styles. Depending on the style, a frame window may attempt to load one or more resources from the application's resources on disk.

You can use Resource Compiler to add icon and accelerator-table resources to the application's executable file. Each resource must have a resource identifier that matches the resource identifier specified in the **FRAMECDATA** structure passed to the **WinCreateWindow** function or in the *idResources* parameter of the **WinCreateStdWindow** function.

The following list gives the frame-control flags and styles that require resources and describes what the resource should be:

Style	Resource
FCF_ICON FS_ICON	Requires an icon resource. The frame window draws the icon whenever the window is minimized.
FCF_MENU	Requires a menu-template resource. A frame window uses the menu template to create a menu containing the commands and menus specified by the resource.
FCF_ACCELTABLE FS_ACCELTABLE	Requires an accelerator-table resource. The frame window uses the accelerator table to translate WM_CHAR messages to WM_COMMAND, WM_SYSCOMMAND, or WM_HELP messages.
FCF_STANDARD FS_STANDARD	Requires a menu template, an accelerator table, and an icon resource.

The application must specify the module containing the resources (typically the application's executable file) when it creates the frame window. The resources must have the same resource identifier and the application must supply this identifier when creating the window.

9.2.10 Frame-Window Class Data

An application can specify class-specific data for a frame window by passing a **FRAMECDATA** structure to the **WinCreateWindow** function. The class-specific data contains the frame-control flags, resource-module handle, and resource identifier to be used when creating the frame window.

Frame-control flags specify what controls to create for the frame window and what window styles to apply to the frame window. The frame-control flags are the same flags (FCF_) used in the **WinCreateStdWindow** function. The resource-module handle and the resource identifier specify where to find resources for the frame window.

Supplying class-specific data with **WinCreateWindow** is similar to using the **WinCreateStdWindow** function without creating a client window.

9.2.11 Frame-Window Data

Frame-window data specifies the state of the frame window at a given time. An application can retrieve the frame-window data by calling the **WinQueryWindowUShort** function. A frame window has the following state flags:

Flag	Description
FF_ACTIVE	The frame window is activated.
FF_DLGDISMISSED	A frame window that is a dialog window has been dismissed by a call to the **WinDismissDlg** function.
FF_FLASHHILITE	The frame window is flashing and its flash state is TRUE.
FF_FLASHWINDOW	The frame window flashes as the result of a call to the **WinFlashWindow** function or a WM_FLASHWINDOW message.
FF_NOACTIVATESWP	The system should do no Z ordering on this frame window.
FF_OWNERHIDDEN	The frame window's owner window is hidden or minimized so the frame window is also hidden.
FF_OWNERDISABLED	For a frame window that is a dialog window, this flag indicates whether the owner window was enabled or disabled when the dialog window was loaded.
FF_SELECTED	The frame window has selection turned on.
FI_FRAME	The window is a frame window.
FI_OWNERHIDE	The frame window should be hidden or shown as a result of its owner window being hidden, shown, minimized, or maximized.
FI_ACTIVATEOK	The window can be activated.
FI_NOMOVEWITHOWNER	The window should move when its owner window moves.

9.2.12 Frame-Window Operation

The frame window maintains the size, position, and visibility of itself, its frame controls, and its client window. It responds to user requests to move, size, minimize, maximize, and redraw the window. It also responds to requests to close (destroy) the window and to change the focus and activation.

When moving or sizing a frame window, all owned windows maintain their position relative to owner window's upper-left corner.

Whenever the frame window redraws itself (for example, after moving or sizing), it draws the frame controls first, then lets the application draw the client window. This order ensures that the rapidly drawn frame controls are drawn before the relatively slowly drawn client window.

The order in which the frame controls are drawn depends on the Z-order position of the controls. Because the frame controls are sibling windows, the Z-order position of one is relative to the others. The following list specifies the Z-order position of the frame controls (from top to bottom):

FID_SYSMENU
FID_TITLEBAR
FID_MENU
FID_VERTSCROLL
FID_HORZSCROLL
FID_CLIENT

Although an application can change the Z-order position of any window, changing the relative positions of frame controls is not recommended.

When a frame window receives a request to minimize the window, it locates an available icon space in the lower part of the screen, hides all frame controls and the client window, and draws its icon. If the frame window has no icon (that is, the window was created without the FCF_ICON style), the frame window hides all but the client window. The client window must then draw the minimized window. An application can determine the size of a minimized frame window by calling **WinQueryWindowUShort** and specifying the QWS_XMINIMIZE and QWS_YMINIMIZE indexes.

When a frame window is maximized, it grows to the size of its parent window, plus an additional amount on each of its four sides equal to the width of its sizing border. Because a window is always clipped to its parent window, a maximized standard frame window does not show its sizing border.

Frame controls owned by a frame window or windows owned by child windows of a frame window are automatically destroyed when the frame window processes the WM_DESTROY message.

9.2.13 Nonstandard Frame Windows

Although most applications use frame windows to create main window and dialog windows, they are not limited to frame windows. Applications can create nonstandard frame windows and still use the standard frame controls, such as the title bar and System menu, within the nonstandard windows. One reason for creating nonstandard frame windows is to expand the capability of the frame window to support special features such as the multiple-document interface.

There are two ways to create nonstandard frame windows: subclass a frame window or create a private frame-window class. An application that subclasses a frame window can intercept the messages sent to the window and process them in new ways. An application that creates private frame-window classes essentially rewrites the frame-window procedure. In either case, the application gains much more control over the placement of frame controls in the frame window by creating nonstandard frame windows.

The messages WM_FORMATFRAME, WM_UPDATEFRAME, and WM_CALCVALIDRECTS control the arrangement of frame controls for applications that subclass. By intercepting these messages, an application can rearrange the placement of frame controls in a frame window.

For applications that create private frame-window classes, the **WinCreateFrameControls**, **WinCalcFrameRect**, and **WinFormatFrame** functions provide much the same capability as frame windows to maintain the size and position of frame controls.

9.2.14 Default Frame-Window Behavior

This section describes all the messages specifically handled by the predefined frame-window class.

Message	Description
WM_ACTIVATE	Sent to a title bar or sizing border so its highlight state matches the frame window's activation state.
WM_BUTTON1DOWN	If the frame window is minimized, captures the mouse. If the window is not minimized, activates the window.
WM_BUTTON2DOWN	Activates the frame window.
WM_BUTTON3DOWN	Activates the frame window.
WM_BUTTON1UP	Processes messages from minimized window frames.
WM_BUTTON2UP	Not processed.
WM_BUTTON3UP	Not processed.
WM_BUTTON1DBLCLK	If the frame window is minimized, posts a WM_SYSCOMMAND message to itself. Otherwise, activates the frame window and any control clicked.
WM_BUTTON2DBLCLK	Not processed.
WM_BUTTON3DBLCLK	Not processed.
WM_HITTEST	If the frame control is minimized, returns HT_ERROR if the window is disabled; otherwise, returns TF_MOVE.

Message	Description
WM_CALCVALIDRECTS	If there is no client window or the client window has the style CS_SIZEREDRAW, returns CVR_REDRAW to invalidate the entire window.
WM_CLOSE	If there is a client window, passes this message to it; otherwise, this message returns **WinDefWindowProc**.
WM_CONTROLHEAP	Attempts to allocate a heap for the frame controls. Returns a handle if successful; otherwise, returns NULL.
WM_CREATE	Creates specified frame controls by calling **WinCreateFrameControls**. Also creates any accelerator tables, loads icons, and adds itself to the switch list in Task Manager. These actions depend on the frame window and frame-control styles specified for the window.
WM_DESTROY	If the focus is held by a child window of the frame window, sets the focus to the frame window's parent window. Destroys any windows owned by the frame window. Destroys any child windows. Frees any control heaps. Destroys any icon created with the FS_ICON style. Destroys any accelerator table created with the FS_ACCELTABLE style.
WM_ENABLE	Returns **WinDefWindowProc**.
WM_ERASEBACKGROUND	Sent by the frame window to itself during WM_PAINT processing. Returns TRUE, signaling that the window should erase the client-window area.
WM_FORMATFRAME	Calls **WinFormatFrame** and **WinSetMultWindowPos** to format and position the frame controls.
WM_MINMAXFRAME	If there is a client window, passes a message to it; otherwise, passes a message via the **WinDefWindowProc** function.

Message	Description
WM_MOUSEMOVE	Determines the correct mouse pointer to use and returns **WinDefWindowProc**.
WM_PAINT	If the frame window is minimized, sends WM_QUERYICON and WM_ERASEBACKGROUND messages to itself and draws the icon. Otherwise, paints all of its controls, sends a WM_ERASEBACKGROUND message to the client window, and paints the client window.
WM_QUERYTRACKINFO	Obtains the default tracking information.
WM_SHOW	Returns **WinDefWindowProc**.
WM_SIZE	Sends a WM_FORMATFRAME message to itself.
WM_SYSCOMMAND	If the mouse is captured, ignores the system command. Otherwise, uses one of the following commands: SC_RESTORE, SC_SIZE, SC_MOVE, SC_CLOSE, SC_TASKMANAGER, SC_NEXT, SC_NEXTFRAME, SC_SYSMENU, SC_APPMENU.
WM_UPDATEFRAME	Calls **WinFormatFrame** to format the frame controls.

9.3 Using Frame Windows

The following sections detail creating and using frame windows in your Presentation Manager applications.

9.3.1 Creating a Main Window

You can create a main window by using the **WinCreateStdWindow** function. The following code fragment creates a typical main window: a frame window that has a System menu, title bar, menu, vertical and horizontal scroll bars, minimize and maximize buttons, and a sizing border:

```
/* Create a main window. */

ULONG flFrameControlFlags =
    FCF_SYSMENU | FCF_TITLEBAR    | FCF_SIZEBORDER |
    FCF_MENU    | FCF_MINMAX      | FCF_HORZSCROLL |
    FCF_VERTSCROLL;

hwndFrame = WinCreateStdWindow(
    HWND_DESKTOP,          /* frame-window parent       */
    OL,                    /* no window styles          */
    &flFrameControlFlags,  /* frame-control flags       */
    "MyClass",             /* client-window class       */
    "Main Window",         /* window title              */
    OL,                    /* no client-window styles   */
    NULL,                  /* app module has resources  */
    1,                     /* resource ID               */
    &hwndClient);          /* client-window handle      */
```

You can also create a "standard" main window for an application by creating a frame window with the FCF_STANDARD style. You create the frame window using the **WinCreateStdWindow** function. The following code fragment creates the window:

```
/* Set the frame-control flags. */

ULONG flFrameControlFlags = FCF_STANDARD;

hwndFrame = WinCreateStdWindow (HWND_DESKTOP, ..., &hwndClient);
```

Another way to create a main window and its frame controls is by calling the **WinCreateWindow** function to create the frame window and the frame controls and then calling **WinCreateWindow** to create the client window. One advantage of this approach is that you can specify an initial size and position of the frame window when you create it. The following code fragment illustrates this:

```
FRAMECDATA fcdata;

fcdata.cb = sizeof(fcdata);
fcdata.flCreateFlags = FCF_STANDARD;
fcdata.hmodResources = NULL;
fcdata.idResources   = idFrame;

hwndFrame = WinCreateWindow(
    HWND_DESKTOP,     /* frame-window parent              */
    WC_FRAME,         /* frame-window class               */
    "Main Window",    /* window title                     */
    OL,               /* initially invisible              */
    O, O, O, O,       /* size and position = O            */
    NULL,             /* no owner                         */
    HWND_TOP,         /* top Z-order position             */
    idFrame,          /* frame-window ID                  */
    &fcdata,          /* pointer to class-specific data   */
    NULL);            /* no presentation parameters       */

hwndClient = WinCreateWindow(
    hwndFrame,        /* client-window parent             */
    "My Class",       /* client-window class              */
    NULL,             /* no title for client window       */
    OL,               /* initially invisible              */
    O, O, O, O,       /* size and position = O            */
    hwndFrame,        /* owner is frame window            */
    HWND_BOTTOM,      /* bottom Z-order position          */
    FID_CLIENT,       /* standard client-window ID        */
    NULL,             /* no class-specific data           */
    NULL);            /* no presentation parameters       */

/* ... continue initialization ... */

WinShowWindow(hwndFrame, TRUE);
```

9.3.2 Retrieving Frame Handles

You can easily retrieve a frame-control handle by using the **WinWindowFromID** function. The following code fragment retrieves the control handle of the title bar:

```
hwndTitleBar = WinWindowFromID(hwndFrame, FID_TITLEBAR);
```

Given a frame-control handle, you can retrieve its parent frame-window handle by using the **WinQueryWindow** function:

```
hwndFrame = WinQueryWindow(hwndTitleBar, QW_PARENT, FALSE);
```

By using identifiers to identify frame controls rather than window classes, you can create your own controls to replace the predefined controls.

9.4 Summary

The following sections list the messages and functions you can use to create and use frame windows.

9.4.1 Functions

The following functions are used to create and use frame windows:

WinCreateStdWindow Creates a standard frame window.

WinCreateWindow Creates a standard frame window.

WinCreateFrameControls Creates standard frame controls for a given window.

WinFormatFrame Calculates the size and position of frame controls within a frame window. This function is typically used by applications that require a non-standard frame-window layout.

WinCalcFrameRect Determines the size of a frame window or its client window.

WinGetMinPosition Obtains a frame window's minimized position.

WinGetMaxPosition Obtains a frame window's maximized position.

WinFlashWindow Starts or stops frame-window flashing.

9.4.2 Messages

The following messages are used to create and use frame windows:

WM_ERASEBACKGROUND Sent to the client window when the background needs to be redrawn.

WM_FLASHWINDOW Sent to a frame window as a result of a call to the **WinFlashWindow** function.

WM_FORMATFRAME Sent to a frame window to calculate the sizes and positions of its component windows.

WM_MINMAXFRAME Sent to a frame window when it is about to be minimized, maximized, or restored.

WM_NEXTMENU Sent to the owner window (the frame window) to obtain the next or previous menu window.

WM_QUERYACCELTABLE Sent to a frame window to obtain the accelerator-table handle.

WM_QUERYBORDERSIZE Sent to the frame window to determine the size of the window border.

WM_QUERYFRAMECTLCOUNT Sent to the frame window to determine the maximum number of frame controls that can exist for a frame window.

WM_QUERYFRAMEINFO Sent to determine the following things about the frame window: whether the window is a frame window; whether the window can be activated; whether the window should move as a result of its owner being moved; whether the window should be hidden or shown as a result of its owner window being hidden, shown, minimized, or maximized.

WM_QUERYICON Sent to the frame window to obtain the icon handle.

WM_QUERYTRACKINFO Sent to the window procedure of the owner (the frame window) of a title-bar control at the start of track-move processing.

WM_SETACCELTABLE Sent to the frame window to set the accelerator-table handle.

WM_SETICON Sent to the frame window to set the icon the frame-window uses when it is minimized.

WM_TRACKFRAME Sent to the frame window to start the tracking operation for a frame window.

WM_TRANSLATEACCEL Sent to the focus window (the frame window) to allow for accelerator-translation of the WM_CHAR message.

WM_UPDATEFRAME Sent after frame controls have been added or removed from the frame window to notify the frame window to update the appearance of the window.

Chapter

10

Control Windows

10.1 Introduction 145

10.2 About Control Windows 145

10.2.1 Control-Window Features 145

10.3 Using Control Windows in an Application 146

10.4 Creating a Custom Control Window 146

10.5 Summary 147

10.5.1 Predefined Control-Window Classes 147

10.5.2 Messages Sent to a Control Window 148

10.5.3 Messages from a Control Window to an Owner Window 148

10.1 Introduction

This chapter describes the functions that allow you to use control windows in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Window Frames and creating standard frame windows
- Window messages and message queues

10.2 About Control Windows

Control windows are predefined window classes that applications use for input and output. Control windows are typically used as part of a dialog window and are defined in the dialog template. Applications can also create control windows by calling the **WinCreateWindow** function with the appropriate window-class specification. The following control-window classes are predefined in MS OS/2:

Control	Description
Button	Buttons or boxes that the user selects by clicking or using the keyboard. Several button types are available, including push buttons, radio buttons, and check boxes.
Entry field	A single line of text that the user can edit.
Static	Text, icons, or bitmaps that do not respond to user input.
List box	A window containing a list of items, usually text strings, from which the user may scroll and make selections.
Menu	A list of items, either text or bitmaps. The items in a menu may be displayed horizontally across the top of a frame window, as in a menu bar, or vertically, in a menu. Menus typically provide the command interface for an application.
Scroll bar	A bar that allows a user to scroll the contents of a window. Scroll-bar controls contain directional arrows and an absolute-position indicator called the slider.
Title bar	A title or caption displayed across the top of a frame window. They can be used by a user to move the window, by dragging the title-bar control.

10.2.1 Control-Window Features

Control windows are always owned by other windows, usually dialog windows or application frame windows. The ownership relationship is important because a control window sends notification messages to its owner whenever an action occurs in the control window. A control-window position is also expressed in the coordinate space of its owner.

Control windows are like other predefined window classes in that they respond to standard window management messages and functions, such as the **WinSetWindowText** and **WinShowWindow** functions.

Control windows are usually painted synchronously. This means that a control window is redrawn as soon as any part of it becomes invalid.

All control windows have a window ID. This ID is set either in a dialog template or when the control is created by the **WinCreateWindow** function. The ID is used when the control window sends notification messages to its owner. Care must be taken to make sure that the control ID for a particular window is not duplicated. Note that the control-window ID should not be the same as the command ID associated with individual menu items.

All control-window classes have a set of specific messages that they send and receive. The summary at the end of this chapter lists the messages that all control windows have in common.

10.3 Using Control Windows in an Application

Control windows can be used in dialog windows and standard-frame or client windows.

To use a control window in a dialog window, an application defines the control. A dialog template typically includes several control windows as part of the dialog template in its resource file. Then, when the dialog resource is loaded and displayed, control windows are automatically displayed as part of the dialog window. The application can then send messages to the control window to change its state. The dialog-window procedure defined by an application receives notification messages from the control window. The nature of these messages depends on the specific type of control window.

To use a control window in a non-dialog window, an application must call the **WinCreateWindow** function using the appropriate window-class specification. An application usually specifies one of its client windows as the owner of the control window. Therefore, the client-window procedure receives notification messages from the control window. In some cases where a control is owned by the frame window (such as a menu control), the notification messages to the frame are passed on to the client window.

10.4 Creating a Custom Control Window

MS OS/2 provides several predefined control-window classes. You can create custom-control windows to fit specific purposes in an application by doing the following:

- Using the user-drawn buttons, list boxes, and menus
- Subclassing an existing control-window class
- Registering and implementing a window class from scratch

Buttons, list boxes, and menus have an optional style designation that marks them as "user-drawn." This means that the owner window of the control with this style receives a message whenever the control must be drawn. (If the owner window is a frame window, it sends owner-drawn messages to its client windows, which should be handled by the client-window procedure.) This allows you to alter the appearance of a control window. For buttons, the owner-drawn style affects the drawing of the entire control. For menus and list boxes, the owner draws the individual items within the control, and the system draws the external outline of the control.

Subclassing an existing control window is an easy way to make custom controls. When you subclass an existing control window, you only alter those behaviors you want to change, letting all other messages through to the original control-window procedure.

The techniques for defining a custom control window are the same as those used in creating a client-window class. However, if you are creating your own custom-control window class, be sure it can send and receive the messages listed in Section 10.5.

If you create a custom control-window class by subclassing a control class or by creating a window class from scratch, you can use its class name in the dialog template just like a predefined window-class constant. For example, if you define and register a window class called "MyControlClass" in an application, you can define a dialog window containing a control window using the following resource definition:

```
DLGTEMPLATE IDD_CUSTOM_TEST
BEGIN
    DIALOG "", IDD_CUSTOM_TEST, 1, 1, 126, 130, FS_DLGBORDER, 0
    BEGIN
        CONTROL "This is Text", IDD_TITLE,
                37, 107, 56, 12,
                WC_STATIC,
                SS_TEXT | DT_CENTER | DT_TOP | DT_WORDBREAK
                | WS_VISIBLE
        CONTROL "Custom Control", IDD_CUSTOM,
                33, 68, 64, 13,
                "MyControlClass",
                WS_VISIBLE
        CONTROL "Okay", DID_OK,
                57, 10, 24, 14,
                WC_BUTTON,
                BS_PUSHBUTTON | BS_DEFAULT | WS_TABSTOP | WS_VISIBLE
    END
END
```

10.5 Summary

The following sections describe the predefined control-window classes and the messages common to all control windows.

10.5.1 Predefined Control-Window Classes

These the predefined control-window classes in MS OS/2:

WC_BUTTON A button control, including push buttons, radio buttons, and check boxes.

WC_ENTRYFIELD An entry-field control that allows single-line text editing.

WC_STATIC A static control that displays text, icons, or bitmap data.

WC_LISTBOX A list box that displays a list of items that can be scrolled.

WC_MENU A menu, including a menu bar and menus.

WC_SCROLLBAR A scroll bar that allows a user to scroll the contents of a window.

WC_TITLEBAR The title of a window at the top of the frame that allows a user to reposition a window on screen.

10.5.2 Messages Sent to a Control Window

All types of control windows receive the following messages:

WM_ADJUSTWINDOWPOS Sent to the control window by the **WinSetWindowPos** function to allow the control to modify its position. The message contains a pointer to an **SWP** structure that specifies the new control size and position. The control can modify the data in the **SWP** structure before the control is actually displayed or moved. For example, the dimensions of an entry-field control define the limits of the area that can be edited (not the border). The entry-field control modifies the fields of the **SWP** structure, specifying the size and position, including the border. List-box controls also modify their size and position, including any borders, and they can adjust their height to display all list items.

WM_QUERYDLGCODE Sent to the control window by the system to determine the kinds of messages the control processes. The control window returns a dialog code that is a combination of bit flags describing the messages the control responds to.

10.5.3 Messages from a Control Window to an Owner Window

The following are messages sent from a control window to an owner window:

WM_COMMAND Posted to the owner-message queue by menus and buttons. The owner window receives this message when the user selects a push button or chooses a menu item. This message also includes information about its source.

WM_CONTROL Sent (with the **WinSendMsg** function) to the owner of the control window. This message includes the control-window ID and other information specific to the type of control and nature of the message.

WM_CONTROLHEAP Sent by the control to its owner window when it needs a handle to a heap to allocate memory. For example, entry-field controls allocate memory to hold text associated with a control window. Generally, an application ignores this message, passing it on to the default window procedure that returns a handle to a heap.

WM_CONTROLPOINTER Sent to an owner window when the mouse pointer moves over the control window. The owner sets the mouse pointer to a different shape, if desired. The control passes an **HPOINTER** to a mouse pointer as part of this message. The owner can alter the default pointer shape by passing a different **HPOINTER** back. Applications that use the default should pass the same **HPOINTER** back as the result of this message or just pass the message on to the **WinDefWindowProc** function.

WM_HELP Posted by controls with the appropriate style. This message is like the WM_COMMAND message. The owner window receives this message and responds with help information, depending on the context information included in the message.

WM_SYSCOMMAND Posted by controls with the appropriate style. This message is like the WM_COMMAND message. It is not passed from a frame window to a client window. Generally, the only control sending this message is the system menu in a frame window.

Chapter

11

Title-Bar Controls

11.1 Introduction 153

11.2 About Title Bars 153

11.3 Using Title-Bar Controls in Applications 153

11.3.1 Altering Dragging Action 154

11.4 Default Title-Bar Behavior 154

11.5 Summary 155

11.1 Introduction

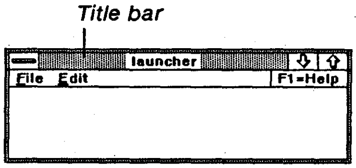
This chapter describes creating and using title-bar control windows. The title-bar control window is part of a standard frame window. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Standard frame windows
- Window messages and message queues

11.2 About Title Bars

A standard frame window is made up of several overlapping control windows that give the window its distinctive look and behavior. This chapter discusses one of these control windows: the title bar. Menus and scroll bars, the other control-window types that can be part of a frame window, are discussed in other chapters in this manual. Figure 11.1 shows a standard frame window with its title bar:

Figure 11.1
Frame Window with Title Bar



The title bar in a standard frame window performs four functions. First, it displays the title of the window across the top of the frame. Second, it changes its highlight appearance to show whether the frame window is active or not. Normally, the topmost window in a screen display is the active window. Third, the title bar responds to the user—for example, when the user drags the frame window to a new location on the screen. Finally, the title bar flashes (as a result of the **WinFlashWindow** function).

Title-bar control windows, like all control windows, must be owned by another window. Title-bar controls are owned by the frame window. A title-bar control sends messages to its owner when the control receives user input.

11.3 Using Title-Bar Controls in Applications

Typically, you need not be too concerned with the title-bar control. The default behavior of the title-bar control follows the standard user-interface guidelines. Most applications allow the title-bar control to operate according to these guidelines.

To include a title-bar control in a standard frame window, the application must compare (by using the OR operator) constants representing each control type and pass the resulting value to the **WinCreateStdWindow** function. The following code fragment shows the creation of a standard frame window with a title bar, a minimum/maximum control, a size border, a System menu, and an application menu. (The System menu and application menu are considered frame controls. For more information about frame controls, see Chapter 9, "Frame Windows.")

```
ULONG lControlStyle = FCF_TITLEBAR | FCF_SIZEBORDER |  FCF_MINMAX |
    FCF_SYSMENU | FCF_MENU;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE | FS_ACCELTABLE,
    &lControlStyle,
    szClassName,
    szClassName,
    OL, NULL,
    ID_MENU_RESOURCE,
    &hwndClient);
```

Once the frame controls are in place in the frame window, most applications can ignore them. The system handles the frame controls. In some cases, the application may take control of the title bar by sending messages to the title-bar control window.

To get the window handle of a title-bar control in a frame window, the application calls the **WinWindowFromID** function with the frame-window handle and a constant identifying the title-bar control, as shown in the following code:

```
hwndTitleBar = WinWindowFromID(hwndFrame, FID_TITLEBAR);
```

To change the text of a title bar, the application sets the window text of the frame window by calling the **WinSetWindowText** function. The frame window passes the message to the title bar. This changes the title-bar text.

11.3.1 Altering Dragging Action

When the user clicks in the title-bar control, the title bar sends its owner (the frame window) a WM_TRACKFRAME message. The frame window also sends a WM_QUERYTRACKINFO message to itself to fill in a **TRACKINFO** structure that defines the tracking parameters and boundaries. To modify the default behavior, you must subclass the frame window and intercept the message WM_QUERYTRACKINFO and modify the **TRACKINFO** structure. If you return TRUE for the WM_QUERYTRACKINFO message, the tracking information proceeds according to the information in the **TRACKINFO** structure. If you return FALSE, no tracking occurs.

11.4 Default Title-Bar Behavior

This section describes all the messages specifically handled by the predefined title-bar control class.

Message	Description
WM_CREATE	Sets the window text for the control. Returns FALSE if creation succeeds.

Message	Description
WM_DESTROY	Frees the window text for the control.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETWINDOWPARAMS	Sets the specified window parameters.
WM_PAINT	Draws the title bar.
WM_HITTEST	Always returns HT_NORMAL, so the title bar does not beep when it is disabled (it is disabled when the frame window is maximized).
WM_BUTTON1DOWN	Sends the WM_TRACKFRAME message to the owner (typically a frame window) to start tracking.
WM_BUTTON1DBLCLK	Restores the window if the owner window is minimized or maximized. If the window is neither, maximizes the window.
WM_ADJUSTWINDOWPOS	Returns FALSE. Process this message to prevent the **WinDefWindowProc** function from sending the size and show messages.
WM_QUERYDLGCODE	Returns the predefined constant DLGC_STATIC. A user cannot use the TAB key to move to this window in a dialog box.
TBM_QUERYHILITE	Returns the highlight state of the title bar.
TBM_SETHILITE	Sets the highlight state of the title bar, repainting it if the state is changing.

11.5 Summary

The following messages are associated with frame-control windows:

TBM_SETHILITE Sets the highlight state of the title bar to TRUE or FALSE. The system usually sends this message to a title bar to show whether or not the frame window containing the title bar is active.

TBM_QUERYHILITE Returns the highlight state (TRUE or FALSE) for a title-bar control window.

Chapter

12

Button Controls

12.1 Introduction 159

12.2 About Button Controls 159

12.3 Using Button Controls in an Application 160

12.3.1 Buttons in a Dialog Window 160

12.3.2 Buttons in a Client Window 162

12.3.3 Responding to a Button-Notification Message 162

12.3.4 Changing the Button State 163

12.3.5 Owner-Drawn Buttons 163

12.4 Default Button Behavior 164

12.5 Summary 165

12.5.1 Button Styles 166

12.5.2 Messages Sent to Button Controls 167

12.5.3 Messages Sent from Buttons to Owner Windows 167

12.1 Introduction

This chapter describes how to use button-control windows in your applications. You should also be familiar with the following topics:

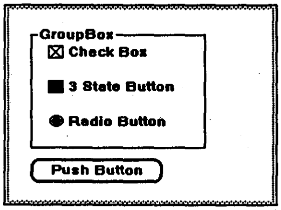
- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Window messages and message queues

12.2 About Button Controls

A button control is a window that represents a button that a user can select using the mouse or keyboard. Buttons can appear by themselves or in groups, and can appear with or without label text. A user can select a button by clicking it with the mouse or pressing the ENTER key when the button window has the keyboard focus. Buttons typically change appearance when selected.

There are four main types of buttons: push buttons, radio buttons, check boxes, and three-state check boxes. The appearance and behavior of button controls is determined by the style of the button. Figure 12.1 shows the different types of button controls.

Figure 12.1
Button Types



Radio buttons, check boxes, and three-state check boxes are used to control attributes of an operation. Push buttons are used to initiate operations. For example, a user might indicate paper size, print quality, and printer type in a print-command dialog window containing an array of radio buttons and check boxes. Once the user sets each option, a push button can be used to tell an application that printing should begin (or be canceled). The application queries the state of each radio button and check box to determine the printing parameters.

Push buttons are rounded-corner rectangular windows containing text strings. Push buttons become highlighted when they are selected by a user. They return to an unhighlighted state when the user releases the mouse button or the SPACEBAR. Push buttons are typically used to start or stop operations. A push button posts a WM_COMMAND message to its owner window.

Radio buttons are windows with text displayed to the right of a small, circular indicator. A radio button toggles between selected and unselected, each time the user selects it. The button retains the state until the next selection. Radio buttons usually appear in groups with only one button selected at a time. Radio buttons are appropriate where an exclusive choice is required from a group of related options. A radio button sends a WM_CONTROL message to its owner window.

Check boxes are similar to radio buttons except that they are used by themselves instead of in groups. They also toggle on or off application features. A check box sends a WM_CONTROL message to its owner window.

Three-state check boxes are similar to check boxes except that they can be displayed in halftone as well as selected or unselected. A three-state check box sends a WM_CONTROL message to its owner window.

In addition to the four predefined button-control types, an application can create buttons that appear defined by the owner window. Buttons using this style send BN_PAINT messages to their owner windows when they must be drawn or highlighted.

Button-control windows are always owned by other windows, typically by an application client window or a dialog window. A button control posts WM_COMMAND messages or sends WM_CONTROL notification messages to its owner when a user selects the button. Owner windows can also send messages to button controls to query or set states.

12.3 Using Button Controls in an Application

The most common way to use button controls is in a dialog window. An application defines one or more button controls in the dialog template in the resource file, and processes button messages in the dialog-window procedure.

Buttons can be associated in groups in dialog windows. A user can move from one button in a group to another button in the same group by pressing the direction keys. The TAB key moves from one group to the next. Groups are established by setting the WS_GROUP style bit for the first member of each group in the dialog template.

You can also use button controls in standard client windows. For these windows, create a button-control window by calling the **WinCreateWindow** function with a window class of WC_BUTTON. Specify the client window as the owner of the button window. The owner window receives messages from buttons and can send messages to the buttons to alter their control state. The control state includes highlighting control text, button position, and the enabled/disabled state.

Applications can create custom buttons that appear to be controlled by the application. The BS_USERBUTTON style, used in conjunction with other button styles, creates a button that notifies the application whenever the button must be drawn, allowing the application to draw the button.

12.3.1 Buttons in a Dialog Window

Buttons are typically used in dialog windows. An application can define buttons as part of a dialog-template resource file, as shown in the following sample Resource Compiler source-code fragment:

```
DLGTEMPLATE IDD_BUTTON
BEGIN
  DIALOG  "", 2, 64, 9, 235, 130
  BEGIN
      AUTORADIOBUTTON "R~adio1", ID_RADIO1, 15, 20, 40, 12, WS_GROUP
      AUTORADIOBUTTON "Ra~dio2", ID_RADIO2, 15, 40, 40, 12
      AUTORADIOBUTTON "Rad~io3", ID_RADIO3, 15, 60, 40, 12
      AUTORADIOBUTTON "R~adio4", ID_RADIO4,15, 80, 40, 12

      PUSHBUTTON "Button 1", ID_PUSH1, 100, 50, 14, WS_GROUP
      PUSHBUTTON "Button 2", ID_PUSH2, 75, 100, 50, 14, WS_GROUP
      PUSHBUTTON "Button 3", ID_PUSH3, 130, 100, 60, 14, WS_GROUP

      CHECKBOX "Check Box 1", ID_CHECK1, 150, 20, 58, 12, WS_GROUP
      CHECKBOX "no toggle", ID_CHECK2, 150, 40, 58, 12

      AUTOCHECKBOX "Check Box 3", ID_CHECK3, 150, 60, 58, 12, WS_GROUP
      DEFPUSHBUTTON "OK", DID_OK, 75, 26, 46, 20, WS_GROUP
  END
END
```

Each button item in a dialog window has an ID (for example, ID_RADIO1) that allows an application to identify the source of the WM_COMMAND and WM_CONTROL messages. The ID is also used to retrieve the button-window handle using the **WinWindowFromID** function.

The dialog template also specifies the text for each button, which is displayed in a rectangular box. If the button text is too long to fit in the button, it is clipped to the rectangle. For radio buttons and check boxes, text is displayed to the right of the button. A user selects the button by clicking either the button or the text itself.

The WS_GROUP attribute identifies the beginning of each new group of buttons. In the example above, the first four auto-radio buttons are in the same group, the following push buttons are in their own group, and the following two check boxes are in their own group. The auto-radio buttons in the first group can only be selected one at a time. An application must see that only one check box in a group is selected at a time. The group can wrap around from the end of the item list to the beginning.

Notice the DEFPUSHBUTTON style with the DID_OK identification number in the code fragment above. It is customary to include an OK button with this ID in most dialog windows to provide a uniform user interface. The DEFPUSHBUTTON style draws a thick border around a button and allows a user to select the button by pressing the ENTER key.

The dialog-window procedure for a dialog window containing buttons must respond to WM_COMMAND and WM_CONTROL messages. A common strategy is to use auto-radio buttons and auto-check boxes to allow a user to set a list of attributes for a command, and execute it by choosing the OK button. In this case, the dialog-window procedure ignores all WM_CONTROL messages that come from auto-radio buttons and auto-check boxes. These controls select and deselect themselves. When the dialog-window procedure receives a WM_COMMAND message for the OK button, it should query the auto-radio buttons and auto-check boxes to determine which options have been selected before proceeding with the operation.

12.3.2 Buttons in a Client Window

Applications can also use buttons in non-dialog windows. An application can create a button window using an application client window as the owner, as shown in the following code fragment:

```
/* Create a button window. */

hwndButton = WinCreateWindow(hwndClient,  /* parent       */
    WC_BUTTON,                            /* class        */
    "Test Button",                        /* text         */
    WS_VISIBLE | BS_PUSHBUTTON,           /* style        */
    10, 10,                               /* x, y         */
    60, 70,                               /* cx, cy       */
    hwndClient,                           /* owner        */
    HWND_TOP,                             /* behind       */
    ID_PBWINDOW,                          /* ID           */
    NULL,                                 /* control data */
    NULL);                                /* parameters   */
```

Once created in the client window, the button posts a WM_COMMAND message, or sends a WM_CONTROL message to the client-window procedure. The window procedure should examine the message ID to determine the button that sent the message.

Applications that have client-window buttons may move and size the buttons when the window receives a WM_SIZE message. This message indicates that the window is changing size. Buttons can be moved and sized using the **WinSetWindowPos** function. You can obtain a window handle for a button by calling the **WinWindowFromID** function using the parent window and the window ID for each button.

12.3.3 Responding to a Button-Notification Message

A button control sends a message to its owner window when a user selects the button, by either using the mouse or the keyboard. Buttons created with the BS_PUSHBUTTON or BS_USERBUTTON styles generate a WM_COMMAND message each time they are selected (this can be altered by specifying the BS_HELP or BS_SYSCOMMAND style when the button is created). All other button styles generate WM_CONTROL messages when selected.

A push button is automatically highlighted when a user selects it using the mouse. The button tracks the mouse until the user releases the mouse button. The highlight is turned off if the mouse moves outside the button boundaries. The push button does not generate any messages to its owner window until the user releases the mouse button, and then only if the mouse button is released inside the push-button boundary. When the owner window receives a WM_COMMAND message from a push button, the low word of the first parameter in the message contains the button window ID, as specified in the dialog template or when the button was created.

You should avoid duplicating IDs in menu commands and buttons because they both send IDs to owner windows as WM_COMMAND messages. However, it is possible to tell whether a WM_COMMAND message came from a menu or a push button by looking for the value CMDSRC_PUSHBUTTON or CMDSRC_MENU in the low word of the second parameter of the message.

Button types other than push buttons generate WM_CONTROL messages when selected. Applications can examine the low word of the first parameter in the message to find the button ID and the high word of the first parameter to determine the notification code for the control message. The notification code tells the application whether the control message originated from the user clicking or double clicking, or if the button needs to be drawn.

When a check box or radio button is selected, it sends a WM_CONTROL message with a notification BN_CLICKED code to the owner window. The owner window responds by sending a message back to the button to toggle its state.

In the case of auto-check boxes and auto-radio buttons, an application need not respond because these buttons toggle themselves in response to the mouse. The application still receives WM_CONTROL messages each time the button is selected. Most applications that use this default for radio buttons and check boxes should also use the automatic versions of these buttons and ignore any WM_COMMAND messages.

12.3.4 Changing the Button State

An application can query and set the highlight and checked state of its buttons by sending messages to button windows. The window handle for a button can be obtained by calling the **WinWindowFromID** function using the parent window and the window ID of the button. In the case of a dialog window, the parent would be the dialog window and the ID would be the button item ID from the dialog template.

Button-window text is stored as window text, and is accessible by using the **WinSetWindowText** and **WinQueryWindowText** functions. The size, position, and visibility of a button are set using standard window functions.

12.3.5 Owner-Drawn Buttons

An application can create custom buttons by using the BS_USERBUTTON style in combination with other styles. For example, an application can create a custom auto-radio button that works like an auto-radio button but whose appearance is controlled by the application. The owner window receives WM_CONTROL messages for these buttons whenever they must be drawn, highlighted, or disabled.

When a button must be drawn, the owner window receives a WM_CONTROL message with the low word of the first parameter equal to BN_PAINT. The second parameter is a pointer to a **USERBUTTON** structure that contains necessary information the application needs to draw the button. The **USERBUTTON** structure is shown below.

```
typedef struct _USERBUTTON {
    HWND hwnd;
    HPS hps;
    USHORT fsState;
    USHORT fsStateOld;
} USERBUTTON;
```

An application uses the **hwnd** field in this structure to find the bounding rectangle for the button. The **hps** field is used as a presentation space for any drawing. The high byte of the **fsState** field contains flags that tell an application how to draw the button: highlighted, unhighlighted, or disabled. The high byte of the **fsStateOld** field contains flags describing the current highlighted, unhighlighted, or disabled state of the button.

12.4 Default Button Behavior

This section describes the messages specifically handled by the predefined button-control window class.

Message	Description
WM_CREATE	Validates the requested button style and sets the window text.
WM_DESTROY	Frees the memory containing the window text.
WM_PAINT	Draws the button according to its style and current state.
WM_SETFOCUS	Creates a cursor if receiving the focus, destroys the cursor if losing the focus.
WM_BUTTON1DOWN	Sets mouse capture for the button window.
WM_MOUSEMOVE	Sets the default mouse pointer. If the button has the mouse capture, the button highlight state changes as the mouse pointer moves in and out of button area.
WM_BUTTON1UP	If the button has mouse capture, releases the mouse capture and sends notification message to the owner window if the mouse pointer is inside the button when the mouse button is released. If the button is a BS_PUSHBUTTON, a WM_COMMAND message is posted, otherwise a WM_CONTROL message with the BN_CLICKED code is sent.
WM_BUTTON1DBLCLK	Marks the button, sending a BN_DBLCLICKED notification code when the button-up message arrives.
WM_CHAR	Sets mouse capture when the SPACEBAR is pressed, releases capture when the SPACEBAR is released. Passes other key messages to the default window procedure.

Message	Description
WM_QUERYDLGCODE	Returns DLGC_BUTTON combined using the OR operator with the appropriate bits to designate the particular button type.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETWINDOWPARAMS	Sets the requested window parameters and redraws the button, including the cursor, if the window has the focus.
WM_ENABLE	Draws the button.
WM_MATCHMNEMONIC	Returns TRUE if *mp1* matches a button mnemonic.
BM_QUERYCHECKINDEX	Returns the zero-based index to the selected item in the same group as the button. Returns -1 if no button in the group is selected or if the button receiving the message is not a radio button or autoradio button.
BM_CLICK	Sends a WM_BUTTON1DOWN and WM_BUTTON1UP message to itself to simulate a user-button selection.
BM_QUERYCHECK	Returns the checked state of the button.
BM_SETCHECK	Sets the checked state of the button, returns the previous checked state.
BM_QUERYHILITE	Returns the highlight state of the button.
BM_SETHILITE	Sets the highlight state of the button, returns the previous highlight state.
BM_SETDEFAULT	Sets the default button state, redraws the button.

12.5 Summary

The following section lists the available styles for buttons and the messages associated with button controls.

12.5.1 Button Styles

The following are available button styles:

BS_3STATE Similar to a check box, except that it toggles between selected, unselected, and halftone states. The owner window receives a WM_CONTROL message and changes the state of the three-state check box when it is selected.

BS_AUTO3STATE Similar to a three-state check box, except it changes its state when selected.

BS_AUTOCHECKBOX Similar to a check box, except that it automatically changes its state when selected. An auto-check box sends a WM_CONTROL message to its owner window.

BS_AUTORADIOBUTTON Similar to a radio button, except that it automatically responds to a selection by changing its state and the state of all other radio buttons in its group.

BS_CHECKBOX A check box is a small square window that is empty when it is unselected, and contains an "x" when selected. Check-box text is displayed to the right of the check box. The owner window is notified with a WM_CONTROL message when the check box is selected and is responsible for changing the check-box state.

BS_DEFAULT A button with this style is outlined with a heavy black border. A user can select this button by pressing the ENTER key. This is useful for allowing a user to quickly select the most likely option, the default option, by pressing the ENTER key.

BS_HELP A button with this style posts a WM_HELP message when selected.

BS_NOBORDER A button with this style is drawn without a border.

BS_NOPOINTERFOCUS A button with this style does not receive the focus when selected.

BS_PUSHBUTTON A push button. The button posts a WM_COMMAND message to the owner window when selected.

BS_RADIOBUTTON Similar to a check box, except that it is used in groups of mutually exclusive choices. The owner window is notified with a WM_CONTROL message when a radio button is selected, and changes the state of the selected radio button and all other buttons in the group.

BS_SYSCOMMAND A button with this style posts a WM_SYSCOMMAND message when selected.

BS_USERBUTTON An application-defined button. This style, used in conjunction with other styles, allows the owner window to draw the button control in a highlighted, unhighlighted, enabled, or disabled state. The owner window receives a BN_PAINT message when the button must be drawn.

12.5.2 Messages Sent to Button Controls

The following messages are sent to button controls:

BM_CLICK The button responds as if it were selected using the mouse or the keyboard. This is useful when simulating a user selection for a particular button.

BM_QUERYCHECK Sent to a check box and radio button. Returns 0 if the button is unselected, 1 if the button state is selected, and 2 if the button state is indeterminate (for example, the grayed state for three-state buttons).

BM_QUERYCHECKINDEX The button responds to this message by setting the result to the zero-based index of the selected radio button in the same group as the sender. Returns - 1 if no button in the group is selected or the receiver is not a radio button, or if the radio button has no parent window.

BM_QUERYHILITE For push buttons, returns TRUE if the button is currently highlighted, FALSE otherwise. Returns FALSE if the button receiving the message is not a push button.

BM_SETCHECK Sets the state to unselected, selected, or indeterminate (for three-state buttons) for check boxes and radio buttons. If the button receiving the message also has the style BS_USERBUTTON, a WM_CONTROL message is sent to the button owner telling it to select or unselect the button.

BM_SETDEFAULT Sets or removes the BS_DEFAULT style bit in the receiving button.

BM_SETHILITE Sent to a push button. Sets the state of the button to highlighted or unhighlighted, depending on the parameters supplied with the message. If the button receiving the message has the BS_USERBUTTON style, a WM_CONTROL message is sent to the button owner telling it to highlight or unhighlight the button.

12.5.3 Messages Sent from Buttons to Owner Windows

The following messages are sent from buttons to their owner windows:

WM_COMMAND Posted from a push button to its owner window when a user selects the button. Additional parameters in the message indicate whether the selection was done using the mouse or the keyboard, although applications generally do not track how the button was selected.

WM_CONTROL Sent from a button control to its owner to indicate that a user has selected the button or when the owner must draw the button. Included in the message is one of the following notification codes:

Code	Description
BN_CLICKED	A user selected the button.
BN_DBLCLICKED	A user double-clicked the button.
BN_PAINT	Sent from a BS_USERBUTTON button to the owner window instructing it to draw the button control. The WM_CONTROL message contains a pointer to a **USERBUTTON** structure.

WM_HELP Posted instead of the WM_COMMAND message from a push button to its owner window when the button has the BS_HELP style.

WM_SYSCOMMAND Posted instead of the WM_COMMAND message from a push button to its owner window when the button has the style BS_SYSCOMMAND.

Chapter

13

Entry-Field Controls

13.1 Introduction 171

13.2 About Entry-Field Controls 171

13.3 Using Entry-Field Controls in an Application 171
13.3.1 Entry-Field Controls in a Dialog Window 172
13.3.2 Entry-Field Controls in a Client Window 172
13.3.3 Responding to a Message from an Entry-Field Control 173
13.3.4 Changing the State of an Entry-Field Control 173
13.3.5 Communicating with the System Clipboard 174

13.4 Default Entry-Field Behavior 174

13.5 Summary 176
13.5.1 Entry-Field Control Styles 176
13.5.2 Messages Sent to Entry-Field Controls 177
13.5.3 Messages Sent from an Entry-Field to an Owner Window 178

13.1 Introduction

This chapter describes the functions that allow you to use entry-field control windows in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- System clipboard
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Basic concepts of window messages and the message queue

13.2 About Entry-Field Controls

An entry-field control is a rectangular window that displays a single line of text a user can edit. When the entry-field control has the focus, it displays a flashing bar to mark the current insertion point. It also allows a user to select text by dragging the mouse or by using the keyboard. Entry-field controls allow applications to provide standard-interface text editing to users for short selections.

Users can select a range of text in an entry-field control. Many text-editing operations on the contents of an entry-field control affect the current selection rather than the entire text.

Entry-field controls are typically used in dialog windows, although they may be used in non-dialog windows as well. Entry-field control windows are always owned by other windows. The entry-field control window sends notification messages to its owner when it gains or loses the focus, or when its contents change or are scrolled.

Entry-field control windows have style bits that determine whether the text is left, center, or right-justified in the window, and whether the text automatically scrolls horizontally, showing the current insertion point. Style bits also control whether entry-fields have borders. These styles are set when the control is created.

13.3 Using Entry-Field Controls in an Application

Entry-field controls can be used in dialog windows and regular client windows. As part of a dialog window, the entry-field control is defined as part of the dialog template in the application resource file. In a client window, the application creates a window with the window class WC_ENTRYFIELD.

Once created, the contents, font, and selection range of the entry-field control can be changed by sending appropriate messages to the entry-field control window. Entry-field controls hold up to 32 characters by default, but applications can expand or reduce this limit, depending on memory limits, once the control is created. Applications can query the current contents or the current selection in an entry-field control.

Applications can also transfer text and data between the entry-field control and the system clipboard by using cut, copy, clear, and paste operations. This facility is useful for moving text from an entry-field control to another window or process.

13.3.1 Entry-Field Controls in a Dialog Window

Entry-field controls are typically used in dialog windows. The dialog window serves as the parent and owner window for the entry-field control. The application dialog procedure receives notification messages from entry-field controls. Generally, a dialog window includes a button that signals that the user wants to carry out an operation. The application ignores most notification messages from an entry-field control, allowing default text editing to occur. When the user selects the button that indicates an operation should begin, the application queries the contents of the entry-field control and proceeds with the operation.

To include an entry-field control in a dialog window, include a definition for an entry-field control item in the dialog-template definition in the resource file. The definition sets up the initial text, window ID, size, position, and style of the entry-field control. A sample dialog template containing an entry-field control is shown below.

```
DLGTEMPLATE IDD_SAMPLE
{
DIALOG "Sample Dialog", 50, 7, 7, 253, 145, FS_DLGBORDER,0
    {
        DEFPUSHBUTTON "~Ok", DID_OK, 8, 151, 50, 23, WS_GROUP
        ENTRYFIELD "Here is some text", ID_ED1, 42, 46, 68, 15,
            ES_MARGIN | ES_AUTOSCROLL
    }
}
```

Once created as part of a dialog window, the entry-field control sends notification messages to the dialog window. An application handles these messages in its dialog-window procedure.

An application communicates with an entry-field control in a dialog window by sending messages to the entry-field control window. The handle of the entry-field control window is obtained by calling the **WinWindowFromID** function (using the dialog window as the parent window and the window ID for the entry-field control as defined in the dialog template).

13.3.2 Entry-Field Controls in a Client Window

To have an entry-field control in a non-dialog window, an application calls the **WinCreateWindow** function with the window class WC_ENTRYFIELD. An application client window owns the entry-field control. The client-window procedure receives notification messages from the entry-field control. The following code fragment shows how to create an entry-field control window in a client window:

```
hwndEntryField = WinCreateWindow(hwndClient,      /* parent    */
   WC_ENTRYFIELD,                                 /* class     */
   "initial text contents",                       /* text      */
   WS_VISIBLE | ES_AUTOSCROLL | ES_MARGIN,        /* style     */
   xPos, yPos,                                    /* x, y      */
   xSize, ySize,                                  /* cx, cy    */
   hwndClient,                                    /* owner     */
   HWND_TOP,                                      /* behind    */
   hwndEntryField,                                /* win ID    */
   (PVOID) NULL,                                  /* ctl data  */
   (PVOID) NULL);                                 /* reserved  */
```

The entry-field control created in the preceding example has a 32-character default limit. An application can change this limit by sending an EM_SETTEXTLIMIT message to the control window. The limit can be set to a non-default value at creation by supplying a pointer to an **ENTRYFDATA** structure as the *ctldata* parameter to the **WinCreateWindow** function.

The other fields of the **ENTRYFDATA** structure can be set to specify the selection length and the first visible character at the left edge of the control window. Entry-field control attributes can also be changed by sending messages to the control after it is created. It is not necessary to provide this information in the **ENTRYFDATA** structure for creation to occur.

13.3.3 Responding to a Message from an Entry-Field Control

An entry-field control communicates with its owner by sending WM_CONTROL messages. These messages contain notification codes that specify the exact nature of the message. Typically, an application does not respond to notification messages from an entry-field control for default text-editing. For more specialized uses, an application uses notification messages to perform input filtering.

For example, if an application has an entry-field control that is intended for entering a number, it can use the EN_CHANGED message to check the contents after each new character is entered. This tells a user that an inappropriate character has been entered.

On a deeper level, an application can use the EN_SETFOCUS and EN_KILLFOCUS messages to toggle the input filtering that occurs before the character messages are sent to the entry-field control. An application can use conditional code in its main-event loop to filter incoming WM_CHAR messages whenever the entry-field control has the focus. By intercepting WM_CHAR messages before they are dispatched using the **WinDispatchMsg** function, an application can prevent inappropriate characters from reaching the entry-field control. You might also want to apply special interpretation to certain keystrokes, such as the ENTER key, as long as the entry-field control has the focus.

13.3.4 Changing the State of an Entry-Field Control

You can set or retrieve text in an entry-field control window by calling the **WinSetWindowText** or **WinQueryWindowText** function. To retrieve the text for the current selection, an application first calls the **WinQueryWindowText** function to retrieve the contents, and then sends an EM_QUERYSEL message to retrieve the offsets to the first and last character of the text selection. These offsets are used to retrieve selected text from the entire text.

Edit fields containing numerical values can be set or queried by calling the **WinSetDlgItemShort** or **WinQueryDlgItemShort** function and passing the entry-field ID and the parent window. The **WinSetDlgItemShort** function converts a signed or unsigned integer into a text string and sets the field text with it. The **WinQueryDlgItemShort** function converts the entry-field text to a signed or unsigned integer and returns the value in a specified variable.

An application can set or query the selection range, although the entry-field control automatically handles selection changes in response to user input in keeping with the standard user-interface guidelines. However, it can be useful to have an application select the entire text prior to cutting or pasting to the system clipboard.

13.3.5 Communicating with the System Clipboard

Entry-field controls respond to messages designed to simplify transferring data to and from the system clipboard. These messages support the standard cut, copy, clear, and paste operations defined by the standard user-interface guidelines. All clipboard messages for entry-field controls use the CF_TEXT clipboard-data format. See the summary section at the end of this chapter for a description of each message.

13.4 Default Entry-Field Behavior

This section describes all the messages specifically handled by the predefined entry-field control-window class.

Message	Description
WM_CREATE	Validate the requested style and set the window text.
WM_DESTROY	Free the memory used for the window text.
WM_BUTTON1DOWN	Set the mouse capture and keyboard focus to the entry-field and prepare to track the mouse during WM_MOUSEMOVE messages.
WM_BUTTON1DBLCLK	Set the mouse capture and keyboard focus to the entry-field and prepare to track the mouse during WM_MOUSEMOVE messages.
WM_BUTTON1UP	Release the mouse capture.
WM_BUTTON2DOWN	Return TRUE to prevent this message from being processed further.
WM_BUTTON3DOWN	Return TRUE to prevent this message from being processed further.
WM_PAINT	Draw the entry-field control and text.
WM_CHAR	Handle key events according to the standard user-interface guidelines.

Message	Description
WM_SETSELECTION	Invert the current selection range.
WM_SETFOCUS	If gaining the focus, create a cursor and send to the owner window a WM_CONTROL message with the EN_SETFOCUS control code. If losing the focus, destroy the current cursor and send to the owner window a WM_CONTROL message with the control code EN_KILLFOCUS.
WM_QUERYDLGCODE	Return the predefined constant DLGC_ENTRYFIELD.
WM_QUERYWINDOWPARAMS	Return the requested window parameters.
WM_SETWINDOWPARAMS	Set the specified window parameters, redraw the control, and send to the owner window a WM_CONTROL message with the EN_CHANGED control code.
WM_MOUSEMOVE	If the mouse button is down, track the text selection. If the mouse button is up, set the mouse pointer to the default arrow shape.
WM_ENABLE	Invalidate the entire entry-field control window, causing a WM_PAINT message to be sent to the control.
WM_TIMER	Blink the insertion point if the control has the focus. Scroll the text, if necessary, while extending a selection to text not visible in the window.
WM_ADJUSTWINDOWPOS	Change the rectangle for the control size to adjust for the margin if the control has the ES_MARGIN style.
EM_QUERYFIRSTCHAR	Return the offset to the first character visible at the left edge of the control window.
EM_SETFIRSTCHAR	Scrolls the text so that the character at the specified offset is the first character visible at the left edge of the control window. Returns TRUE if successful, or FALSE if it is not.

Message	Description
EM_QUERYCHANGED	Returns TRUE if the text has changed since the last EM_QUERYCHANGED message.
EM_QUERYSEL	Returns a long word that contains the offsets for the first and last characters of the current selection in the control window.
EM_SETSEL	Sets the current selection to the specified character offsets.
EM_SETTEXTLIMIT	Allocates memory from the control heap for the specified maximum number of characters, returning TRUE if it is successful, FALSE if it is not. Failure causes a WM_CONTROL message with the EM_MEMERROR control code to be sent to the owner window.
EM_CUT	Copies the current text selection to the system clipboard in CF_TEXT format and deletes the selection from the control window.
EM_COPY	Copies the current text selection to the system clipboard in CF_TEXT format.
EM_CLEAR	Deletes the current text selection from the control window.
EM_PASTE	Copies the current contents of the system clipboard that have CF_TEXT format, replacing the current text selection in the control.

13.5 Summary

The following sections describe the styles and messages associated with entry-field controls.

13.5.1 Entry-Field Control Styles

The following style constants, specified when the entry-field control is created, determine its behavior and appearance:

ES_AUTOSCROLL Scrolls text horizontally to show the current insertion point.

ES_CENTER Displays text centered within the control window.

ES_LEFT Displays text on the left within the control window.

ES_MARGIN Paints a wide border around the control window. The border is one-half characters wide and one-fourth characters high. Without this style, no border is drawn around the control window. The entry-field window rectangle is inflated (outset) on all sides by this margin. After an ES_MARGIN style entry-field is created, the **WinQueryWindowRect** function returns a larger rectangle, with a different origin than the one specified at creation because it now includes this margin. Note this when moving or sizing an entry-field control or it will become larger after each move or size operation.

ES_RIGHT Displays text on the right within the control window.

13.5.2 Messages Sent to Entry-Field Controls

This section describes the messages that are specific to entry-field controls:

EM_CLEAR Deletes the current selection in the entry-field control.

EM_COPY Sends the current selection to the system clipboard in CF_TEXT format.

EM_CUT Sends the current selection to the system clipboard in CF_TEXT format and then deletes the selection from the entry-field control.

EM_PASTE Replaces the current selection (or inserts text if the current selection is an insertion point) with text from the system clipboard. No replacement occurs if the clipboard does not contain any CF_TEXT data.

EM_QUERYCHANGED Returns TRUE if the contents of the entry-field control have changed since last receiving a WM_QUERYWINDOWPARAMS or EM_GETCHANGED message; it returns FALSE if the contents have not changed.

EM_QUERYFIRSTCHAR Returns the zero-based byte offset of the first character visible at the left edge of the control window.

EM_QUERYSEL Returns the offsets for the first and last characters of the current selection as the low and high word of the function return value.

EM_SETFIRSTCHAR Displays a character, specified by its zero-based byte offset, as the first character visible at the left edge of the control window, scrolling the text if necessary.

EM_SETSEL Sets the selection range between the supplied first and last character positions. If the first-character position is zero and the last-character position is greater than the number of characters in the control window, the entire text is selected.

EM_SETTEXTLIMIT Sets the maximum number of characters that the entry-field control can hold. It returns TRUE if the operation is successful, or FALSE if there was not enough memory.

13.5.3 Messages Sent from an Entry-Field to an Owner Window

The following section describes messages that an entry-field control sends to its owner window:

WM_CONTROL Notifies the owner window of a significant change in the control. The low word of the first parameter contains the window ID of the entry-field control window. The high word of the first parameter contains one of the following notification codes:

Control code	Description
EN_CHANGE	Contents of the entry-field control have changed and the change is displayed on screen.
EN_KILLFOCUS	Entry-field control loses the keyboard focus.
EN_MEMERROR	Entry-field control cannot allocate enough memory to perform the requested operation, such as extending the text limit.
EN_SCROLL	Entry-field control is about to scroll horizontally. This happens when a user enters text beyond the edge of the entry-field control boundary, requiring the text to scroll to continue displaying the insertion point.
EN_SETFOCUS	Entry-field control receives the keyboard focus.

Chapter
14

List-Box Controls

14.1 Introduction 181

14.2 About List Boxes 181

14.3 Using a List Box in an Application 181
- 14.3.1 Creating a List-Box Window 182
- 14.3.2 List Boxes in Dialog Windows 183
- 14.3.3 Adding and Deleting an Item in a List Box 183
- 14.3.4 Responding to a User Selection in a List Box 184
- 14.3.5 Handling Multiple Selections 185
- 14.3.6 Owner-Drawn List Items 185

14.4 Default List-Box Behavior 187

14.5 Summary 189
- 14.5.1 List-Box Styles 189
- 14.5.2 Messages Sent from a List Box to an Owner Window 190
- 14.5.3 Messages Sent to a List Box 191

14.1 Introduction

This chapter describes creating and using list-box control windows in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Window messages and message queues

14.2 About List Boxes

A list box is a control window containing a list of items. Each item contains a text string and a handle. The text string is usually displayed in the list-box window. The handle is available to the application to reference other data associated with the list item.

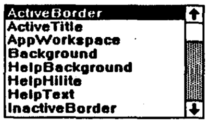
A list-box window must be owned by another window. This ownership relationship is important because the owner window receives messages from the list box when events occur—for example, when a user selects an item from the list box. Typically, the owner window is a client window of an application frame window or a dialog window. The client-window procedure or the dialog-window procedure defined by an application responds to messages sent from the list box.

A list box always contains a scroll bar. If the list box contains more items than can be displayed in the list-box window, the scroll bar is enabled. Otherwise, the scroll bar is disabled. The list box responds to clicks in the scroll bar by scrolling the list.

The maximum number of items in a list box is 32,767. This limit on list-box items is controlled by the 64K heap-size limit used in storing the list-box items.

Figure 14.1 shows a typical list-box control.

Figure 14.1
Typical List-Box Control



ActiveBorder
ActiveTitle
AppWorkspace
Background
HelpBackground
HelpHilite
HelpText
InactiveBorder

14.3 Using a List Box in an Application

An application uses a list-box control to display a list in a window. List boxes can be displayed in standard application windows, although they are more commonly used in dialog windows. Either way, notification messages are sent from the list box to the owner window, allowing the application to respond to user actions in the list. In practice, if a list box is owned by a dialog window, messages are handled in dialog-window procedures. If the list box is owned by a client window, notification messages are handled in the client-window procedure.

Once a list box is created, the application controls inserting and deleting list items. Items can be inserted at the end of the list, automatically sorted into the list, or inserted at a specified index position. Applications can turn list drawing on and off to speed up the process of inserting numerous items into a list.

The window procedure of the owner window of the list box receives messages when a user manipulates the list-box data. Most default list actions (for example, highlighting selections and scrolling) are handled automatically by the list box itself. The application controls the responses when the user chooses an item in the list, either by double-clicking the item or pressing ENTER after an item is highlighted. The application is also notified whenever the selection changes or when the list is scrolled.

Normally, list items are text strings drawn by a list box. An application can also draw and highlight the items in a list. This allows customized lists containing graphics or special fonts to be created. When an application creates a list box with the LS_OWNERDRAW style, the owner of the list box receives a WM_DRAWITEM message for each item that should be drawn or highlighted. This is similar to the owner-drawn style for menus, but unlike menus, the owner-drawn style applies to the entire list rather than to individual items.

14.3.1 Creating a List-Box Window

List boxes are WC_LISTBOX class windows and are predefined by the system. Applications can create list boxes by calling the **WinCreateWindow** function, using WC_LISTBOX as the window-class parameter.

A list box passes notification messages to its owner window, so an application uses its client window, rather than the frame window, as the owner of the list. The client-window procedure receives the messages sent from the list box.

For example, to create a list box that completely fills the client area of a frame window, an application would make the client window the owner and parent of the list-box window and make the list-box window the same size as the client window. This is shown in the following code fragment:

```
/* How big is the client window? */

WinQueryWindowRect(hwndClient, &rect);

/* Make a list-box window. */

hwndList = WinCreateWindow(hwndClient,       /* parent        */
    WC_LISTBOX,                              /* class         */
    "",                                      /* name          */
    WS_VISIBLE | LS_NOADJUSTPOS,             /* style         */
    0, 0,                                    /* x, y          */
    rect.xRight, rect.yTop,                  /* cx, cy        */
    hwndClient,                              /* owner         */
    HWND_TOP,                                /* behind        */
    ID_LISTWINDOW,                           /* ID            */
    NULL,                                    /* control data */
    NULL);                                   /* parameters    */
```

Because the list box draws its own border and a frame-window border already surrounds the client area of a frame window (because of the adjacent frame controls), the effect is a double-thick border around the list box. To change this, call the **WinInflateRect** function to overlap the list-box border with the surrounding frame-window border. This results in one list-box border.

Notice that the code specifies the list-box window style LS_NOADJUSTPOS. This ensures that the list box is created in exactly the size specified. If the LS_NOADJUSTPOS style is not specified, the list-box height is rounded down, if necessary, to make it a multiple of the item height. Allowing a list box to automatically adjust its height is useful for preventing partial items from being displayed at the bottom of a list box.

14.3.2 List Boxes in Dialog Windows

List boxes are most commonly used in dialog windows. A list box in a dialog box is a control window, like a push button or an entry field. Typically, the application defines a list box as one item in a dialog template in the resource-definition file, as shown in the following Resource Compiler source-code fragment:

```
DLGTEMPLATE IDD_OPEN
BEGIN
   DIALOG "Open...", IDD_OPEN, 35, 35, 150, 135,
         FS_DLGBORDER, FCF_TITLEBAR
      BEGIN
          LISTBOX        IDD_FILELIST, 15, 15, 90, 90
          PUSHBUTTON     "Drive", IDD_DRIVEBUTTON, 115, 70, 30, 14
          DEFPUSHBUTTON  "Open", IDD_OPENBUTTON, 115, 40, 30, 14
          PUSHBUTTON     "Cancel", IDD_CANCELBUTTON, 115, 15, 30, 14
      END
END
```

Once the dialog resource is defined, the application loads and displays the dialog box as it normally would. The application should insert items into the list when processing the WM_INITDLG message. The dialog-window procedure gets the window handle for the list box by calling the **WinWindowFromID** function using the list-box ID given in the dialog template. The following code fragment from a dialog-window procedure illustrates this:

```
case WM_INITDLG:
    hwndList = WinWindowFromID(hwndDialog, IDD_FILELIST);
        .
        . /* Now use hwndList to send LM_INSERTITEM messages. */
        .
    return OL;
```

It is very common for a dialog window with a list box to have an OK button. The user may select items in the list and then indicate a final selection by double-clicking, pressing ENTER, or clicking the OK button. When the dialog-window procedure receives a message that the user has clicked the OK button, it should query the list box to determine the current selection (or selections if the list allows multiple selections) and then respond as if it received a WM_CONTROL message with the LN_ENTER notification code.

14.3.3 Adding and Deleting an Item in a List Box

Applications can add or delete items in a list box by sending LM_INSERTITEM and LM_DELETEITEM messages to the list-box window. Items in a list are specified with a zero-based index (beginning at the top of the list). A new list is always created empty. The application must initialize the list by inserting it

The application specifies the text and position for each new item. It an absolute-position index or one of the following predefined in

Value	Meaning
LIT_END	Insert item at end of list.
LIT_SORTASCENDING	Insert item alphabetically ascending into list.
LIT_SORTDESCENDING	Insert item alphabetically descending into list.

The application must send an LM_DELETEITEM message and supply the absolute index position of the item when deleting items from a list. The LM_DELETEALL message deletes all items in a list.

One way an application can speed up the process of inserting list items is to suspend drawing in the list while inserting items. The list is redrawn after the insertion process is finished. This is a particularly valuable approach when using a sorted insertion process, when inserting one item can cause rearrangement of the entire list. List drawing is turned off by calling the **WinEnableWindowUpdate** function with FALSE for the enable parameter, and then calling the **WinShowWindow** function. This forces a complete update when insertion is complete. The following code fragment illustrates this concept:

```
/* Disable updates while filling the list. */

WinEnableWindowUpdate(hwndFileList, FALSE);
    .
    . /* Send LM_INSERTITEM messages to insert all new items. */
    .

/* Now cause the window to update and show the new information. */

WinShowWindow(hwndFileList, TRUE);
```

Note that this optimization is not necessary if adding list items when processing a WM_INITDLG message because the list box is not visible and the list-box routines are internally optimized.

14.3.4 Responding to a User Selection in a List Box

The primary notification an application receives when a user chooses an item in a list is a WM_CONTROL message with the LN_ENTER control code sent to the owner window of the list. The owner window is an application client window or a dialog window. Within the window procedure for the owner window, the application responds to the LN_ENTER control code by querying the list box for the current selection (or selections, in the case of an LS_MULTIPLESEL list box).

The LN_ENTER control code notifies the application that the user has selected a list item. A WM_CONTROL message with an LN_SELECT control code is sent to the list-box owner whenever a selection in a list changes, such as when a user moves the mouse pointer up and down a list while pressing the mouse button. In this case, items are selected but not yet chosen. An application may ignore LN_SELECT control codes when the selection changes, responding only when the item is actually chosen. An application might use the LN_SELECT control code to display context-dependent information that changes rapidly with each selection made by the user.

14.3.5 Handling Multiple Selections

When a list box has the style LS_MULTIPLESEL, more than one item may be selected at a time. An application must use different strategies when working with this type of list. For example, when responding to an LN_ENTER control code, it is not sufficient to send a single LM_QUERYSELECTION message because that message will find only the first selection. To find all current selections, an application should continue sending LM_QUERYSELECTION messages, using the return index of the previous message as the starting index of the next message, until no items are returned.

14.3.6 Owner-Drawn List Items

To draw its own list items, an application must create a list that has the style LS_OWNERDRAW. The owner window of the list box must respond to the WM_MEASUREITEM and WM_DRAWITEM messages.

When the owner window receives a WM_MEASUREITEM message, it must return the height of the list item. All items in a list must have the same height (greater than or equal to 1). The WM_MEASUREITEM message is sent only once, when the list box is created. You can change the item height by sending an LM_SETITEMHEIGHT message to the list-box window.

The owner window receives a WM_DRAWITEM message whenever an item in an owner-drawn list should be drawn or highlighted. The owner window returns FALSE if the list box must draw the text of the item. The owner window returns TRUE if it actually draws the item. This tells the list box not to draw the item. This is useful if the owner window alters the appearance of certain items, allowing other items to be drawn by the list box.

Although it is quite common for an owner-drawn list to draw items, it is less common to override the system-default method of highlighting. (The system-default highlighting method inverts the rectangle that contains the item.) Do not create your own highlighting unless the system-default method is unacceptable to you.

The WM_DRAWITEM message contains a pointer to an **OWNERITEM** data structure. The **OWNERITEM** data structure contains the window ID for the list box, a presentation-space handle, a bounding rectangle for the item, the position index for the item, and the application-defined item handle. This structure also contains two fields that determine if a message draws, highlights, or removes the highlighting from an item. The **OWNERITEM** data structure has the following form:

```
typedef struct _OWNERITEM {
    HWND    hwnd;
    HPS     hps;
    USHORT  fsState;
    USHORT  fsAttribute;
    USHORT  fsStateOld;
    USHORT  fsAttributeOld;
    RECTL   rclItem;
    SHORT   idItem;
    ULONG   hItem;
} OWNERITEM;
```

When the item must be drawn, the owner window receives a WM_DRAWITEM message with the **fsState** field set differently than the **fsStateOld** field. If the owner window draws the item in response to this message, it returns TRUE, telling the system not to draw the item. If the owner window returns FALSE, the system draws the item using the default list-item drawing method.

You can get the text of a list item by sending an LM_QUERYITEMTEXT message to the list-box window. You should draw the item using the **hps** and **rclItem** arguments provided in the **OWNERITEM** structure.

If the item being drawn is currently selected, then the **fsState** and **fsStateOld** fields will both be TRUE; they will both be FALSE if the item is not currently selected. The window receiving a WM_DRAWITEM message can use this information to highlight the selected item at the same time it draws the item. If the owner window highlights the item, it should leave the **fsState** and **fsStateOld** fields equal to each other. If the system provides default highlighting for the item (by inverting the item rectangle), the owner window should set the **fsState** field to 1 and the **fsStateOld** field to 0 before returning from the WM_DRAWITEM message.

The owner window also receives a WM_DRAWITEM message when the highlight state of a list item changes. For example, when a user clicks an item, the highlighting must be removed from the currently selected item and the new selection must be highlighted. If these items are owner-drawn, then the owner window receives one WM_DRAWITEM message for each unhighlighted item and one message for the newly highlighted item. To highlight an item, the **fsState** field must equal TRUE and the **fsStateOld** field must equal FALSE. In this case, the application should highlight the item and return the **fsState** and **fsStateOld** fields equal to FALSE. This tells the system not to highlight the item. The application can also return the **fsState** and **fsStateOld** fields with two different values (not equal) and the list box will highlight the item (the default).

To remove highlighting from an item, the **fsState** field must equal FALSE and the **fsStateOld** field must equal TRUE. An application can remove the highlighting and return both the **fsState** and **fsStateOld** fields as FALSE, or it can return the **fsState** field with a value that is not equal to the **fsStateOld** field and the system will remove the highlighting (the default).

The following code fragment shows these selection processes:

```
case WM_DRAWITEM:

    /* Test to see if this is drawing or highlighting/unhighlighting. */

    if (((POWNERITEM) mp2)->fsState !=
            ((POWNERITEM) mp2)->fsStateOld) {

        /* This is either highlighting or unhighlighting. */

        if (((POWNERITEM) mp2)->fsState) {
              .
              . /* Highlight the item. */
              .
        } else {
              .
              . /* Remove the highlighting.*/
              .
        }
```

```
      /* Set fsState = fsStateOld to tell system you did it. */

      ((POWNERITEM) mp2)->fsState =
          ((POWNERITEM) mp2)->fsStateOld = 0;

      return (TRUE); /* Tells list box you did the highlighting. */

    } else {
      .
      . /* Draw the item. */
      .

      /* Check to see if item is selected. */

      if (((POWNERITEM) mp2)->fsState) {
        .
        . /* Highlight the item. */
        .

        /* Set fsState = fsStateOld to tell system you did it. */

        ((POWNERITEM) mp2)->fsState =
            ((POWNERITEM) mp2) ->fsStateOld = 0;
     }
     return (TRUE); /* Tells list box you did the drawing. */
    }
```

14.4 Default List-Box Behavior

This section lists all the messages handled by the predefined list-box window-class procedure.

Message	Description
WM_CREATE	Creates an empty list box with a scroll bar.
WM_DESTROY	Destroys the list and deallocates any memory allocated during its existence.
WM_PAINT	Draws the list box and its items.
WM_CHAR	Processes virtual keys for line and page scrolling. Sends an LN_ENTER notification code for the ENTER key. Returns TRUE if the key is processed; otherwise, passes the message to the **WinDefWindowProc** function.
WM_SETFOCUS	If gaining the focus, creates a cursor and sends an LN_SETFOCUS notication code to the owner window. If losing the focus, destroys the cursor and sends an LN_KILLFOCUS notification code to the owner window.

Message	Description
WM_ADJUSTWINDOWPOS	If the list box has the style LS_NOADJUSTPOS, makes no changes to the **SWP** structure and returns FALSE. Otherwise, adjusts the height of the list box so that a partial item is not shown at the bottom of the list. Returns TRUE if the **SWP** structure is changed.
WM_ENABLE	Enables the scroll bar if there are more items than can be displayed in a list window.
WM_TIMER	Uses timers to control automatic scrolling that occurs when a user drags the mouse pointer outside the window.
WM_BUTTON2DOWN	Returns TRUE; the message is ignored.
WM_BUTTON3DOWN	Returns TRUE; the message is ignored.
WM_MOUSEMOVE	Sets the mouse pointer to the arrow shape and returns TRUE to show that the message was processed.
WM_VSCROLL	Handles scrolling indicated by the list-box scroll bar.
LM_QUERYITEMCOUNT	Returns the number of items in the list.
LM_INSERTITEM	Inserts a new item in the list according to the position information passed with the message.
LM_SETTOPINDEX	Shows the specified item as the top item in the list window, scrolling the list as necessary.
LM_QUERYTOPINDEX	Returns the zero-based index to the item currently visible at the top of the list.
LM_DELETEITEM	Removes the specified item from the list, redrawing the list as necessary. Returns the number of items remaining in the list.
LM_SELECTITEM	Selects the specified item. If the list is a single-selection list,

Message	Description
	deselects the previous selection. Sends a WM_CONTROL message (with the LN_SELECT code) to the owner window.
LM_QUERYSELECTION	For a single-selection list box, returns the zero-based index of the currently selected item. For multiple-selection list boxes, returns the next selected item or LIT_NONE if no more items are selected.
LM_SETITEMTEXT	Sets the text for the specified item.
LM_QUERYITEMTEXTLENGTH	Returns the length of the specified item text.
LM_QUERYITEMTEXT	Copies the specified item's text to a buffer supplied by the message sender.
LM_SETITEMHANDLE	Sets the specified item handle.
LM_QUERYITEMHANDLE	Returns the specified item handle.
LM_SEARCHSTRING	Searches the list for a match to the specified string.
LM_SETITEMHEIGHT	Sets the item height for the list. All items in the list have the same height.
LM_DELETEALL	Deletes all items in the list.

14.5 Summary

List boxes are control windows that have style bits and that can send and receive messages. These styles and messages are listed in the following sections.

14.5.1 List-Box Styles

The style of a list box determines how it displays its items and how it responds to user input. The following styles are used by list boxes:

LS_MULTIPLESEL Allows more than one list item to be selected at a time.

LS_NOADJUSTPOS Automatically adjusts the list-box height (by default) so that it is a multiple of the item height. This is done so that a list box will not display a partial item at the bottom of the list. The size of the list box can be different than the size specified by the application that created it. The style LS_NOADJUSTPOS tells the list box not to adjust the height of window in response to a WM_SIZE message. Applications that need absolute control over list-box size should use the LS_NOADJUSTPOS style (for example, when a list box needs to completely fill a client area). This style may cause items at the bottom of the box to be displayed partially.

LS_OWNERDRAW Causes the owner window to receive a WM_DRAWITEM message each time an item must be drawn or highlighted.

14.5.2 Messages Sent from a List Box to an Owner Window

Messages sent from a list box to an owner window notify the owner of events in the list box, such as when a user selects an item. The following messages are sent from list boxes to owner windows:

WM_CONTROL Sent to the owner window of the list box when a user event occurs in the list box. This message contains a control code that notifies the owner that an event occurred and indicates the type of event. The following are possible control codes:

Code	Description
LN_ENTER	Sent to the owner window when the user presses ENTER or double-clicks an item while the list box has the focus. This code indicates that the user has chosen an item. The owner window may then query the current selection and respond accordingly.
LN_KILLFOCUS	Sent to the owner window when the list box loses the focus.
LN_SCROLL	Sent to the owner window when the list box scrolls.
LN_SELECT	Sent to the owner window when a different item in the list is selected. The owner window can use this code to query the current selection and respond accordingly.
LN_SETFOCUS	Sent to the owner window when the list box receives the focus.

WM_DRAWITEM Sent to the owner window of a list box with the style LS_OWNERDRAW each time an item must be drawn. The owner window should return TRUE if it actually draws the item; otherwise, it should return FALSE. If the item contains text, the owner window can return FALSE and the list box will draw the item. This message allows the owner window to draw the item or allows default text drawing by the list box. This message contains a structure with a presentation space and a bounding rectangle in which to draw the item. It is also sent when an item must be highlighted. The owner window can handle the highlighting or indicate that the list box should handle this process.

WM_MEASUREITEM This message is sent to the owner window of a list box that has the style LS_OWNERDRAW. The owner window returns a value that is the height of an item in the list. Note that all items in a list must have the same height and that this must be greater than or equal to 1. It is not necessary to specify the width of an item since the item is clipped to the width of the list box when drawn.

14.5.3 Messages Sent to a List Box

Messages sent to a list box set or query the list data. The following messages are sent to list-box controls:

LM_DELETEALL Deletes all items in the list.

LM_DELETEITEM Deletes a specified item from the list.

LM_INSERTITEM Inserts an item in the list box. Items can be inserted at a specified index, at the beginning or the end, or sorted in ascending or descending order.

LM_QUERYITEMCOUNT Returns the number of items in the list box.

LM_QUERYITEMHANDLE Returns the item handle for the specified item.

LM_QUERYITEMTEXT Copies the text for a specified item into a buffer provided by the caller. The size of the required buffer can be determined by sending an LM_QUERYITEMTEXTLENGTH message for the item.

LM_QUERYITEMTEXTLENGTH Returns the length of the text for a specified item in the list.

LM_QUERYSELECTION Returns the index of the selected item in the list or LIT_NONE if no item is selected. In a multiple selection list, this message returns the first selected item, starting at a specified index.

LM_QUERYTOPINDEX Returns the index of the item currently displayed at the top of the list-box window or LIT_NONE if the list is empty.

LM_SEARCHSTRING Searches the list for a match with the specified string, returning the first matching item. Match criteria can be set using flags for case sensitivity and substring matching. Another parameter allows the search to start at a specified index, allowing iterative searches to start from the previous matching item.

LM_SELECTITEM Sets the selection state of a specified item. If the list box allows only a single selection, the previous selection is deselected. An index of LIT_NONE deselects all items in the list. Sending an LM_SELECTITEM message with LIT_NONE set to a multiple-selection list box does not deselect anything. However, it does remove the cursor.

LM_SETITEMHANDLE Sets the item handle for the specified item.

LM_SETITEMHEIGHT Sets the height of all items in the list, redrawing the list box and all visible items.

LM_SETITEMTEXT Sets the text of a specified item in the list.

LM_SETTOPINDEX Displays a specified item at the top of the list box, scrolling the list as necessary.

Chapter

15

Static Controls

15.1 Introduction 195

15.2 About Static Controls 195

15.3 Using Static Controls in an Application 195

15.3.1 Static Controls in a Dialog Window 195

15.3.2 Static Controls in Client Windows 196

15.3.3 Changing the Static-Control Handle 196

15.4 Default Static-Control Behavior 197

15.5 Summary 198

15.5.1 Static-Control Styles 198

15.5.2 Messages Sent to Static Controls 199

15.1 Introduction

This chapter describes how to use static control windows in your applications. You should already be familiar with the following topics:

- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Window messages and message queues

15.2 About Static Controls

Static controls are simple text fields, bitmaps, or icons that can be used to label, enclose, or separate other control windows. Static controls do not accept user input and they do not send notification messages to their owners.

Static controls have style bits that determine whether the control displays text, draws a simple box containing text, displays an icon or a bitmap, or shows framed or unframed colored boxes. The various styles for static controls are discussed individually in Section 15.5.

Static text controls are most commonly used in dialog windows as labels. Iconic and bitmap static controls can be used to provide graphic objects in dialog windows. One advantage of static controls is that, once created, they provide labels and graphics and require little attention from an application.

Static controls never accept the keyboard focus. When a static control receives a WM_SETFOCUS message, or when a user clicks a static control, that control advances the focus to the next sibling window that is not a static control. If there are no sibling windows to the static control, the focus is given to the owner of the static control.

15.3 Using Static Controls in an Application

Static controls can be used in dialog windows and client windows. There is usually very little interaction between an application and the static control once the control is created. However, an application can change the state of the static control.

Static controls also are associated with a long-word handle which may be set and queried by an application. By default, icon and bitmap static controls use this handle to contain a handle to their display object. Applications can modify this handle by using the SM_SETHANDLE and SM_QUERYHANDLE messages to change static-control appearance.

15.3.1 Static Controls in a Dialog Window

Static controls are most commonly used as labels and separators in dialog windows. As such, they are defined as part of a dialog template in the application resource file. Once the dialog window is displayed, the static controls do not interact with the application unless the application changes their state. Typically, an application might change the text or position of a static control. These operations are achieved by using the **WinSetWindowText** and **WinSetWindowPos** functions.

When defining icon or bitmap static controls in a dialog template, the text for the control is interpreted as the resource ID of the bitmap or the icon. There are two ways that the text can be interpreted. If the first byte is '#', the rest of the text is assumed to be an ASCII decimal representation of the icon-or bitmap-resource ID. If the first byte of the text is 0xFF, the second byte is the low byte of the resource ID, and the third byte is the high byte of the resource ID. The following are two sample Resource Compiler definitions:

```
CONTROL "#256",ID_ICON1, 140, 20, 0, 0, WC_STATIC,
    SS_ICON | WS_VISIBLE

CONTROL 0xFF000100, ID_ICON2, 140, 20, 0, 0, WC_STATIC,
    SS_ICON | WS_VISIBLE
```

Each definition specifies an SS_ICON static control that uses an icon with resource ID 256 (0x0100). The icon is assumed to be in the current application resource file.

The window handle for a static-control window can be obtained by calling the **WinWindowFromID** function using the dialog-window handle and the window ID of the static control as defined in the dialog template.

15.3.2 Static Controls in Client Windows

Applications can create static-control windows in non-dialog windows by calling the **WinCreateWindow** function with a WC_STATIC window class. The appearance of the control is defined by the style parameter to the **WinCreateWindow** function.

If the application creates a control with SS_ICON or SS_BITMAP style, it must ensure that the resource ID specified by the window text corresponds to an actual resource in the application resource file or the static control will not be created.

Once created, the static-control window can be moved and sized just like any other child window. An application can obtain the window handle of the static control by calling the **WinWindowFromID** function, supplying the parent window and the window ID of the static control.

15.3.3 Changing the Static-Control Handle

The static-window handle contains a handle to an icon or bitmap. Applications can query and set this handle by using the SM_QUERYHANDLE and SM_SETHANDLE messages. Setting the handle causes the static item to be redrawn, so the handle must be a valid icon or bitmap handle.

For non-icon and non-bitmap static-control items, the handle is available for use by the application and has no effect on the appearance of the control.

15.4 Default Static-Control Behavior

This section describes all the messages specifically handled by the predefined static-control class.

Message	Description
WM_PAINT	Draws the static control based on its style attributes.
WM_CREATE	Sets the window text for static-text controls. Loads the bitmap or icon resource for the bitmap or icon static controls. Returns TRUE if the resource cannot be loaded.
WM_DESTROY	Frees the text for static-text controls. Destroys the bitmap or icon for the bitmap and icon static controls. The icon for a sysicon static control is not destroyed, because it belongs to the system.
WM_ADJUSTWINDOWPOS	Adjusts the **SWP** structure so that the new window size matches the bitmap, icon, or sysicon dimensions associated with the control.
WM_QUERYWINDOWPARAMS	Returns the requested window parameters.
WM_SETWINDOWPARAMS	Allows the window text to be set for static-text controls only.
WM_ENABLE	Invalidates the entire control window, forcing it to be redrawn.
WM_QUERYDLGCODE	Returns the predefined constant DLGC_STATIC.
WM_MOUSEMOVE	Sets the mouse pointer to the arrow pointer and returns TRUE.
WM_SETFOCUS	Sets the focus to the next sibling window that can accept the focus, or if no such sibling exists, it sets the focus to the parent window.
SM_SETHANDLE	Sets the handle associated with the static control and invalidates the control window, forcing it to be redrawn.
SM_QUERYHANDLE	Returns the handle associated with the static-control window.

Message	Description
WM_MATCHMNEMONIC	Returns TRUE if the mnemonic passed in the *mp1* parameter matches the mnemonic in the control-window text.
WM_HITTEST	Returns HT_TRANSPARENT for the following static control styles: SS_GROUPBOX, SS_FGNDRECT, SS_HALFTONERECT, SS_BKGNDRECT, SS_FGNDFRAME, SS_HALFTONEFRAME, SS_BKGNDFRAME. For other styles, returns the **WinDefWindowProc** function.

15.5 Summary

The following sections describe the styles and messages associated with static-control windows.

15.5.1 Static-Control Styles

The following styles are associated with static-control windows:

SS_BITMAP Draws a bitmap. The resource ID for the bitmap resource is determined in the same way as for SS_ICON controls. The bitmap resource is assumed to be in the current application resource file.

SS_BKGNDFRAME Draws a rectangular frame with the current background color.

SS_BKGNDRECT Draws a filled rectangle with the current background color.

SS_FGNDFRAME Draws a rectangular frame with the current foreground color.

SS_FGNDRECT Draws a filled rectangle with the current foreground color.

SS_GROUPBOX Draws a box with control text in the upper-right corner of the box. This style is useful for enclosing groups of radio buttons or check boxes in a box.

SS_HALFTONEFRAME Draws a rectangular frame with a halftone pattern in the current foreground color.

SS_HALFTONERECT Draws a filled rectangle with a halftone pattern in the current foreground color.

SS_ICON Draws an icon. The text of the control is interpreted as the resource ID of the icon. The bytes that make up the text can be interpreted as numeric values or as ASCII representations of numbers, depending on the value of the first byte. If the first byte is '#', the remaining text is assumed to be an ASCII decimal representation of the icon resource ID. If the first byte of the text is 0xFF, the second byte is the low byte of the resource ID and the third byte is the high byte of the resource ID.

SS_SYSICON Displays a system pointer icon. The ID of the icon is extracted from the control text, as in the SS_ICON style. To display this icon, the system calls the **WinQuerySysPointer** function with the specified ID.

SS_TEXT Allows various formatting options to be combined with the SS_TEXT style to produce formatted text in the boundaries of the control. The formatting option flags are the same as those used for the **WinDrawText** function.

15.5.2 Messages Sent to Static Controls

The following messages are associated with static-control windows:

SM_QUERYHANDLE—Returns the private handle for the static control. For controls with SS_ICON, SS_BITMAP, or SS_SYSICON style, the handle of the icon or bitmap is returned. For all other types of static controls, the handle is available for application-defined purposes.

SM_SETHANDLE—Sets the private handle for the static control. Static controls with SS_ICON, SS_BITMAP, or SS_SYSICON style automatically use this handle to store the icon or bitmap when the control is created. An application can change the appearance of these controls by setting the handle. For all other types of static controls, the handle is available for application-defined purposes.

Chapter

16

Scroll-Bar Controls

16.1 Introduction 203

16.2 About Scroll Bars 203
16.2.1 Scroll-Bar Creation 203
16.2.2 Scroll-Bar Range and Position 204
16.2.3 Scroll-Bar Notification Messages 204
16.2.4 Scroll Bars and the Keyboard 206

16.3 Using Scroll Bars 207
16.3.1 Creating Scroll Bars 208
16.3.2 Retrieving a Scroll-Bar Handle............................. 208
16.3.3 Using the Scroll-Bar Range and Position 209

16.4 Summary 210
16.4.1 Messages 210
16.4.2 System Values.. 210

16.1 Introduction

This chapter describes creating and using scroll bars in Presentation Manager applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Windows
- Frame windows
- Messages and message queues
- Control windows

16.2 About Scroll Bars

Scroll bars are control windows that convert mouse and keyboard input into integer values. Applications typically use scroll bars to control scrolling the contents in a client window.

A scroll bar has several parts: the bar, arrows, and the slider. These are found on vertical and on horizontal scroll bars. Arrows are located at each end of a scroll bar. The left scroll arrow, on the left side of a horizontal scroll bar, lets the user scroll toward the left in a document. The right scroll arrow lets the user scroll toward the right. The upper scroll arrow lets the user scroll upward in the document. The lower scroll arrow lets the user scroll downward.

The slider, a hollow box, lies between the two scroll arrows. The slider position in the scroll bar the reflects current value of the scroll bar. When the slider is against the left or top scroll arrow, the scroll-bar value is at a minimum; when the slider is against the right or bottom arrow, the scroll-bar value is at a maximum. The area between the scroll arrows is called the slider background.

Scroll bars monitor the slider position and send notification messages to the owner window when the slider position changes through mouse or keyboard input.

Scroll bars are often used in frame windows. A frame window automatically places scroll bars at the right and bottom sides of a window, depending on the scroll-bar style. The frame window automatically passes scroll-bar messages on to its client window. The client window handles these messages by adjusting the display. The client window can also send messages directly to the scroll bar, directing the scroll bar to adjust its range so that it will map to the data scrolling in the client window.

An application can use scroll bars as stand-alone controls in any size or shape, at any position, in any sort of window. Scroll bars can be used as parts of other controls—for example, list-box controls use a scroll bar to let the user view items when the list box is too small to display all the items.

16.2.1 Scroll-Bar Creation

An application creates a scroll bar by using the preregistered window class WC_SCROLLBAR. The application can create a scroll bar by using the **WinCreateWindow** function. There are two scroll-bar styles: SBS_HORZ and SBS_VERT. The style SBS_HORZ creates a horizontal scroll bar; SBS_VERT creates a vertical scroll bar. Although most applications specify an owner when creating a scroll bar, an owner is not required. If no owner is specified, the scroll bar does not send notification messages. An application can retrieve the scroll bar's current slider position by sending the SBM_QUERYPOS message to the scroll bar.

An application can specify class-specific data when creating a scroll bar. The **SBCDATA** structure specifies the initial range and slider position for the scroll bar.

If a scroll bar is a descendant of a frame window, its position relative to the parent window may change when the frame window's position changes. Frame windows draw scroll bars relative to the upper-left corner of the frame window (rather than the lower-left corner). The frame window may adjust the *y*-coordinate of the scroll-bar position. This is desirable when the scroll bar is an immediate child window of the frame window, but may be undesirable if the scroll bar is not an immediate child window.

16.2.2 Scroll-Bar Range and Position

Every scroll bar has a range and a slider position. The range specifies the maximum and minimum values for the slider position. As the user moves the slider in the scroll bar, its position is reported as an integer value in the range. If the slider position is the minimum value, the slider is at the top of a vertical scroll bar or at the left end of a horizontal scroll bar. If the slider position is the maximum value, the slider is at the bottom or right end of the vertical or horizontal scroll bar, respectively.

An application can adjust the range to convenient integer values by using the SBM_SETSCROLLBAR message (or initially, by using the **SBCDATA** structure). This makes it easy to translate the slider position to a value that corresponds to the data being scrolled. For example, an application that has 260 lines of text to display in a window that can show only 16 lines at a time can set the vertical scroll-bar range to 1 through 244. When the slider is at position 1, the first line is at the top of the window. When the slider is at position 244, the last line is at the bottom of the window.

To keep the scroll-bar range in useful relationship with the data, an application must adjust the range whenever the data or the size of the window changes. This means an application should adjust the range as part of processing WM_SIZE messages.

An application must move the slider in the scroll bar. Although the user makes a request for scrolling in the scroll bar, the scroll bar does not update the slider position. Instead, it passes the request to the owner window. The owner window must scroll the data and update the slider position by using the SBM_SETPOS message. Because the application controls the slider movement, it can move the slider in increments that work best for the data being scrolled.

16.2.3 Scroll-Bar Notification Messages

The scroll bar sends notification messages to the scroll-bar owner whenever the user clicks the scroll bar. The WM_VSCROLL and WM_HSCROLL messages are the notification messages for vertical and horizontal scroll bars, respectively. If the scroll bar is a frame-control window, the message is passed by the frame window to the client window.

Each notification message includes the scroll-bar identifier, the specific scroll-bar command code that corresponds to the user's action, and, in some cases, the current position of the slider. If a scroll bar is created as part of a frame-control window, the scroll-bar identifier is one of the predefined constants FID_VERTSCROLL or FID_HORZSCROLL. Otherwise, it is the identifier given in the **WinCreateWindow** function.

The scroll-bar command codes specify the action the user has taken. The code specifies where the user has clicked the mouse. MS OS/2 user-interface guidelines recommend certain responses for each action. The following is a list of the command codes and the recommended responses. In each case, a "unit" is defined by the application and should be appropriate for the given data. For example, when scrolling text vertically, the unit is typically a line.

Command code	Description
SB_LINEUP	User clicked the top scroll arrow. Decrement the slider position by one and scroll toward the top of the data by one unit.
SB_LINEDOWN	User clicked the bottom scroll arrow. Increment the slider position by one and scroll toward the bottom of the data by one unit.
SB_LINELEFT	User clicked the left scroll arrow. Decrement the slider position by one and scroll toward the left end of the data by one unit.
SB_LINERIGHT	User clicked the right scroll arrow. Increment the slider position by one and scroll toward the right end of the data one unit.
SB_PAGEUP	User clicked the scroll-bar background above the slider. Decrement the slider position by the number of data units in the window and scroll toward the top of the data by the same number of units.
SB_PAGEDOWN	User clicked the scroll-bar background below the slider. Increment the slider position by the number of data units in the window and scroll toward the bottom of the data by the same number of units.
SB_PAGELEFT	User clicked the scroll-bar background to the left of the slider. Decrement the slider position by the number of data units in the window and scroll toward the left end of the data by the same number of units.
SB_PAGERIGHT	User clicked the scroll-bar background to the right of the slider. Increment the slider position by the number of data units in the window and scroll toward the right end of the data by the same number of units.

Command code	Description
SB_SLIDERTRACK	User is dragging the slider. Applications that draw data quickly can set the slider to the position given in the message and scroll the data by the same number of units the slider has moved. Applications that cannot draw data quickly should wait for the SB_SLIDERPOSITION code before moving the slider and scrolling the data.
SB_SLIDERPOSITION	User released the slider after dragging it. Set the slider to the position given in the message and scroll the data by the same number of units the slider has moved.
SB_ENDSCROLL	User released the mouse after holding it on an arrow or in the scroll-bar background. No action is necessary.

If the scroll-bar command code is either SB_SLIDERPOSITION or SB_SLIDERTRACK, indicating that the user is moving the scroll-bar slider, the notification message also contains the current position of the slider.

The owner window can send a message to the scroll bar to read its current value and range or to reset its current value. The owner window can adjust data controlled by the scroll bar to reflect any changes in the state of the scroll bar.

An application can disable a scroll bar by using the **WinEnableWindow** function. A disabled scroll-bar window ignores the user's actions, sending out no notification messages when the user tries to manipulate it. If an application has no data to scroll or all data fits in the client window, it should disable the scroll bar.

Scroll bars have their own system color, SYSCLR_SCROLLBAR. This color is used to paint the scroll-bar background. Other system colors are used in other parts of the scroll bar.

16.2.4 Scroll Bars and the Keyboard

When a scroll bar has the keyboard focus, it generates notification messages for the following keys:

Key	Command code
UP	SB_LINEUP or SB_LINELEFT
LEFT	SB_LINEUP or SB_LINELEFT
DOWN	SB_LINEDOWN or SB_LINERIGHT
RIGHT	SB_LINEDOWN or SB_LINERIGHT
PAGE UP	SB_PAGEUP or SB_PAGELEFT
PAGE DOWN	SB_PAGEDOWN or SB_PAGERIGHT

If an application uses scroll bars to scroll data but does not give the scroll bar the input focus, the window with the focus should process keyboard input itself. The window can generate scroll-bar notification messages or carry out the indicated scrolling. The following list gives the keys a window should process and what action to take for each:

Key	Command
UP	SB_LINEUP
DOWN	SB_LINEDOWN
PAGE UP	SB_PAGEUP
PAGE DOWN	SB_PAGEDOWN
CONTROL+HOME	SB_SLIDERTRACK with slider set to minimum position
CONTROL+END	SB_SLIDERTRACK with slider set to maximum position
LEFT	SB_LINELEFT
RIGHT	SB_LINERIGHT
CONTROL+PAGE UP	SB_PAGELEFT
CONTROL+PAGE DOWN	SB_PAGERIGHT
HOME	SB_SLIDERTRACK with slider set to minimum position
END	SB_SLIDERTRACK with slider set to maximum position

Vertical scroll bars that are part of list boxes have the following keyboard interface:

Key	Command
CONTROL+UP	SB_SLIDERTRACK with slider set to minimum position
CONTROL+DOWN	SB_SLIDERTRACK with slider set to maximum position
F7	SB_PAGEUP
F8	SB_PAGEDOWN

The application must implement the suggested scroll-bar/keyboard interface. This can be accomplished by appropriate handling of WM_CHAR messages.

16.3 Using Scroll Bars

This section explains how to create and use scroll bars in an application. Scroll bars are most often used in frame windows to let the user scroll data in the corresponding client window.

16.3.1 Creating Scroll Bars

You can add scroll bars to a frame window by using the FCF_HORZSCROLL flag, the FCF_VERTSCROLL flag, or both flags when creating the frame window with the **WinCreateStdWindow** function. This adds a horizontal and/or a vertical scroll bar to the frame window. Because the frame window owns the scroll bars, it passes notification messages from these controls to the client window.

The following code fragment adds scroll bars to a frame window:

```
/* Set flags for a main window with scroll bars. */

ULONG ulFrameControlFlags =
    FCF_STANDARD | FCF_HORZSCROLL | FCF_VERTSCROLL;

/* Create the window. */

hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE,
    &ulFrameControlFlags, szClientClass, szFrameTitle,
    OL, NULL, OL, &hwndClient);
```

Scroll bars created this way have the window identifier FID_HORZSCROLL or FID_VERTSCROLL. The frame window determines the size and position of the scroll bars. A frame window uses the standard size specified by the system values SV_CXVSCROLL and SV_CYHSCROLL. The position is always the right and bottom edges of the frame window.

Another way to create scroll bars is by using the **WinCreateWindow** function. This is most common for stand-alone scroll bars. Creating scroll bars in this way lets you set the size and position of the scroll bars. You can also specify the window to receive notification messages.

The following code fragment creates a stand-alone scroll bar:

```
HWND hwndScroll;                /* scroll-bar handle             */

hwndScroll = WinCreateWindow(
    hwndClient                  /* scroll-bar parent             */
    WC_SCROLLBAR,               /* preregistered scroll-bar class */
    NULL,                       /* no window title               */
    SBS_VERT | WS_VISIBLE,      /* vertical style and visible    */
    10, 10,                     /* position                      */
    20, 100,                    /* size                          */
    hwndClient,                 /* owner                         */
    HWND_TOP,                   /* Z-order position              */
    1,                          /* scroll-bar ID                 */
    NULL,                       /* no class-specific data        */
    NULL);                      /* no presentation parameters    */
```

16.3.2 Retrieving a Scroll-Bar Handle

If you create a scroll bar as a child window of the frame window by using the **WinCreateStdWindow** function, you need a way to retrieve the scroll-bar handle. One way is to use the **WinWindowFromID** function, the frame-window handle, and a predefined identifier (such as FID_HORZSCROLL or FID_VERTSCROLL) to retrieve the scroll-bar handle:

```
hwndHorzScroll = WinWindowFromID(hwndFrame, FID_HORZSCROLL);
hwndVertScroll = WinWindowFromID(hwndFrame, FID_VERTSCROLL);
```

If the standard frame window includes a client window, you can use that handle to access the scroll bars. The idea is to retrieve the frame-window handle first, then the scroll-bar handle. This is illustrated by the following code fragment:

```
/* Get a handle to the horizontal scroll bar. */

hwndScroll = WinWindowFromID(
    WinQueryWindow(hwndClient, QW_PARENT, FALSE),
    FID_HORZSCROLL);
```

16.3.3 Using the Scroll-Bar Range and Position

You can initialize a scroll bar's current value and range to nondefault values by sending the **SBCDATA** structure with class-specific data for a call to the **WinCreateWindow** function:

```
SBCDATA sbcd;

/* Set up scroll-bar control data. */

sbcd.posFirst = 200;
sbcd.posLast = 400;
sbcd.posThumb = 300;

/* Create the scroll bar. */

hwndScroll = WinCreateWindow(hwndClient, WC_SCROLLBAR, NULL,
    SBS_VERT | WS_VISIBLE,
    10, 10, 20, 100,
    hwndClient, HWND_TOP, 1,
    &sbcd,       /* class-specific data */
    NULL);
```

You can adjust a scroll-bar value and range by sending it an SBM_SETSCROLLBAR message:

```
/* Set the scroll-bar value and range. */

WinSendMsg(hwndScroll, SBM_SETSCROLLBAR,
    MPFROM2SHORT(300, 0),
    MPFROM2SHORT(200, 400));
```

You can read a scroll-bar value by sending it an SBM_QUERYPOS message:

```
USHORT usSliderPos;

/* Read the scroll-bar value. */

usSliderPos = (USHORT) WinSendMsg(hwndScroll,
    SBM_QUERYPOS, NULL, NULL);
```

Similarly, you can set a scroll-bar value by sending an SBM_SETPOS message:

```
/* Set the vertical scroll-bar value. */

WinSendMsg(hwndScroll, SBM_QUERYPOS, MPFROM2SHORT(300, 0), NULL);
```

You can read a scroll-bar range by sending it an SBM_QUERYRANGE message:

```
MRESULT mr;
USHORT iMinimum, iMaximum;

/* Read the vertical scroll-bar range. */

mr = WinSendMsg(hwndScroll, SBM_QUERYRANGE, NULL, NULL);

iMinimum = SHORT1FROMMR(mr);  /* minimum in the low word  */
iMaximum = SHORT2FROMMR(mr);  /* maximum in the high word */
```

16.4 Summary

This section lists the messages and system values that applications use to create and control scroll-bar control windows.

16.4.1 Messages

Applications use the following messages to create and control scroll bars:

SBM_QUERYHILITE Sent to a scroll bar to obtain its highlight state.

SBM_QUERYPOS Sent to a scroll bar to obtain the current value of the scroll bar.

SBM_QUERYRANGE Sent to a scroll bar to obtain the scroll-bar range.

SBM_SETHILITE Sent to a scroll bar to set the highlight state.

SBM_SETPOS Sent to a scroll bar to set the current value of the scroll bar.

SBM_SETSCROLLBAR Sent to a scroll bar to set the current value and range.

WM_HSCROLL Sent by a horizontal scroll bar to its owner window when the user changes the state of the scroll bar. The high word of the second parameter (*mp2*) contains a scroll-bar command code.

WM_VSCROLL Sent by a vertical scroll bar to its owner window when the user changes the state of the scroll bar. The high word of the second parameter (*mp2*) contains a scroll-bar command code.

16.4.2 System Values

Applications use the following system values to create and control scroll bars:

SV_CXHSCROLLARROW Width (in pels) of the scroll-arrow area in a horizontal scroll bar.

SV_CXVSCROLL Width (in pels) of a standard vertical scroll bar.

SV_CYHSCROLL Height (in pels) of a standard horizontal scroll bar.

SV_CYVSCROLLARROW Height (in pels) of the scroll-arrow area in a vertical scroll bar.

SV_FIRSTSCROLLRATE Initial rate at which a scroll bar sends notification messages when the user clicks the scroll arrows or scroll-bar background.

SV_SCROLLRATE Similar to SV_FIRSTSCROLLRATE, this is the rate at which the scroll bar sends messages.

SYSCLR_SCROLLBAR Color for drawing scroll-bar backgrounds.

TID_SCROLL Timer ID for a reserved scrolling timer. This timer is used for sending notification messages when a scroll arrow or scroll-bar background is clicked.

Chapter

17

Menus

17.1 Introduction 215

17.2 About Menus 215

17.2.1 Menu-Bar and Pull-Down Menus 215

17.2.2 System Menu 216

17.2.3 Menu-Item Styles 217

17.2.4 Menu-Item Attributes 217

17.3 Defining Menu Items in a Resource File 217

17.4 Menu Data Structures 219

17.4.1 Menu Template 220

17.5 Using Menus in your Applications 221

17.5.1 Including a Menu in a Standard Window 222

17.5.2 Adding Menus to a Dialog Window 223

17.5.3 Accessing the System Menu 223

17.5.4 Responding to a User's Menu Choice 223

17.5.5 Using Menus with the Keyboard 224

17.5.6 Using Keyboard Accelerators 225

17.5.7 Help Item in the Menu Bar 226

17.5.8 Setting and Querying Menu-Item Attributes 226

17.5.9 Setting and Querying Menu-Item Contents 227

17.5.10 Adding and Deleting Menu Items 227

17.5.11 Owner-Drawn Menu Items 230

17.6 Summary 232

17.6.1 Menu-Item Styles 232

17.6.2 Menu-Item Attributes 233

17.6.3 Menu Functions 233

17.6.4 Messages Sent from a Menu to an Owner Window 234

17.6.5 Messages Sent to a Menu 235

17.1 Introduction

This chapter describes how to use menus in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Accelerator tables
- Creating standard frame windows
- Window messages and message queues

17.2 About Menus

Menus are windows that contain a list of items. These items can be text strings, bitmaps, or images drawn by the application. Menus allow the user to use the mouse or keyboard to choose from a predetermined list of choices. When a user makes a choice from a menu, the menu posts a message containing the item's unique menu-item identifier to the menu's owner window.

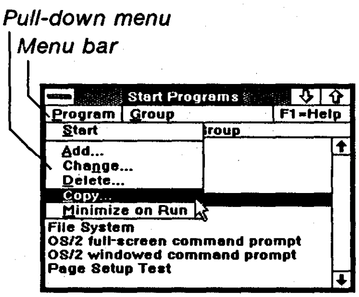
Typically, an application defines its menus by using Resource Compiler and associates the menus with an frame window when the window is created. Applications can also create menus by filling in menu-template data structures and then creating windows with the WC_MENU class. Either way, applications can dynamically add, delete, or change menu items by sending messages to menu windows.

Menu windows are always owned by another window; this is important because a menu sends messages to its owner whenever a menu item is highlighted or chosen by the user. Owner windows send messages to menus to add, delete, or change menu items.

17.2.1 Menu-Bar and Pull-Down Menus

Typically, an application uses a menu-bar menu and several pull-down submenus. The menu bar is a child window in the parent window frame. The submenus are normally hidden and become visible when the user makes selections in the menu bar. Figure 17.1 shows a typical menu-bar and submenu layout in a standard frame window:

Figure 17.1
Menu-Bar and Pull-Down Menus



There are two main types of menu items: command items and submenu items. When the user chooses a command item, a command message is immediately posted to the parent window. When the user selects a submenu item, a pull-down menu is displayed from which the user may choose another command item. Since a pull-down menu window can also contain a submenu item, pull-down menus can originate from other pull-down menus. An item in the menu bar may be a command item or a submenu item.

When a command item is selected, either from the menu bar or from a pull-down menu, the menu system posts a WM_COMMAND, WM_HELP, or WM_SYSCOMMAND message to the owner window, depending on the menu item's style bits.

The menu bar is a child window of the frame window; the menu-bar window handle is the key to communicating with menus. The handle of the menu-bar window can be obtained by calling the **WinWindowFromID** function with the handle of the parent window and the FID_MENU frame-control identifier. Most messages for menus and submenus can be sent to the menu-bar window. Flags in the messages tell the window whether to search submenus for requested menu items.

17.2.2 System Menu

The System menu in the upper-left corner of a standard frame window is different from the menu-bar and pull-down menus defined by the application. The System menu is controlled and defined almost exclusively by the system. Your only decision about the System menu is whether or not to include it when creating a frame window. (It is unusual for a frame window not to include a System menu.) The System menu generates WM_SYSCOMMAND messages instead of WM_COMMAND messages. Most applications simply allow the default behavior for WM_SYSCOMMAND messages.

If necessary, you can obtain the handle of the System menu by calling the **WinWindowFromID** function with the handle of the parent (frame) window and the FID_SYSMENU frame-control identifier. The application can add, delete, and change System-menu entries.

17.2.3 Menu-Item Styles

All menu items have a combination of style bits that determine what kind of data the item contains and what kind of message it generates when it is chosen by the user. For instance, a menu item can have the MIS_TEXT, MIS_BITMAP, or other styles, specifying what kind of display object visually represents the menu item on the screen. Other styles determine what kinds of messages the item sends to its owner window and whether the owner window draws the item. Menu-item styles typically do not change during program execution, but they can be queried and set by sending MM_SETITEM and MM_QUERYITEM messages to the menu with the identifier of the item. The possible menu-item styles are described in Section 17.6.1.

17.2.4 Menu-Item Attributes

Menu items have attributes that determine how they are displayed and whether or not the user can choose them. Menu-item attributes can be set and queried by sending MM_SETITEMATTR and MM_QUERYITEMATTR messages with the identifier of the item to the menu-bar menu window. If the specified item is in a submenu, you must set a flag in the message so that submenus are searched for the item. The possible attributes of a menu item are listed in Section 17.6.2.

17.3 Defining Menu Items in a Resource File

A menu resource consists of command items and submenu items. One menu resource typically represents the menu bar and all its submenus. An application can specify the identifier of the menu resource when creating a standard window, or it can load the menu resource directly by using the **WinLoadMenu** function. A menu-item definition is organized as follows:

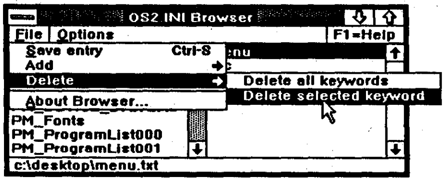
MENUITEM *item text, item identifier, item style, item attributes*

The menu resource-definition file specifies the text of each item in the menu, unique identifier, its style and its attributes, and whether it is a command item or a submenu item. Following is sample source code that defines a menu resource for Resource Compiler. The code defines a menu with three submenu items in the menu bar (File, Edit, and Fonts) and a command item (Help). Each submenu has several command items, and the Fonts submenu has two other submenus within it.

```
MENU ID_MENU_RESOURCE
BEGIN
    SUBMENU "~File", IDM_FILE
        BEGIN
            MENUITEM "~Open...",       IDM_FI_OPEN
            MENUITEM "~Close\tF3",     IDM_FI_CLOSE, MIS_DISABLED
            MENUITEM "~Quit",          IDM_FI_QUIT
            MENUITEM "",               IDM_FI_SEP1, MIS_SEPARATOR
            MENUITEM "~About Sample",  IDM_FI_ABOUT
        END
    SUBMENU "~Edit", IDM_EDIT
        BEGIN
            MENUITEM "~Undo",          IDM_ED_UNDO, 0, MIA_DISABLED
            MENUITEM "",               IDM_ED_SEP1, MIS_SEPARATOR
            MENUITEM "~Cut",           IDM_ED_CUT
            MENUITEM "C~opy",          IDM_ED_COPY
            MENUITEM "~Paste",         IDM_ED_PASTE
            MENUITEM "C~lear",         IDM_ED_CLEAR
        END
    SUBMENU "Font", IDM_FONT
        BEGIN
            SUBMENU "Style",           IDM_FONT_STYLE
                BEGIN
                    MENUITEM "Plain",  IDM_FONT_STYLE_PLAIN
                    MENUITEM "Bold",   IDM_FONT_STYLE_BOLD
                    MENUITEM "Italic", IDM_FONT_STYLE_ITALIC
                END
            SUBMENU "Size",            IDM_FONT_SIZE
                BEGIN
                    MENUITEM "10",     IDM_FONT_SIZE_10
                    MENUITEM "12",     IDM_FONT_SIZE_12
                    MENUITEM "14",     IDM_FONT_SIZE_14
                END
        END
    MENUITEM "F1=Help", 0x00, MIS_TEXT | MIS_BUTTONSEPARATOR | MIS_HELP
END
```

Figure 17.2 shows how the submenus within a Delete submenu are displayed.

Figure 17.2
Submenus



You can indicate a mnemonic keystroke for the menu item by preceding that character in the item text with a tilde, as in "~File". The user can choose that item by pressing the mnemonic key when the menu is active. (The menu bar is active when the user presses and releases the ALT key, and the first item in the menu bar is highlighted. A pull-down menu is active when it is open.)

In addition to mnemonics, a menu item can have an associated keyboard accelerator. Accelerators are different from mnemonics, in that the menu does not have to be active for the accelerator key to work. If a menu item has a keyboard accelerator associated with it, the corresponding menu item should display the accelerator to the right of the menu item. This is done by placing a tab character (\t) in the menu text before the characters that should be displayed on the right. For example, if the Close item had the F3 function key as its keyboard accelerator, the text for the item would be "Close\tF3". For more information on accelerators, see Section 17.5.6.

Each entry that defines a menu item specifies the text for the item, its identifier, and the style and attributes of the item. A menu item that has no specification for style or attributes has the default style of MIS_TEXT and all attribute bits off, indicating that the item is enabled. The MIS_SEPARATOR style identifies nonselectable lines between menu items.

To define a menu item with the MIS_BITMAP style, an application should use a tool such as Icon Editor to create a bitmap, include the bitmap in the application's resource-definition file, and define a menu in the file (as shown in the following code fragment). The text for the bitmap menu items is an ASCII representation of the resource identifier of the bitmap resource to be displayed for that item.

```
/* Bring externally created bitmaps into the resource file. */

BITMAP 101 button.bmp
BITMAP 102 hirest.bmp
BITMAP 103 hizoom.bmp
BITMAP 104 hired.bmp

/* Connect a menu item with a bitmap. */

SUBMENU "~Bitmaps", IDM_BITMAP
    BEGIN
        MENUITEM "#101", IDM_BM_01, MIS_BITMAP
        MENUITEM "#102", IDM_BM_02, MIS_BITMAP
        MENUITEM "#103", IDM_BM_03, MIS_BITMAP
        MENUITEM "#104", IDM_BM_04, MIS_BITMAP
    END
```

17.4 Menu Data Structures

There are two main data structures that define the contents of a menu: the menu-item structure and the menu-template structure. The menu-item structure defines a single menu item, and the menu-template structure contains all the menu items that make up a menu resource, including the menu bar and all its pull-down menus.

A single menu item is defined by the **MENUITEM** data structure. This data structure is used with the MM_INSERTITEM message to insert items into a menu, or to query and set item characteristics with the MM_QUERYITEM and MM_SETITEM messages. The **MENUITEM** data structure has the following form:

```
typedef struct _MENUITEM {
    SHORT  iPosition;
    USHORT afStyle;
    USHORT afAttribute;
    USHORT id;
    HWND   hwndSubMenu;
    ULONG  hItem;
} MENUITEM;
```

The values of most of the fields in the data structure can be derived directly from the resource-definition file shown in Section 17.3. The last field in the structure, **hItem**, depends on the style of the menu item.

The **iPosition** field specifies the ordinal position of the item within its menu window. If the item is part of the menu bar, **iPosition** gives its relative left-to-right position, with zero being the leftmost item. If the item is part of a submenu, **iPosition** gives its relative top-to-bottom and left-to-right position, with zero being the upper-left item. An item with the MIS_BREAKSEPARATOR style in a pull-down menu will cause a new column to begin.

The **afStyle** field contains the style bits of the item. The **afAttribute** field contains the attribute bits.

The **id** field contains the identifier for the menu item. The identifier should be unique but does not have to be. When multiple items have the same identifier, they post the same command number in the WM_COMMAND, WM_HELP, and WM_SYSCOMMAND messages. Also, any message that specifies a menu item with a nonunique identifier will find the first item that has that identifier.

The **hwndSubMenu** field contains the window handle of a pull-down menu window (if the item is a submenu item). The **hwndSubMenu** field is NULL for command items.

The **hItem** field contains a handle to the display object for the item, unless the item has the MIS_TEXT style, in which case **hItem** is NULL. For example, a menu item with the MIS_BITMAP style has an **hItem** field that is equal to its bitmap handle.

17.4.1 Menu Template

A menu template is a variable-length data structure that represents the entire menu, including all items and submenus. A menu template is made up of a series of variable-length records. Each record represents a single menu item. If the item is a submenu, the template that describes the submenu is nested after the submenu item record.

The menu template is a representation of the menu as it is defined in the resource-definition file. Typically, applications require information about the internal structure of a menu template only when creating a menu template without using a resource-definition file.

A template is defined as shown in the following code fragment:

```
typedef struct _MT {
    USHORT cb;          /* length of template in bytes            */
    USHORT version;     /* version; set to zero                   */
    USHORT codepage;    /* code page                              */
    USHORT iInputsize;  /* length of input field for host terminals */
    USHORT cMti;        /* count of items                         */
    MTI    rgMti(cMti);
} MT;
```

MS OS/2 version 1.1 sets the version, code-page, and input-size fields to zero, and ignores the contents of these fields if set by an application. The **cMti** field specifies the number of menu-template items that follow. Each menu-template item describes one item in the menu. Since each menu item can require a different amount of storage, the following variable definition of a menu-template item is used:

```
typedef struct _MTI {
    USHORT afStyle;
    USHORT afAttribute;
    USHORT idItem;
    if (afStyle AND MIS_BITMAP)
      CHAR szItemString ? ;
    if (afStyle AND MIS_OWNERDRAW)
      VOID;
    if (afStyle AND MIS_TEXT)
      CHAR szItemString ? ;
    if (aStyle AND MIS_SEPARATOR)
      VOID;
    if (afStyle AND MIS_SUBMENU)
      MT MenuTemplate;
} MTI;
```

The first three fields of a structure for a menu-template item specify the style, attributes, and identifier of the item. The data that follows these fields is determined by the style of the item. The cases can be summarized as follows:

If the **afStyle** field is MIS_TEXT, the data that follows the **idItem** field is a null-terminated string representing the menu-item text.

If the **afStyle** field is MIS_BITMAP, the data that follows **idItem** is a null-terminated string that can represent one of three things:

- If the first byte is NULL, then no bitmap resource is defined; the application provides a bitmap handle for the item.
- If the first byte is "#", subsequent characters make up the decimal representation of the bitmap resource-identifier.
- If neither of the previous cases apply, the handle is set to NULL, and the application must set it manually.

If the **afStyle** field is MIS_OWNERDRAW or MIS_SEPARATOR, there is no data following the **idItem** field.

If the **afStyle** field is MIS_SUBMENU, a complete menu-template structure for the submenu follows the **idItem** field.

17.5 Using Menus in your Applications

Typically, an application that uses menus defines them in a resource-definition file and includes them in a standard frame window. During program execution, the application's window procedure receives messages generated by the menu windows in response to user input, either from the mouse or the keyboard. The application responds to these messages by using the identifier of the menu item that is contained in each menu message.

The application can also load menu resources at run time and associate them with a particular owner window. This is useful for putting menus in windows other than the standard frame window.

The user-interface guidelines specify that a user should not be required to have a mouse to operate an MS OS/2 application. Applications can define keyboard equivalents to menu items to aid users without mice.

It is also suggested that all applications have a Help item in their menu bar. This presents a standard interface for the novice user across all applications. The Help item is defined with a particular style, attributes, and position in the menu, as shown in the resource-definition file in Section 17.3. The Help command item generates a WM_HELP message when chosen by the user, allowing the application to respond appropriately. For more information on the help system, see Chapter 29, "Help."

Applications can change the attributes, style, and contents of menu items, and insert and delete items at run time, to reflect changes in the command environment. An application can also add or delete items from the menu bar or submenus. For example, an application might maintain a menu of the currently available fonts in the system. This application would use **Gpi** calls to determine which fonts were available, and then insert a menu item for each font into a submenu. Furthermore, the application might set the "checked" attribute of the menu item for the currently chosen font. When the user chooses a new font, the application would remove the check-mark attribute from the previous choice and add it to the new choice.

An application can also draw a menu item itself by setting the attribute MIS_OWNERDRAW for the menu item. Typically, this is done by specifying the MIS_OWNERDRAW attribute for the menu item in the resource-definition file, but it can also be done at run time. When the application draws a menu item, it must respond to messages from the menu each time the item needs to be drawn.

17.5.1 Including a Menu in a Standard Window

If you have defined a menu resource in a resource-definition file, you can include the menu in a standard window. You include the menu by using the FCF_MENU attribute flag and specifying the menu resource identifier in a call to the **WinCreateStdWindow** function, as shown in the following code fragment:

```
ULONG lControlStyle = FCF_MENU | FCF_SIZEBORDER |
    FCF_TITLEBAR | FCF_ACCELTABLE;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP,
    WS_VISIBLE,
    &lControlStyle,
    szClassName,
    szClassName,
    OL, NULL,
    ID_MENU_RESOURCE,
    &hwndClient);
```

After you make this call, MS OS/2 automatically includes the menu in the window, drawing the menu bar across the top of the window. When the user chooses an item from the menu, the menu posts the message to the frame window. The frame window passes any WM_COMMAND messages to the client window. (The frame window does not pass WM_SYSCOMMAND messages to the client window.) WM_HELP messages are posted to the focus window. The **WinDefWindowProc** function passes WM_HELP messages to the parent window. If a WM_HELP message is passed to a frame window, the frame window calls the HK_HELP hook. Your client window procedure should process these messages to respond to the user's actions. The details of responding to menu selections are shown in Section 17.5.4.

17.5.2 Adding Menus to a Dialog Window

You may want to use menus in windows that were not created by using the **WinCreateStdWindow** function. For these windows, you can load a menu resource by using the **WinLoadMenu** function and specify the parent window for the menu. **WinLoadMenu** assigns the specified menu resource to the parent window. To see the menu in the window, you must send a WM_UPDATEFRAME message to the parent window after loading the menu resource. This strategy is especially useful for adding menus to a window created as a dialog window, but it can be used no matter what type of window is specified as the parent window.

17.5.3 Accessing the System Menu

Although most applications do not alter the System menu, you can obtain the handle of the System menu by calling the **WinWindowFromID** function with a frame-window handle (or dialog-window handle) and the FID_SYSMENU identifier. Once you have the handle of the System menu, you can access the individual menu items by using predefined constants. For example, the following code fragment shows how to disable the Close menu item in the System menu of a window:

```
HWND hwndSysMenu;

hwndSysMenu = WinWindowFromID(hwndFrame, FID_SYSMENU);

WinSendMsg(hwndSysMenu, MM_SETITEMATTR,
    MPFROM2SHORT(SC_CLOSE, TRUE),
    MPFROM2SHORT(MIA_DISABLED, MIA_DISABLED));
```

17.5.4 Responding to a User's Menu Choice

When a user chooses a menu, item your client window procedure receives a WM_COMMAND message with the low word of the *mp1* parameter equal to the menu identifier of the selected item. Your application should use the menu identifier to guide its response to the selection. Typically, the code in the client window procedure resembles the following code fragment:

```
case WM_COMMAND:
    DoMenuCommand(hwnd, LOUSHORT(mp1));
    return 0;
```

The function that translates the menu identifier into an action typically resembles the following code fragment:

```
VOID DoMenuCommand(hwnd, usItemID)
HWND hwnd;
USHORT usItemID;
{

    /* Test the menu item. */

    switch (usItemID) {
        case IDM_FI_NEW:
            DoNew(hwnd);
            break;
          .
          .
          .
    }
}
```

The menu system sends a WM_MENUSELECT message every time the menu selection changes. The low word of the *mp1* parameter contains the identifier of the item that is changing state and the high word is a 6-bit Boolean value that describes whether or not the item is chosen; the *mp2* parameter contains the handle of the menu.

If the Boolean value is FALSE, the item is selected but not chosen—for example, the user may have moved the cursor or mouse pointer over the item while the button was down. An application can use this message to display help information at the bottom of the application window. The return value is ignored.

If the Boolean value is TRUE, the item is chosen—that is, the user pressed ENTER or released the mouse button when an item was selected. If the application returns FALSE, the menu does not generate a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message, and the menu is not dismissed.

17.5.5 Using Menus with the Keyboard

MS OS/2 is designed to work with or without a mouse or other pointing device. The system provides default behavior to allow a user to interact with menus without using a mouse. The following are the keystrokes that produce this default behavior:

Keystroke	Action
ALT	Toggles into and out of menu mode.
ALT, SPACEBAR	Shows the System menu.
ESC	Backs up one level. If a pull-down menu is displayed, it is canceled. If no pull-down menu is displayed, this keystroke exits from menu mode.

Keystroke	Action
RIGHT	Cycles to the next menu-bar item. If the selected item is at the far-left side of the menu bar, the menu code sends a WM_NEXTMENU message to the frame. The default processing by the frame is to cycle between the application and System menus. (An application can modify this behavior by subclassing the frame window.) If the selected item is in a pull-down menu, the next column in the pull-down menu is selected or the next menu-bar item is selected; this key may also send or process a WM_NEXTMENU message.
LEFT	Works like the RIGHT key, except in the opposite direction. In pull-down menus, this keystroke backs up one column, except when the currently selected item is in the far-left column, in which case the previous pull-down menu is selected.
UP or DOWN	Activates a pull-down menu when in the menu bar. This keystroke selects the next or previous item when in a pull-down menu.
ENTER	Activates a pull-down menu and highlights the first item if an item has a pull-down menu associated with it; otherwise, this keystroke chooses the item as if the user released the mouse button while the item was selected.
Alphabetic characters	Selects the first menu item with the specified character as its mnemonic key. A mnemonic is defined for a menu item by placing a tilde (~) before the character in the menu text. If the selected item has a pull-down menu or secondary menu associated with it, the menu is displayed and the first item is highlighted; otherwise, the item is chosen.

An application does not need to support this default behavior with any unusual code. The application receives a message when a menu item is chosen by using the keyboard just as if it had been chosen by using a mouse.

17.5.6 Using Keyboard Accelerators

Applications can define accelerator tables to make user interaction with menus more efficient. Accelerator tables are resources that associate keystrokes with menu command items. For example, an application can define an accelerator table resource that makes the F8 key generate a WM_COMMAND message that is identical to the message generated when the user chooses the Quit item from the File menu. Accelerator tables provide a shortcut for proficient users that allows them to work more quickly with the application.

A sample resource-definition file for an accelerator table is shown in the following code fragment:

```
ACCELTABLE ID_ACCEL_RESOURCE
BEGIN
    VK_F8, IDM_FILE_QUIT, VIRTUALKEY
    VK_F3, IDM_SEARCH_FIND, VIRTUALKEY
    VK_F1, NULL, VIRTUALKEY, HELP
END
```

The resource-definition file associates keystrokes with menu-item command identifiers. Notice that the definition uses virtual-keystroke definitions that are independent of the particular scan codes generated by different keyboard hardware.

In order to use an accelerator table with a window, the window should be created with the FS_ACCELTABLE style, and the resource identifier of the accelerator table must be the same as the identifier of the window's menu. Alternatively, you can associate an accelerator table with a frame window after the window is created, by calling the **WinSetAccelTable** function.

For more information on keyboard accelerators, see Chapter 18, "Accelerator Tables."

17.5.7 Help Item in the Menu Bar

The user-interface guidelines suggest that all applications have a Help menu item at the far-right end of the menu bar. The item should read "F1=Help", have an identifier of zero, and have the MIS_BUTTONSEPARATOR or MIS_HELP item style. The Help menu item should be the last item in the menu template, so that it is displayed at the far-right end of the menu bar.

The user can use either a mouse or the F1 key to select the Help menu item, if the application uses the system default accelerator table. (For more information on the system default accelerator table, see Section 17.5.6.) The Help item generates a WM_HELP message instead of a WM_COMMAND message.

17.5.8 Setting and Querying Menu-Item Attributes

Menu-item attributes are represented in the **afAttribute** field of the **MENUITEM** data structure. Typically, attributes are set in the resource-definition file of the menu and changed at run time as required. Applications can use the MM_SETITEMATTR and MM_QUERYITEMATTR messages to set and query attributes for a particular menu item. One of the most common uses of these messages is to check and uncheck menu items to let the user know what option is currently selected. For example, if you have a menu item that should toggle between checked and unchecked each time it is chosen, you could use the following code to change the checked attribute. You first send an MM_QUERYITEMATTR message to the menu item to obtain its current checked attribute, then use the exclusive OR operator to toggle the state, and then you send the new attribute-state back to the item by using an MM_SETITEMATTR message.

```
usAttrib = (SHORT) WinSendMsg(hwndMenu,       /* submenu window        */
    MM_QUERYITEMATTR,                         /* message               */
    MPFROMSHORT(itemID),                      /* identifier of item    */
    MPFROMSHORT(MIA_CHECKED));                /* attribute mask        */

usAttrib ^= MIA_CHECKED;     /* XOR to toggle checked attribute      */

WinSendMsg(hwndMenu,                          /* submenu window        */
    MM_SETITEMATTR,                           /* message               */
    MPFROMSHORT(itemID),                      /* identifier of item    */
    MPFROM2SHORT(MIA_CHECKED, usAttrib)); /* attribute mask, value */
```

There are two methods for manipulating individual menu items in submenus. The first is to send MM_SETITEMATTR and MM_QUERYITEMATTR messages to the menu-bar menu, specifying the identifier of the item and setting a flag so that the message searches all submenus for the item. The handle of the menu-bar window can be obtained by calling the **WinWindowFromID** function with the handle of the frame window and the FID_MENU frame-control identifier.

The second method, which is more efficient if you want to work with more than one item in a submenu or set the same item several times, involves two steps:

1 Sending an MM_QUERYITEM message to the menu-bar window with the identifier of the submenu. The updated **MENUITEM** structure contains the window handle of the submenu.

2 Send an MM_QUERYITEMATTR (or MM_SETITEMATTR) message to the submenu window, specifying the identifier of the item in the submenu.

17.5.9 Setting and Querying Menu-Item Contents

Applications can change the contents of a menu item dynamically by sending messages to the menu. For MIS_TEXT items, an MM_SETITEMTEXT message sets the text. The MM_QUERYITEMTEXT message queries the item's text.

For nontext menu items, the **hItem** field of the **MENUITEM** data structure typically contains a handle of a display object, such as a bitmap handle for MIS_BITMAP menu items. You can change the **hItem** field for a menu item by sending an MM_QUERYITEM message to the menu to fill in the **MENUITEM** structure, changing the relevant fields, and then sending the structure back to the menu with an MM_SETITEM message.

17.5.10 Adding and Deleting Menu Items

An application can add and delete items from its menus dynamically by sending MM_INSERTITEM and MM_DELETEITEM messages to the menu window. Any item, including those in submenus, can be deleted by sending a message to the menu-bar window. Messages to insert items in submenus must be sent to the submenu's pull-down menu window (rather than to the menu-bar window). The handle of the pull-down menu window can be obtained by sending an MM_QUERYITEM message to the menu-bar window and specifying the identifier of the submenu item for the submenu, as shown in the following code fragment:

```
/* IDM_MYMENUID is the ID of the submenu containing the item. */

MENUITEM mi;

hwndActionBar = WinWindowFromID(hwndFrame, FID_MENU);
WinSendMsg(hwndActionBar,                   /* handle of menu bar  */
    MM_QUERYITEM,                           /* message             */
    MPFROM2SHORT(IDM_MYMENUID, TRUE),       /* submenu identifier  */
    (MPARAM) &mi);                          /* pointer to MENUITEM */

hwndpulldown = mi.hwndSubMenu;              /* handle to submenu   */
```

Once the application has the handle of the pull-down menu window, it can insert an item by filling in a **MENUITEM** structure and sending an MM_INSERTITEM message to the pull-down menu. For text-menu items, the application must send a pointer to the text string as well as to the **MENUITEM** structure.

```
mi.iPosition = MIT_END;
mi.afStyle = MIS_TEXT;
mi.afAttribute = 0;
mi.id = IDM_MYMENU_FIRST;
mi.hwndSubMenu = NULL;
mi.hItem = NULL;

WinSendMsg(hwndpulldown, MM_INSERTITEM, (PMENUITEM) &mi,
    (MPARAM) szNewItemString);
```

To delete an item, the application sends an MM_DELETEITEM message to the menu-bar window, specifying the identifier of the item to delete. For example, to clear all the items following IDM_MYMENU_FIRST in a submenu in which the items are numbered sequentially, you would use the following code fragment:

```
/* Clear all the items in MYMENU. */

hwndActionBar = WinWindowFromID(hwndFrame, FID_MENU);
usItemNum = IDM_MYMENU_FIRST;
while (WinSendMsg(hwndActionBar, MM_DELETEITEM,
    MPFROM2SHORT(usItemNum++, TRUE), (MPARAM) 0L));
```

Adding a complete submenu to the menu bar is a more complicated procedure than shown in the previous examples. There are two strategies. The recommended technique is to define all possible submenus in your resource-definition file and then selectively remove and insert the submenus as needed as your program runs.

For example, assume that your program has a submenu that you want to be displayed only when a particular application tool is in use. You define the submenu as part of the main menu resource in your resource-definition file, so that the system reads in the resource menu template and creates the submenu window along with the rest of the menu. You can then remove the submenu from the menu bar, saving the title of the submenu and the **MENUITEM** data structure that defines the submenu, as shown in the following code fragment:

```
HWND hwndActionBar;
MENUITEM mi;
CHAR szMenuTitle[MAX_STRINGSIZE];

/* Remove a submenu so that you can replace it later. */

/* Obtain the handle of a menu menu. */

hwndActionBar = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT, FALSE),
    FID_MENU);

/* Obtain information on the item to remove. */

WinSendMsg(hwndActionBar, MM_QUERYITEM,
    MPFROM2SHORT(IDM_MENUID, TRUE), /* TRUE to search submenus */
    &mi);

/* Save the text for the submenu item. */

WinSendMsg(hwndActionBar, MM_QUERYITEMTEXT,
    MPFROM2SHORT(IDM_FONT, MAX_STRINGSIZE),
    szMenuTitle);

/* Remove the item, but retain mi and szMenuTitle. */

WinSendMsg(hwndMenu, MM_REMOVEITEM,
    MPFROM2SHORT(IDM_FONT, TRUE), NULL);
```

It is important to use the MM_REMOVEITEM message to remove the item, rather than the MM_DELETEITEM message; deleting the item destroys the submenu window but removing it does not destroy the submenu. The submenu should remain intact so that you can insert it later.

To reinsert the submenu, send an MM_INSERTITEM message to the menu bar, passing the **MENUITEM** structure and menu title that you saved when you removed the item. The following code fragment shows how to insert a submenu that was removed by using the previous code example:

```
/* Put the submenu back in and obtain the handle of the menu bar. */

hwndMenu = WinWindowFromID(WinQueryWindow(hwnd, QW_PARENT, FALSE),
                           FID_MENU);

/* Use the information that you saved when you removed the menu. */

WinSendMsg(hwndMenu, MM_INSERTITEM, &mi, szMenuTitle);
```

The other technique that you can use to insert a submenu in the menu bar is to build up a data structure as a menu template in memory and use that template and the **WinCreateWindow** function to create a submenu window. The resulting submenu window handle is then placed in the **hwndSubMenu** field of a **MENUITEM** structure and the menu item is sent to the menu bar with an MM_INSERTITEM message.

You also can create an empty submenu window by using the **WinCreateWindow** function. Pass NULL for the *pCtlData* and *pPresParams* parameters, instead of building the menu template in memory. Then insert a new menu item into the menu bar by using the MM_INSERTITEM message, setting the style MIS_SUBMENU, and putting the window handle of the created menu into the *hwndSubMenu* parameter. Then use the MM_INSERTITEM message to insert the items into the new pull-down menu.

17.5.11 Owner-Drawn Menu Items

Applications can customize the appearance of an individual menu item by setting the MIS_OWNERDRAW style bit for the item. MS OS/2 sends two different messages to an application that includes owner-drawn menu items: WM_MEASUREITEM and WM_DRAWITEM. Both messages include a pointer to an **OWNERITEM** data structure, as shown in the following code fragment:

```
typedef struct _OWNERITEM {
    HWND    hwnd;           /* handle of menu window              */
    HPS     hps;            /* presentation space in which to draw */
    USHORT  fsState;        /* requested style                    */
    USHORT  fsAttribute;    /* requested attribute                */
    USHORT  fsStateOld;     /* current style                      */
    USHORT  fsAttributeOld; /* current attribute                  */
    RECTL   rclItem;        /* bounding rectangle of item         */
    SHORT   idItem;         /* item identifier                    */
    ULONG   hItem;          /* handle of item-display object      */
} OWNERITEM;
```

The WM_MEASUREITEM message is sent only once for each owner-drawn item, when the menu is initialized. The message is sent to the menu's owner window (typically, a frame window), which forwards the message to its client window. Typically, the client window procedure processes the message WM_MEASUREITEM by filling in the **yTop** and **xRight** fields of the **RECTL** structure specified by the **rclItem** field of this **OWNERITEM** structure; this specifies the size of the rectangle needed to enclose the item when it is drawn. The code fragment shown below responds to a WM_MEASUREITEM message. If your owner-drawn menu contains text, you should compute the size of the items from the height of the system font or some other system characteristic.

```
case WM_MEASUREITEM:
    (POWNERITEM) mp2->rclItem.xRight = 26;
    (POWNERITEM) mp2->rclItem.yTop = 10;
    return ((MRESULT) 0);
```

If a menu item has the MIS_OWNERDRAW style, the owner window receives a WM_DRAWITEM message every time the menu item needs to be drawn. You should process this message by using the **hps** and **rclItem** fields of the **OWNERITEM** structure to draw the item. There are two situations in which the owner window receives a WM_DRAWITEM message:

- The item needs to be completely redrawn.
- The item needs to highlighted or unhighlighted.

You can choose to handle one or both of these situations. Typically, you handle the drawing of the item. The system default behavior of inverting the item rectangle to show that the item is selected is often acceptable, however, so you may not want to do this yourself.

The two situations in which a WM_DRAWITEM message is received are detected by comparing the values of the **fsAttribute** and **fsAttributeOld** fields of the **OWNERITEM** structure that is sent as part of the message. If the two fields are the same, you should draw the item. When drawing the item, you can also check the attributes of the item to see if the item has the MIA_CHECKED, MIA_FRAMED, or MIA_DISABLED attribute. You should draw the item according to the settings of the attribute bits.

For example, when the checked attribute of a owner-drawn menu item changes, the system sends a WM_DRAWITEM message to the item so that it can redraw itself and either draw or remove the check mark. If you want the system default check mark, simply draw the item and leave the **fsAttribute** and **fsAttributeOld** fields unchanged; the system will draw the check mark, if necessary. If you draw the check mark yourself, clear the MIA_CHECKED bit in both **fsAttribute** and **fsAttributeOld**, so that the system does not attempt to draw a check mark.

If the **fsAttribute** and **fsAttributeOld** fields of the **OWNERITEM** structure in a WM_DRAWITEM message are not equal, the highlight showing that an item is selected needs to change. The MIA_HILITED bit of the **fsAttribute** field is set if the item needs to be highlighted and is not set if the item needs to be unhighlighted. If you do not wish to provide your own, you should ignore any WM_DRAWITEM message in which **fsAttribute** and **fsAttributeOld** are not equal. If you do not alter these two fields, the system performs its default highlighting operation, which is to invert the bits within the item rectangle. If, however, you wish to provide your own visual cue that an item is selected, you should respond to a WM_DRAWITEM message in which **fsAttribute** and **fsAttributeOld** are not equal by providing the cue and then clearing the MIA_HILITED bit of both **fsAttribute** and **fsAttributeOld** before returning from the message.

Likewise, the MIA_CHECKED and MIA_FRAMED bits of the **fsAttribute** and **fsAttributeOld** fields can either be used to perform the corresponding action or can be passed on unchanged so that the system performs the action.

The following code fragment shows how to respond to a WM_DRAWITEM message when you want to draw the item and also be responsible for its highlighted/unhighlighted state:

```
case WM_DRAWITEM:
    POWNERITEM poi;
    RECTL      rcl;

    poi = (POWNERITEM) mp2;

    /*
     * If the new attribute equals the old attribute,
     * redraw the entire item.
     */

    if (poi->fsAttribute == poi->fsAttributeOld) {

        /*
         * Draw the item in poi->hps and poi->rclItem,
         * and check the attributes for check marks. If you
         * produce your own check marks, use this line of code:
         * poi->fsAttributeOld = (poi->fsAttribute &= ~MIA_CHECKED;
         */

    /* Else the item should be highlighted or unhighlighted. */

    } else if ((poi->fsAttribute & MIA_HILITED) !=
              (poi->fsAttributeOld & MIA_HILITED)) {

        /*
         * Set bits the same so that the menu system does not make
         * the item highlighted or unhighlighted.
         */

        poi->fsAttributeOld = (poi->fsAttribute &= ~MIA_HILITED);
    }
    return TRUE; /* TRUE means item is drawn */
```

17.6 Summary

This section describes the styles, attributes, functions, and messages associated with menus.

17.6.1 Menu-Item Styles

Menu item styles determine what kind of data a menu contains (text, bitmap, etc.), how the menu is displayed (whether or not it is drawn by the owner), and what kind of message it generates when chosen (WM_COMMAND, WM_SYSCOMMAND, or WM_HELP). Menu-item styles are set when the menu item is created and are not typically changed at run time. Menu item attributes, described in the next section, are used for the aspects of a menu item that change frequently while a program is running.

MIS_BITMAP The menu-display object is a bitmap.

MIS_BREAK The item begins a new row or column.

MIS_BREAKSEPARATOR Same as MIS_BREAK, except that it draws a separator between rows or columns.

MIS_BUTTONSEPARATOR The item cannot be selected by using the cursor keys, but it can be selected by using the mouse or the appropriate accelerator key. A menu bar can have zero, one, or two button-separator items. They are always placed at the right side of the menu bar or at the bottom of a pull-down menu.

MIS_HELP A command item with this style notifies its owner window that it has been chosen by using a WM_HELP message rather than a WM_COMMAND message.

MIS_OWNERDRAW The item is drawn by the owner window. The menu sends WM_DRAWITEM and WM_MEASUREITEM messages to the owner window to draw the item and specify its size.

MIS_SEPARATOR This item is a horizontal dividing line in a pull-down menu. It cannot be checked, disabled, or selected.

MIS_STATIC The item is for information only. It cannot be selected by using the mouse or keyboard.

MIS_SUBMENU The item is a submenu item. When the user selects a submenu item, a pull-down menu window is displayed from which the user can choose a command item.

MIS_SYSCOMMAND A command item with this style notifies its owner window that it has been chosen by using a WM_SYSCOMMAND message, rather than a WM_COMMAND message.

MIS_TEXT The menu-display object is a text string. This is the default menu-item style.

The following menu-item styles are mutually exclusive; they may not be specified in combination with each other:

MIS_BITMAP
MIS_OWNERITEM
MIS_SEPARATOR
MIS_TEXT

Likewise, the following menu-item styles are mutually exclusive:

MIS_HELP
MIS_SYSCOMMAND

17.6.2 Menu-Item Attributes

Menu-item attributes specify changing display aspects of a menu item, such as its highlighted, and checked state. Attributes are set when the item is created and typically can change frequently as the program executes and the user interacts with the menus.

MIA_CHECKED Set to produce a check mark to the left of the item.

MIA_ENABLED Set when the item can be selected by the user. If not set, the item is drawn grayed and cannot be selected by the user. An application should disable a menu item when choosing it would be inappropriate—for example, a Save menu item should be disabled when there have been no changes since the last save operation.

MIA_FRAMED Set when a submenu item in the menu bar is framed by vertical lines to the left and right when its pull-down menu is displayed. This is typically handled by the system; an application does not usually have to set this attribute.

MIA_HILITED Set only when the item is currently selected. The application rarely sets this attribute directly, relying instead on the default processing of user input to set the highlighted state of an item.

17.6.3 Menu Functions

Most applications will not use the menu functions listed below, relying instead on the automatic association of menus and frame windows provided by menu resources and the **WinCreateStdWindow** function. These menu functions are useful if you want to use menus in a nonstandard way.

WinCreateMenu Creates a menu window from a menu-template data structure, assigning ownership to the specified window. This function is like the **WinLoadMenu** function, except that the menu data is stored as a menu template in memory, rather than in a resource-definition file.

WinLoadMenu Loads a menu resource from the specified resource-definition file (NULL for the current application's resource file) and assigns ownership to the specified window. The menu is owned by the specified window and is displayed when the owner window receives a WM_UPDATEFRAME message.

17.6.4 Messages Sent from a Menu to an Owner Window

These messages are sent from a menu to an owner. If the owner window is a standard frame window, the messages are passed to the client window's window procedure. All applications that use menus must respond to WM_COMMAND messages. Other messages in this section are appropriate for applications that use the more advanced features of menus.

WM_COMMAND Notifies the owner window when the user chooses a menu item. Applications must respond to this message to use menus.

WM_SYSCOMMAND Notifies the owner window when the user chooses a System menu item; this is equal to the WM_COMMAND message except that the menu item has the MIS_SYSCOMMAND style. The frame window usually does not pass this message to the client window. To process a WM_SYSCOMMAND message, the application must subclass the frame window.

WM_HELP Notifies the owner window when the user chooses a Help menu item; equal to the WM_COMMAND message except that the menu item has the MIS_HELP style. This message is usually generated by the "F1=Help" command item in the menu bar. Applications should respond to this message with a help dialog-box or by using the help-hook facility.

WM_INITMENU Notifies the owner window that the menu or submenu is about to be displayed. This message allows an application to change the state of a menu before the menu is displayed.

WM_MENUSELECT Notifies the owner window each time a menu item is selected. Applications do not need to handle this message to obtain the default menu behavior. For example, an application receives multiple WM_MENUSELECT messages when a user moves the mouse pointer up and down in a menu while the mouse button is pressed. This message allows an application to perform some other action coincident with the selection of a menu item, such as displaying a context-appropriate message in another part of the window. This message is also sent when the user actually chooses a menu item. If the application returns FALSE in response to this message when the user chooses a menu item, the command associated with the menu item is not posted, and the menu is not dismissed.

WM_MENUEND Notifies the owner window when exiting from menu mode.

WM_DRAWITEM Notifies the owner window when an item with the style MIA_OWNERDRAW needs to be drawn. Applications with owner-drawn menu items must respond to this message. The message contains a pointer to a data structure containing a presentation space handle and a rectangle in which to draw the item.

WM_MEASUREITEM Allows the owner window to specify the dimensions of an owner-drawn menu item. Applications with owner-drawn menu items must respond to this message.

WM_QUERYFOCUSCHAIN Temporarily sets the focus to the menu bar while in menu mode. This message is routed to the window from which the menu took the focus.

WM_FOCUSCHANGE Exits from menu mode, if the menu is losing the focus. If the exit operation fails, this message sets the state and is passed to the window that had the focus before menu mode was started.

WM_SETFOCUS Posts an MM_STARTMENUMODE message to initiate menu processing, if receiving the focus. If losing the focus, call the **WinDefWindowProc** function.

WM_QUERYDLGCODE Returns DLGC_MENU or DLGC_STATIC to indicate that this is a menu control and that the menu should not receive the focus when the user presses the DIRECTION keys or the TAB key.

WM_PAINT Draws the menu.

WM_CREATE Creates a list of items from the menu-template structure.

WM_DESTROY Destroys the menu and all its submenus and any display objects associated with the menu items.

WM_ENABLE Invalidates the window rectangle, causing it to be redrawn.

WM_ADJUSTWINDOWPOS Reformats the contents of the menu window.

WM_CONTROLHEAP Notifies the owner of a menu that a control needs the handle of a heap from which memory will be allocated.

WM_CONTROLPOINTER Notifies the owner of a menu that the mouse pointer is over the window.

WM_BUTTON1DOWN Begins processing a user's menu choice.

WM_MOUSEMOVE Sets the default mouse pointer (arrow cursor).

WM_BUTTON2DOWN Activates the menu window.

WM_BUTTON3DOWN Activates the menu window.

WM_QUERYCONVERTPOS Determines whether or not to begin double-byte character set (Kanji) conversion. Menus in MS OS/2 version 1.1 return QCP_NOCONVERT, which indicates that the menu code does not set up the rectangle pointed to by the *mp1* parameter with the cursor bounding rectangle and that conversion should not be performed. Edit controls return QCP_CONVERT and fill in a **RECTL** structure with the cursor boundaries. (Programs can use this rectangle to position a Kanji window next to the cursor.)

17.6.5 Messages Sent to a Menu

The messages in this section are sent to menus, either by the system or by applications. Many of these messages are for manipulating the data that represents the state of menu items. Applications will find these messages useful for dynamically adjusting menus to reflect the current processing environment. Other messages in this section control the display of menu items during menu selection; these typically are sent automatically by the system, although an application can send them to control its menus more directly and override the default behavior.

MM_QUERYITEMCOUNT Returns the number of items in the menu. For the menu bar, this is the number of items in the menu bar. For a submenu, this is the number of items in the submenu.

MM_STARTMENUMODE Starts menu-selection processing, including mouse tracking for menu-item selection.

MM_ENDMENUMODE Exits from menu mode and hides any active submenus.

MM_INSERTITEM Insert the specified menu item in the menu.

MM_DELETEITEM Removes the specified item from the menu and destroys any resources and data structures for that item (such as display objects or submenus).

MM_REMOVEITEM Same as the MM_DELETEITEM message, except that it does not destroy associated submenus or display objects.

MM_SELECTITEM Selects the specified item; and if the *fDismiss* flag is set, posts a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message.

MM_QUERYSELITEMID Returns the identifier of the currently selected item.

MM_QUERYITEM Copies information about a specified item into a caller-supplied **MENUITEM** data structure.

MM_QUERYITEMTEXT Copies the text for a specified menu item into a caller-supplied buffer.

MM_QUERYITEMTEXTLENGTH Returns the length of the text, not including the NULL terminator, for a specified item.

MM_SETITEMHANDLE Sets the menu-item handle for a nontext item and forces the item to be redrawn.

MM_SETITEMTEXT Sets the text for a menu item and forces the item to be redrawn.

MM_ISITEMVALID Returns TRUE if item can be selected.

MM_SETITEM Sets the state of an item, based on the data in a **MENUITEM** structure, and forces the item to be redrawn.

MM_ITEMPOSITIONFROMID Returns the position of a menu item in a menu window, searching submenus if requested.

MM_ITEMIDFROMPOSITION Returns the identifier of the menu item at the specified position in the menu window.

MM_QUERYITEMATTR Returns the current attributes of a menu item.

MM_SETITEMATTR Sets the specified attributes of a menu item.

Chapter
18

Accelerator Tables

18.1 Introduction 239
18.2 About Accelerator Tables 239
18.3 Accelerator Tables in a Resource-Definition File 239
18.4 Accelerator-Table Data Structures 240
18.5 Using an Accelerator Table in an Application 241
18.5.1 Including an Accelerator Table in a Frame Window ... 241
18.5.2 Modifying an Accelerator Table 242
18.6 Summary 242
18.6.1 Functions 242
18.6.2 Accelerator-Item Styles 243
18.6.3 Messages 243

18.1 Introduction

This chapter describes how to use accelerator tables in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Menus
- Creating standard frame windows
- Window messages and the message queue

18.2 About Accelerator Tables

Accelerators are keystrokes that generate command messages for an application; they elicit the same behavior as choosing a menu item. Menus provide an easy way to learn an application's command set, but accelerators provide quicker access to those commands.

Accelerators filter keyboard input. Accelerator keystrokes are translated into command messages before they reach the application. When an accelerator is used, the application receives a command message rather than a keyboard message.

Accelerators function differently from the usual keyboard-to-menu interface. By default, a user can use the ALT key to access submenus and the arrow keys to move along the menu bar. Accelerators provide single keystrokes that generate command messages without the visual effects of pulling down menus or stepping from one item to another.

Like menu items, accelerators can generate WM_COMMAND, WM_HELP, and WM_SYSCOMMAND messages, depending on the setting of the accelerator's style bits. Although accelerators are normally used to generate commands that already exist as menu items, they can also send commands that have no equivalent menu items.

An accelerator table contains an array of accelerators. Accelerator tables exist at two levels within MS OS/2. MS OS/2 maintains a single accelerator table for the system queue and individual accelerator tables for application windows. Accelerators in the system queue apply to all applications—for example, the F1 key always generates a WM_HELP message. Having accelerators for individual application windows ensures that an application can define its own accelerators without interfering with other applications. An accelerator for an application window overrides the accelerator in the system queue. An application can modify both its own accelerator table and the system accelerator table.

18.3 Accelerator Tables in a Resource-Definition File

An application that uses accelerators typically creates an accelerator table resource-definition file containing its accelerators and associates that resource with a standard frame window when the window is created.

The resource-definition file of an accelerator table is a list of accelerator items. Each item defines the keystroke that triggers the accelerator, the command that the accelerator generates, and the accelerator's style. The style bits specify whether the keystroke is a virtual key, a character, or a scan code, and whether the message that is generated is WM_COMMAND, WM_SYSCOMMAND, or WM_HELP. (WM_COMMAND is the default message.)

A resource-definition file for an accelerator table is shown in the following code fragment:

```
ACCELTABLE      ID_ACCEL_RESOURCE
BEGIN
    VK_ESC,     IDM_ED_UNDO,   VIRTUALKEY,  SHIFT
    VK_DELETE,  IDM_ED_CUT,    VIRTUALKEY
    VK_F2,      IDM_ED_COPY,   VIRTUALKEY
    VK_INSERT,  IDM_ED_PASTE,  VIRTUALKEY
END
```

This accelerator table has four accelerator items. The first one is triggered when the user presses SHIFT+ESC; it sends a WM_COMMAND message (the default) just as if the IDM_ED_UNDO menu item had been chosen.

The accelerator table resource-definition file has a resource-identification number that is usually the same as the identifier of the application's menu resource; this allows the accelerator table to be associated with a standard frame window when the frame window is created. You can also define accelerator-table resources with other identification numbers and associate them with windows after the windows are created.

18.4 Accelerator-Table Data Structures

Applications that manipulate accelerator tables can refer to them with a 32-bit handle (**HACCEL**). Using this handle allows an application to make most API function calls for accelerators without needing to account for the internal structures that define the accelerator table. To use accelerator tables in the default manner, it is sufficient to define the table in the resource-definition file and associate it with a standard frame window when creating the window. When an application needs to dynamically create or change an accelerator table, it must use the **ACCEL** and **ACCELTABLE** data structures.

An accelerator table is made up of individual accelerator items. Each item is represented by an **ACCEL** data structure that defines the accelerator's style, keystroke, and command identifier. The **ACCEL** structure has the following form:

```
typedef struct _ACCEL {
    USHORT fs;
    USHORT key;
    USHORT cmd;
} ACCEL;
```

Typically, an application defines the aspects of the accelerator in the resource-definition file for the accelerator, but the data structure can be built in memory at run time, if necessary.

An accelerator table is made up of one or more accelerator items and information that specifies the number of accelerator items in the table and the code page used for the keystrokes in the accelerator items. The **ACCELTABLE** structure has the following form:

```
typedef struct _ACCELTABLE {
    SHORT  cAccel;
    USHORT codepage;
    ACCEL  aaccel[1];
} ACCELTABLE;
```

Notice that the array of accelerator items is defined as having only one member. Applications that use accelerator-table data structures directly must allocate sufficient memory to hold all the items in the table.

18.5 Using an Accelerator Table in an Application

An application can automatically load an accelerator table resource-definition file when creating a standard frame window, or it can load the resource independently and associate it with a window or with the entire system.

An application can set and query the accelerator tables for a specific window or for the entire system. For example, an application could query the system accelerator table, copy it, modify the copied accelerator-table data structures, and then use the modified copy to set the system accelerator table. An application that does this should maintain the original accelerator table and restore it when the application terminates. An application can also modify its window's accelerator table at run time to respond more appropriately to the current environment.

18.5.1 Including an Accelerator Table in a Frame Window

An application can add an accelerator table to a frame window either by using the **WinSetAccelTable** function or by defining an accelerator-table resource and creating a frame window with the FCF_ACCELTABLE frame style. The second method is shown in the following code fragment:

```
ULONG lControlStyle = FCF_MENU | FCF_SIZEBORDER
                    | FCF_TITLEBAR | FCF_ACCELTABLE;

hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE,
    &lControlStyle, szClassName, szTitle, OL, NULL, ID_MENU_RESOURCE,
    &hwndClient);
```

Note that if you set the *lControlStyle* parameter to FCF_STANDARD you must define an accelerator-table resource, because FCF_STANDARD includes the FCF_ACCELTABLE flag.

If the window being created also has a menu, the menu resource and the accelerator resource must have the same resource identifier; this is because the **WinCreateStdWindow** function has only one input parameter to specify the resource ID of menus, accelerator tables, and icons. If an application creates a resource-definition file for the accelerator table and then opens a standard frame window, as shown in the preceding example, the accelerator table is automatically installed in the window's input queue and keyboard events are translated during normal events-processing. The application responds to WM_COMMAND, WM_SYSCOMMAND, and WM_HELP messages; it does not matter whether they come from a menu or from an accelerator.

An application can also add an accelerator table to a window by calling the **WinSetAccelTable** function with an accelerator-table handle and a frame-window handle. The application can call either the **WinLoadAccelTable** function to retrieve an accelerator table from a resource file or the **WinCreateAccelTable** function to create an accelerator table from an accelerator-table data structure in memory.

18.5.2 Modifying an Accelerator Table

An application can modify an accelerator table, either for its own windows or for the system, by retrieving the handle of the accelerator table, using the handle to copy the accelerator-table data to an application-supplied buffer, changing the data in the buffer, and then using the data in the buffer to create a new accelerator table. The application can then use the new accelerator-table handle to set the accelerator table, either for a window or for the system. This process is outlined in the following list:

1. Call the **WinQueryAccelTable** function to retrieve an accelerator-table handle.
2. Call the **WinCopyAccelTable** function with a null buffer handle to determine how many bytes are in the table.
3. Allocate sufficient memory for the accelerator-table data.
4. Call the **WinCopyAccelTable** function with a pointer to the allocated memory.
5. Modify the data in the buffer (assuming it has the form of an **ACCELTABLE** data structure).
6. Call the **WinCreateAccelTable** function, passing a pointer to the buffer with the modified accelerator-table data.
7. Call the **WinSetAccelTable** function with the handle returned by the **WinCreateAccelTable** function.

18.6 Summary

This section summarizes the functions, styles, and messages related to accelerator tables.

18.6.1 Functions

The following functions allow your application to use accelerator tables:

WinCopyAccelTable Copies the specified accelerator table to an application-provided **ACCELTABLE** data structure.

WinCreateAccelTable Creates an accelerator table using an **ACCELTABLE** data structure. This is similar to the **WinLoadAccelTable** function except that this function does not use resources.

WinDestroyAccelTable Destroys the specified accelerator table.

WinLoadAccelTable Loads a specified accelerator table from a specified dynamic-link module (the module handle is NULL for the current application) and returns a handle of the accelerator table.

WinQueryAccelTable Returns the accelerator-table handle for the specified window, or for the system if the window handle is NULL.

WinSetAccelTable Sets the accelerator table for the specified window, or for the system if the window handle is NULL. **WinSetAccelTable** will remove an existing accelerator table if the accelerator-table handle is NULL.

WinTranslateAccel Translates a WM_CHAR message into a WM_COMMAND, WM_SYSCOMMAND, or WM_HELP message by using the specified accelerator table. This function is normally called automatically by the **WinGetMsg** or **WinPeekMsg** function when a WM_CHAR message is received.

18.6.2 Accelerator-Item Styles

The following accelerator-item styles are specified in the **fs** field of the **ACCEL** structure:

AF_ALT Means the ALT key must be held down when the key is pressed.

AF_CHAR Means the keystroke is a translated character, using the code page for the accelerator table. This is the default style.

AF_CONTROL Means the CTRL key must be held down when the key is pressed.

AF_HELP Means the accelerator generates a WM_HELP message instead of a WM_COMMAND message.

AF_LONEKEY Means no other key is pressed while the key is down. This style is typically used with the ALT key to specify that simply pressing and releasing the ALT key triggers the accelerator.

AF_SCANCODE Means the keystroke is an untranslated scan code from the keyboard.

AF_SHIFT Means the SHIFT key must be held down when the key is pressed.

AF_SYSCOMMAND Means a WM_SYSCOMMAND message is generated by the accelerator, instead of a WM_COMMAND message.

AF_VIRTUALKEY Means the keystroke is a virtual key—for example, the F1 function key.

18.6.3 Messages

The following messages are used in the management of accelerator tables:

WM_QUERYACCELTABLE Sent to a frame window by the **WinQueryAccelTable** function.

WM_SETACCELTABLE Sent to a frame window by the **WinSetAccelTable** function.

WM_TRANSLATEACCEL Sent to a frame window by the **WinTranslateAccel** function.

Chapter

19

Dialog Windows

19.1 Introduction 247

19.2 About Dialog Windows 247
19.2.1 Modal and Modeless Dialog Windows 247
19.2.2 Dialog Items 247
19.2.3 Dialog-Control Groups 248
19.2.4 Message Boxes 248

19.3 Dialog Data Structures 249
19.3.1 Dialog Coordinates 250

19.4 Dialog Resources 250

19.5 Using Message and Dialog Boxes 251
19.5.1 Message Boxes 251
19.5.1.1 System-Modal Message Boxes 252
19.5.2 Dialog Boxes 253
19.5.2.1 Modal Dialog Boxes 253
19.5.2.2 Modeless Dialog Boxes 254
19.5.2.3 Initializing a Dialog Box 255
19.5.2.4 Menus in Dialog Boxes 255
19.5.2.5 Dialog Procedure 256
19.5.2.6 Manipulating Dialog Items 257

19.6 Summary 258
19.6.1 Dialog-Window Styles 258
19.6.2 Message-Box Styles 258
19.6.3 Message-Box Return Values 259
19.6.4 Functions 260
19.6.5 Messages Sent to Dialog Boxes and Dialog Items 261

19.1 Introduction

This chapter describes creating and using dialog windows and message boxes in your applications. You should also be familiar with the following topics:

- Standard user interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Control windows
- Window messages and message queues

19.2 About Dialog Windows

A dialog window (also called a dialog box or a dialog) is a window that contains one or more child control windows and is typically used to display messages to and gather input from the user. It is often a temporary window that an application creates to gather specific input, destroying the window immediately after use.

Dialog windows provide a high-level method for applications to display and gather information. MS OS/2 contains many functions and messages that help manage the control windows that make up a dialog window, thus easing the burden of maintaining complex input and output systems.

19.2.1 Modal and Modeless Dialog Windows

Dialog windows can be modal or modeless. A modal dialog window requires that the dialog window be dismissed before the user can activate other windows in the same application. Generally, an application uses a modal dialog window to get essential information from the user before proceeding with an operation. A modeless dialog window allows the user to activate other windows without dismissing the dialog window.

Both modal and modeless dialog windows allow the user to activate windows in another application before responding to the dialog window. For more information, see Section 19.2.4.

Modal dialog windows are simpler for an application to manage because they are created, perform their task, and close, all in a single function call.

Modeless dialog windows require more attention from the application because they exist until explicitly dismissed. Modeless dialog windows provide a more flexible interface, however, by allowing the user to move to other windows in the application before responding to the dialog window.

19.2.2 Dialog Items

A dialog item is a child window of the dialog window. The dialog window is usually a window of class WC_FRAME. MS OS/2 provides many predefined window classes, called control windows, that are used as dialog items. Predefined control windows include static display boxes, text-entry fields, buttons, and list boxes. Customized window classes can also be used as dialog items.

Because dialog items are windows, they can be manipulated by all window-management functions relating to size, position, and visibility. Dialog items are always owned by the dialog frame window. Most predefined control-window classes send notification messages to their owners when the user interacts with their control windows. The dialog frame window receives these notification messages and passes them on to the application through the application-defined dialog procedure.

19.2.3 Dialog-Control Groups

Items within a dialog window can be organized into groups. When items are arranged in a group, the user can move from one item to another in the same group by using the direction keys. When the user presses a direction key, the focus moves from one item in a group to the next item of the same group, but not to items of other groups within the dialog window.

Arranging items in groups is useful for radio buttons. Although other control types can also be displayed this way, entry-field controls cannot; they process direction keys themselves.

The first item in a dialog-control group has the WS_GROUP window style. All subsequent items in the dialog template are considered part of that group until another item is given the WS_GROUP style, which begins a new group.

The WS_TABSTOP style is often used along with the WS_GROUP style. This style marks the items that can receive the focus when the user presses the TAB key. Each time the user presses the TAB key, the focus moves to the next item that has the WS_TABSTOP style. Generally, the WS_GROUP and WS_TABSTOP styles are defined together for the first item of each group in the dialog template. This makes it possible for a user to press the TAB key to move *between* groups of items and to use the direction keys to move between items *within* a group.

The WS_TABSTOP style should not be used for radio buttons because the system automatically maintains a tabstop on any selected item in a radio-button group; the focus will always be on the currently selected item when pressing the TAB key in a group of radio buttons.

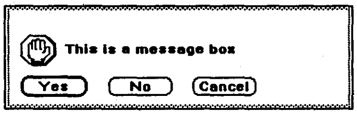
The WS_GROUP and WS_TABSTOP styles are also useful for preventing the user from moving to a particular button when using the keyboard. For example, if the dialog window has an OK and a Cancel button, you should put them in the same group, with the OK button as the first item in the group. The user can press the TAB key to select the OK button, but not the Cancel button. To move to the Cancel button by using the keyboard, the user must first press the TAB key to move to the OK button, and then press a direction key to move the focus to the Cancel button. For more information on how to define groups and tabstops in dialog windows, see Section 19.4.

19.2.4 Message Boxes

Message boxes are dialog windows predefined by the system and used as a simple interface for applications without creating dialog-template resources or dialog procedures. An application can call the **WinMessageBox** function and specify the type of message box and message text. The system displays the message and waits for the user to dismiss the message box by selecting a button in the message box. The system then returns a result code to the application, indicating which button the user selected.

Message boxes are best for short notification messages that require a simple acknowledgment or choice by the user. Applications do not specify a dialog procedure for message boxes, so they cannot readily change the action of a message box. There are many predefined message-box styles. Figure 19.1 shows a sample message box.

Figure 19.1
Sample Message Box



This is a message box

Message boxes can be application-modal or system-modal. Application-modal means that the user cannot activate another window in the current application before responding to the message box, but can switch to another application before responding. System-modal means that the user cannot activate another window in any application while the message box is present. A system-modal message box should be used only to display urgent error messages (running out of memory, for example).

19.3 Dialog Data Structures

A dialog-window item is a control window that is owned by the dialog window. Each dialog-window item is described by a **DLGTITEM** data structure. The **DLGTITEM** structure is rarely accessed directly by an application. Most manipulation of dialog items is handled by system functions. Applications that create dialog items that are not defined as part of a dialog-template resource must create dialog-window-item structures in memory. The format of a **DLGTITEM** structure is as follows:

```
typedef struct _DLGTITEM {
    USHORT    fsItemStatus;
    USHORT    cChildren;
    USHORT    cchClassName;
    USHORT    offClassName;
    USHORT    cchText;
    USHORT    offText;
    ULONG     flStyle;
    SHORT     x;
    SHORT     y;
    SHORT     cx;
    SHORT     cy;
    USHORT    id;
    USHORT    offPresParams;
    USHORT    offCtlData;
} DLGTITEM;
```

Because a dialog window can have many items, a **DLGTEMPLATE** data structure consists of header information followed by an array of dialog-window items. Applications that create dialog windows without using dialog resources must create a dialog template in memory and then call the **WinCreateDlg** function. The format of a **DLGTEMPLATE** structure is as follows:

```
typedef struct _DLGTEMPLATE {
    USHORT      cbTemplate;
    USHORT      type;
    USHORT      codepage;
    USHORT      offadlgti;
    USHORT      fsTemplateStatus;
    USHORT      iItemFocus;
    USHORT      coffPresParams;
    DLGTITEM    adlgti[1];
} DLGTEMPLATE;
```

19.3.1 Dialog Coordinates

Coordinates in a dialog template are specified in dialog coordinates and are based on the size of the system font. A horizontal unit is one-fourth of the system-font-character average width; a vertical unit is one-eighth of the system-font-character average height. The origin of the dialog template is the lower-left corner of the dialog window. MS OS/2 provides the **WinMapDlgPoints** function for converting dialog coordinates into window coordinates.

19.4 Dialog Resources

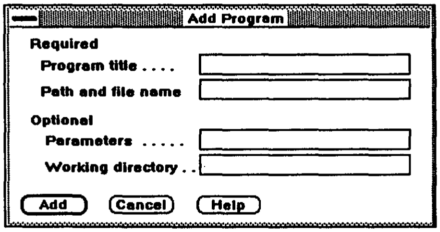
Most applications define dialog templates in resource files rather than constructing template data structures in memory at run time. The dialog-resource file defines the size and style of the dialog-window frame and specifies each control item.

The following source-code fragment creates a dialog template. Notice that the WS_GROUP and WS_TABSTOP style designations are given for the first item in each group. The dimensions and position for each item are given in dialog coordinates rather than in window coordinates.

```
DLGTEMPLATE IDD_ABOUT
BEGIN
  DIALOG "", IDD_ABOUT2, 10, 10, 150, 110, FS_DLGBORDER, 0
  BEGIN
    CONTROL "Attributes:",100,
      10, 30, 100, 70,
      WC_STATIC,
      SS_GROUPBOX | WS_VISIBLE
    CONTROL "Highlighted",101,
      20, 80, 58, 12,
      WC_BUTTON,
      WS_GROUP | WS_TABSTOP | BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Enabled",102,
      20, 60, 58, 12,
      WC_BUTTON,
      BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Checked",103,
      20, 40, 58, 12,
      WC_BUTTON,
      BS_AUTOCHECKBOX | WS_VISIBLE
    CONTROL "Okay", DID_OK,
      10, 10, 50, 14,
      WC_BUTTON,
      WS_GROUP | WS_TABSTOP | BS_PUSHBUTTON | BS_DEFAULT | WS_VISIBLE
    CONTROL "Cancel", DID_CANCEL,
      80, 10, 50, 14,
      WC_BUTTON,
      BS_PUSHBUTTON | WS_VISIBLE
  END
END
```

Figure 19.2 shows the dialog box created by the previous dialog-template resource definition:

Figure 19.2
Sample Dialog Box



19.5 Using Message and Dialog Boxes

The simplest dialog window is the message box. Most message boxes present simple messages and offer the user one, two, or three responses (represented by buttons). A message box is easy to use and is appropriate when an application requires a clearly defined response to a static message. However, message boxes lack flexibility in size and placement on the screen, and they are limited in the choices they offer the user. Applications that require more control over size, position, and content should use regular dialog boxes instead of message boxes.

19.5.1 Message Boxes

Message boxes provide an easy way for applications to display simple messages without creating dialog templates or writing dialog procedures. Message boxes are intended mainly for conveying information to users, although they do have limited input capabilities.

There are several different kinds of predefined message boxes. There are three parts to a message box: the icon, the message, and buttons. Applications specify the icons and buttons using message-box style constants. Message text is specified by a null-terminated string.

To create a message box, the application calls the **WinMessageBox** function, which displays the message box and processes user input until the user selects a button in the message box. The return value of the **WinMessageBox** function indicates which button was selected.

The following code fragment illustrates how to create a message box with a default Yes button, a No button, and a question-mark (?) icon. This example assumes that you have defined a string resource with the identifier MY_MESSAGESTR_ID in the resource file.

```
CHAR szMessageString[255];
USHORT cch;
USHORT usResult;

cch = WinLoadString(hab,
    (HMODULE) NULL,
    MY_MESSAGESTR_ID,
    sizeof(szMessageString),
    szMessageString);

usResult = WinMessageBox(hwndFrame,  /* parent    */
    hwndFrame,                       /* owner     */
    szMessageString,                 /* text      */
    (PSZ) "",                        /* caption   */
    MY_MESSAGEWIN,                   /* window ID */
    MB_YESNO |
    MB_ICONQUESTION |
    MB_DEFBUTTON1);                  /* style     */

 if (usResult == MBID_YES)

      /* do yes case */

 else

      /* do no case  */
```

The **WinMessageBox** function returns predefined values indicating which button has been selected. These values are listed in Section 19.6.3.

Note that strings for message boxes should be defined as string resources to facilitate program translation for other countries. However, there is a danger in using string resources in message boxes that are called in low-memory situations. Loading a string resource in these situations could cause severe memory problems and cause an application to fail. One way to solve this problem is to preload the string resource and make it nondiscardable so it will be available when the message box must be displayed.

19.5.1.1 System-Modal Message Boxes

Message boxes are always modal. The default style for a message box is application-modal. With this style, a user cannot select another window in the same application until the message box is dismissed. However, the user can switch to a different application.

It is possible to create a message box that is system-modal. A system-modal message box prevents a user from selecting another window in the current application or switching to a different application until responding to the message box. A system-modal message box is useful when displaying a warning to the user that there may be serious problems with the system, such as insufficient memory.

There are two levels of modality for system-modal message boxes—soft modal and hard modal. A soft-modal message box does not allow keystrokes or mouse input to reach any other window, but does allow other messages, such as deactivation and timer messages, to reach other windows. A hard-modal message box does not allow any messages to reach other windows. A hard-modal message box is appropriate for serious system warnings.

A hard-modal message box is created by combining the MB_ICONHAND style with the MB_SYSTEMMODAL style. A soft-modal message box is created by using the MB_SYSTEMMODAL style with any style other than MB_ICONHAND. The MB_SYSTEMMODAL icon is always in memory and is available even in low-memory situations.

19.5.2 Dialog Boxes

When using dialog boxes, an application must load the dialog box, process user input, and destroy the dialog box when the user finishes the task. The process for handling dialog boxes varies, depending on whether the dialog box is modal or modeless. A modal dialog box requires the user to dismiss the dialog box before activating another window in the application. However, the user can activate windows in different applications. A modeless dialog box does not require an immediate response from the user. It is similar to a frame window containing control windows. The following sections discuss how to use both types of dialog boxes.

19.5.2.1 Modal Dialog Boxes

Modal dialog boxes present users with information and questions in such a way that they must respond before proceeding with other operations in the application.

The easiest way to use a modal dialog box is to define a dialog template in the resource file and then call the **WinDlgBox** function, specifying the dialog-box resource ID and a pointer to the dialog procedure. The **WinDlgBox** function loads the dialog-box resource, displays the dialog box, and handles all user input until the user dismisses the dialog box. The dialog procedure receives messages when the dialog box is created (WM_INITDLG) and other messages when the user interacts with each dialog item, such as entering text in entry fields or selecting buttons.

You must specify both the parent and owner windows when loading a dialog box using the **WinDlgBox** function. Generally, the parent window should be HWND_DESKTOP and the owner should be a client window in your application.

Dialog boxes typically contain buttons that send WM_COMMAND messages when selected by the user. WM_COMMAND messages passed to the **WinDefDlgProc** function result in the **WinDismissDlg** function being called, with the window ID of the source button as the return code. Dialog boxes with OK or Cancel as the only buttons can ignore WM_COMMAND messages, allowing them to be passed to the **WinDefDlgProc** function. The **WinDefDlgProc** function calls the **WinDismissDlg** function to dismiss the dialog box and returns the DID_OK or DID_CANCEL code.

Passing WM_COMMAND messages to the **WinDefDlgProc** function means that all button presses in the dialog box will dismiss the dialog box. If you want particular buttons to initiate operations without closing the dialog box or if you want to perform some processing without closing the dialog box, you should handle the WM_COMMAND messages in the dialog procedure.

If you handle WM_COMMAND messages in the dialog procedure, you must call the **WinDismissDlg** function to dismiss the dialog box. Your dialog procedure passes the DID_OK code to the **WinDismissDlg** function if the user selects the OK button or the DID_CANCEL code if the user selects the Cancel button.

When you call the **WinDismissDlg** function or pass the WM_COMMAND message to the **WinDefDlgProc** function, the dialog box is dismissed and the **WinDlgBox** function returns the value passed to the **WinDismissDlg** function. This return value identifies the button selected.

An alternative to using the **WinDlgBox** function is to call the individual functions that duplicate its functionality, as shown in the following code fragment:

```
dlg = WinLoadDlg(...);
result = WinProcessDlg(dlg);
WinDestroyWindow(dlg);
```

After calling the **WinProcessDlg** function, your dialog procedure must call the **WinDismissDlg** function to dismiss the dialog box. Although the dialog box is dismissed (hidden), it still exists. You must call the **WinDestroyWindow** function to destroy a dialog box if it was loaded using the **WinLoadDlg** function. The **WinDlgBox** function automatically destroys a dialog box before returning.

If you want to manipulate individual items in a dialog box or add a menu after loading the dialog box (but before calling **WinProcessDlg**), it is better to make individual calls rather than calling the **WinDlgBox** function. Individual calls are also useful for querying individual dialog items, such as the contents of an entry-field control after a dialog box is closed but before it is destroyed. Destroying a dialog box also destroys any dialog-item control windows that are child windows of the dialog box.

19.5.2.2 Modeless Dialog Boxes

A modeless dialog box, unlike a modal dialog box, does not require user interaction to activate another window in the current application.

To use a modeless dialog box in an application, you should create a dialog template in the resource file, just as for a modal dialog box. Because modeless dialog boxes share the screen equally with other frame windows, it is a good idea to give modeless dialog boxes a title bar so they can be moved around the screen. The following Resource Compiler source fragment shows a dialog template for a dialog box with a title bar, a System menu, and a Minimize Box.

```
DLGTEMPLATE IDD_SAMP
BEGIN
    DIALOG "Modeless Dialog", IDD_SAMP, 80, 92, 126, 130,
        WS_VISIBLE | FS_DLGBORDER,
        FCF_TITLEBAR | FCF_SYSMENU | FCF_MINIMIZE
    BEGIN

    /* Put control-window definitions here. */

    END
END
```

The application loads the dialog resource from the resource file by using the **WinLoadDlg** function, receiving in return a window handle to the dialog box. The application treats the dialog box as if it were an ordinary window. Messages for the dialog box are dispatched through the event loop the application uses for its other windows. In fact, an application can have a modeless dialog box as its only window.

The resource for a modeless dialog box is just like that used for a modal dialog box. The difference between modal and modeless dialog boxes is in the way applications handle input to each box. For a modal dialog, the **WinDlgBox** and **WinProcessDlg** functions handle all user input to the dialog box, preventing access to other windows in the application. For a modeless dialog box, the application does not call these functions, relying instead on a normal message loop to dispatch messages to the dialog procedure.

The main difference between a modeless dialog box and a standard frame window with child control windows is that for a modeless dialog box, an application can define child windows for the dialog box in a dialog template, automating the creation process of the window and its child windows. The same effect can be achieved by creating a standard frame window, but the child control windows must be created individually.

It is important that an application keep track of all open modeless dialog boxes so that it can destroy all open windows before terminating.

19.5.2.3 Initializing a Dialog Box

Generally, an application defines a dialog template in its resource file and loads the dialog box by calling the **WinLoadDlg** function or the **WinDlgBox** function (which itself calls **WinLoadDlg**). The dialog box is created as an invisible window unless the window style WS_VISIBLE is specified in the dialog template. A WM_INITDLG message is sent to the dialog procedure before the **WinLoadDlg** function returns. As each control defined in the template is created, the dialog procedure may receive various control notifications before the function returns. A dialog window can be destroyed by using the **WinDestroyWindow** function. The **WinLoadDlg** function returns a handle to the dialog window immediately after creating a dialog box.

In general, it is a good idea to define a dialog box as invisible since this allows for optimization. For example, an experienced user may rapidly type ahead, anticipating the processing of a dialog-box command. In such a case, there is no need to display the dialog box because the user has finished the interaction before the window can be displayed. This is how the **WinProcessDlg** function works—it does not display a dialog box while there are still WM_CHAR messages in the input queue. It allows these messages to be processed first.

As control windows in a dialog box are created from the template, strings in the template are processed by the **WinSubstituteStrings** function. Any WM_SUBSTITUTESTRING messages are sent to the dialog procedure before the **WinLoadDlg** function returns.

When child windows of a dialog window are created, the **WinSubstituteStrings** function is used so child windows can make substitutions in their window text. If any child-window text string contains the percent sign (%) substitution character, the length of the text string is limited to 256 characters after it is returned from the substitution.

19.5.2.4 Menus in Dialog Boxes

To create a menu bar and menus in a dialog box, an application should first load the dialog box to get a handle to the dialog-frame window. The dialog-frame window can be associated with a menu resource by calling the **WinLoadMenu** function. This function requires arguments specifying the menu ID and handle of the parent window for the menu. Finally, the dialog-frame window must incorporate the menu by sending a WM_UPDATEFRAME message to the dialog box. The following code fragment illustrates these operations:

```
/* Get the dialog resource. */

hwndDialog = WinLoadDlg(...);

/* Get the menu resource and attach it to the dialog. */

hwndMenu = WinLoadMenu(hwndDialog, ...);
```

```
/* Inform the dialog that it has a new menu. */

WinSendMsg(hwndDialog, WM_UPDATEFRAME, OL, OL);
```

Applications can create menus in modal and modeless dialog boxes. The code fragment above can be used for either type of dialog box. In the case of a modal dialog box, your application should call the **WinProcessDlg** function to handle user input until the dialog box is dismissed. For a modeless dialog box, your application should call the **WinShowWindow** function to display the dialog box, allowing the message loop to direct messages to the dialog box.

19.5.2.5 Dialog Procedure

The main difference between a dialog procedure and a window procedure is that a dialog procedure does not receive WM_CREATE messages. Instead, a dialog procedure receives WM_INITDLG messages, which are sent after a dialog box is created but before it is displayed. The WM_INITDLG message can be used to do the same type of initialization tasks that are handled by WM_CREATE messages.

For example, if a dialog box contains a list box, you should use the message WM_INITDLG to fill the list box with items. This procedure can also be used to enable or disable buttons in a dialog box, depending on your application.

You can also call the **WinSetDlgItemText** or **WinSetDlgItemShort** function during dialog initialization to set up text items that reflect the current conditions in your application.

Another typical task for the WM_INITDLG message handler is centering a dialog on the screen or within its owner window. The following code fragment illustrates how to center a dialog box on screen using the WM_INITDLG message:

```
case WM_INITDLG:
    /* Center the dialog box and get the screen rectangle. */

    WinQueryWindowRect(HWND_DESKTOP, &rclScreenRect);

    /* Get the dialog-box rectangle. */

    WinQueryWindowRect(hwnd, &rclDialogRect);

    /* Get the dialog-box width. */

    sWidth = (SHORT) (rclDialogRect.xRight - rclDialogRect.xLeft);

    /* Get the dialog-box height. */

    sHeight = (SHORT) (rclDialogRect.yTop - rclDialogRect.yBottom);

    /* Set the lower-left corner horizontal coordinate. */

    sBLCx = ((SHORT) rclScreenRect.xRight - sWidth) / 2;

    /* Set the lower-left corner vertical coordinate. */

    sBLCy = ((SHORT) rclScreenRect.yTop - sHeight) / 2;

    /* Move, size, and show the window. */

    WinSetWindowPos(hwnd,
        HWND_TOP,
        sBLCx, sBLCy,
        O, O,          /* ignores size arguments */
        SWP_MOVE);

    return OL;
```

The dialog procedure receives notification messages from each control-window item in a dialog box whenever a user clicks an item or enters text in an entry field. Most dialog procedures wait for the user to select one or more dialog-window buttons to signal that he or she has finished with the dialog box. When the dialog procedure receives one of these messages, it should call the **WinDismissDlg** function, as shown in the following code fragment. The second argument to the **WinDismissDlg** function is the value returned by the **WinDlgBox** or **WinProcessDlg** function. Generally, the ID of the button that was pressed is returned.

```
MRESULT FAR PASCAL SampDialogProc(hwnd, usMessage, mp1, mp2)
HWND hwnd;
USHORT usMessage;
MPARAM mp1;
MPARAM mp2;
{
    switch (usMessage) {
        case WM_COMMAND:
            switch (SHORT1FROMMP(mp1)) {
                case DID_OK:

                    /*
                     * Final dialog-item queries,
                     * dismiss the dialog.
                     */

                    WinDismissDlg(hwnd, DID_OK);
                    return OL;
            }
            break;
    }
    return (WinDefDlgProc(hwnd, usMessage, mp1, mp2));
}
```

Other dialog-box items send notification messages specific to the type of control window. Your dialog procedure should respond to notification messages from each dialog item. Any messages that a dialog procedure does not handle should be passed to the **WinDefDlgProc** function for default processing. The default dialog procedure is exactly the same as the default frame-window procedure.

The WM_COMMAND message from the OK button indicates that the user has selected the OK button and is finished with the dialog box. If the dialog box has other controls, such as entry fields or check boxes, your dialog procedure should query the contents or state of each control when it receives a message from the OK button. Before dismissing a dialog box, your dialog procedure should collect input from each dialog-box control before closing the dialog box.

19.5.2.6 Manipulating Dialog Items

Dialog items are control windows, and as such they can be manipulated using standard window-management function calls. The window handle is obtained for each dialog item by calling the **WinWindowFromID** function and passing the window handle for the dialog box and the window ID for the dialog item as defined in the dialog template. For example, the following Resource Compiler source-code fragment should be included in your dialog template:

```
DLGTEMPLATE IDD_ABOUT
BEGIN
    DIALOG "", IDD_ABOUT, 80, 92, 126, 130, FS_DLGBORDER, 0
    BEGIN
        PUSHBUTTON "My Button", ITEMID_MYBUTTON, 37, 107, 56, 12

        /* Other item definitions ... */

    END
END
```

Based on the above code fragment, your application will receive the button-item handle by initiating the following call to the **WinWindowFromID** function:

```
hwndItem = WinWindowFromID(hwndDialog, ITEMID_MYBUTTON);
```

Applications often change the contents, enabled state, or position of dialog items at run time. For example, in a dialog box that contains a list box of filenames and an Open button, the Open button should be disabled until the user selects a file from the list. To do this, the button should be defined as disabled in the dialog resource so that it is disabled when the dialog box is first displayed. At run time, the dialog procedure receives a notification message from the list box when the user selects a file. At that time, the dialog procedure calls the **WinEnableWindow** function to enable the Open button.

Applications can also change the text in static dialog items and buttons. This is done by calling the **WinSetWindowText** function and using the window handle of particular dialog item.

19.6 Summary

The following sections summarize the styles, functions, and messages associated with dialog windows and message boxes.

19.6.1 Dialog-Window Styles

The following style constants can be used to specify the border and alignment of a dialog box:

FCF_DLGBORDER Draws the dialog window with a double border that identifies it as a dialog box.

FCF_MOUSEALIGN Draws the dialog window using the *x*- and *y*-position relative to the mouse position at the time the dialog window is created. The dialog window position is modified to keep it within the screen boundaries, if possible. The dialog window can be drawn with the OK button under the mouse pointer by using negative *x*- and *y*-position values in the dialog template.

FCF_SCREENALIGN Draws the dialog window using the *x*- and *y*-position relative to the coordinates of the entire screen rather than using the default coordinates of the owner window.

19.6.2 Message-Box Styles

The following style constants can be used to specify the type of message box created by calling the **WinMessageBox** function:

MB_ABORTRETRYIGNORE Creates a message box that has Abort, Retry, and Ignore buttons.

MB_APPLMODAL Creates an application-modal message box. A user cannot select other windows in the current application, but can switch to other applications.

MB_CANCEL Creates a message box that has a Cancel button.

MB_DEFBUTTON1 Defines the first button in a message box as the default button. All message boxes have this style unless MB_DEFBUTTON2 or MB_DEFBUTTON3 is specified.

MB_DEFBUTTON2 Defines the second button as the default button.

MB_DEFBUTTON3 Defines the third button as the default button.

MB_ENTER Creates a message box that has an Enter button.

MB_ENTERCANCEL Creates a message box that has Enter and Cancel buttons.

MB_HELP Creates a message box that has a Help button.

MB_ICONASTERISK Creates a message box that has an asterisk (*) icon.

MB_ICONEXCLAMATION Creates a message box that has a exclamation-point (!) icon.

MB_ICONHAND Creates a message box that has the hand icon. This icon is always in memory and should be used when displaying message boxes in low-memory situations.

MB_ICONQUESTION Creates a message box that has a question-mark (?) icon.

MB_MOVEABLE Creates a message box that a user can move by using the mouse.

MB_NOICON Creates a message box that has no icons.

MB_OK Creates a message box that has an OK button.

MB_OKCANCEL Creates a message box that has OK and Cancel buttons.

MB_RETRYCANCEL Creates a message box that has Retry and Cancel buttons.

MB_SYSTEMMODAL Creates a system-modal message box. A user cannot select any other window in the current application or switch to another application until this message box is dismissed. This style is used in combination with MB_ICONHAND to prevent any messages from being sent to other windows or applications. This is useful in situations where the system is damaged or there are other serious problems.

MB_YESNO Creates a message box that has Yes and No buttons.

MB_YESNOCANCEL Creates a message box that has Yes, No, and Cancel buttons.

19.6.3 Message-Box Return Values

The following are predefined values returned by the **WinMessageBox** function, indicating which button is pressed to dismiss the message box:

MBID_ABORT Abort button dismisses the message box.

MBID_CANCEL Cancel button dismisses the message box.

MBID_ENTER Enter button dismisses the message box.

MBID_ERROR An error has occurred in processing the message box.

MBID_HELP Help button dismisses the message box.

MBID_IGNORE Ignore button dismisses the message box.

MBID_NO No button dismisses the message box.

MBID_OK OK button dismisses the message box.

MBID_RETRY Retry button dismisses the message box.

MBID_YES Yes button dismisses the message box.

19.6.4 Functions

The following functions are used with dialog windows and message boxes:

WinAlarm Creates an audible signal. The type of sound is specified by one of three predefined constants: WA_WARNING, WA_NOTE, and WA_ERROR. The actual sound emitted for these styles varies, depending on the hardware capabilities.

WinCreateDlg Functions similarly to the **WinLoadDlg** function except that the **WinCreateDlg** function dialog template is in memory rather than in a resource file. This function returns a handle to the dialog window.

WinDefDlgProc Creates the default dialog procedure that processes dialog messages (messages that the application dialog procedure does not process). This action is identical to that produced by the **WinDefWindowProc** function for frame windows.

WinDestroyWindow Destroys the dialog window and all its child control windows. This function uses the dialog-window handle.

WinDismissDlg Hides the dialog box and causes the **WinProcessDlg** or **WinDlgBox** function to return a specified result. Applications call this function from a dialog procedure when a user selects a button indicating the interaction with the dialog box is finished.

WinDlgBox Loads and processes a modal dialog box and returns the result generated by the **WinDismissDlg** function. This function makes the dialog box visible when the message queue is empty. This means that a dialog box defined as invisible will not become visible as long as there is input for it. This allows a user to type ahead and even dismiss the dialog box before the dialog box becomes visible, thus saving the time needed to draw the dialog box. The **WinDlgBox** function is equivalent to the following code fragment:

```
dlg = WinLoadDlg(...);
result = WinProcessDlg(dlg);
WinDestroyWindow(dlg);
return (result);
```

WinEnumDlgItem Searches dialog-box child windows for the next control window that fits a specified characteristic. An application can specify a child window from which to start the search; this facilitates repeated linear searches for the next occurrence of a particular type of child window. This function allows an application to identify the next tabstop or group item.

WinLoadDlg Loads a dialog resource from a specified resource-file module (NULL indicates the current application's executable file). The parent and owner window of the new dialog box must specified, as well as a pointer to the dialog procedure for the application. This function returns a handle to the dialog window.

WinMapDlgPoints Converts dialog coordinates into window coordinates and vice versa.

WinMessageBox Creates a modal message box with specified caption, icon, buttons, and text. The **WinMessageBox** function maintains control until the user selects one of the message-box buttons. The return value is a predefined constant that indicates which button is selected.

WinProcessDlg Processes messages for a modal dialog box, making the dialog box visible when the message queue is empty. This means that a dialog box defined as invisible will not become visible as long as there is input for it. This allows a user to type ahead and even dismiss the dialog box before the dialog box becomes visible, thus saving the time needed to draw the dialog box. This function does not return until the dialog procedure calls the **WinDismissDlg** function.

WinQueryDlgItemShort Translates the text of a specified dialog item into a short integer.

WinQueryDlgItemText Retrieves the window text of a specified dialog item.

WinSendDlgItemMsg Sends a message to a child window in the specified dialog box. This function is used for child windows in standard frame windows.

WinSetDlgItemShort Sets the text of the specified dialog item to the text representation of the specified short integer.

WinSetDlgItemText Sets the window text for a specified dialog item.

WinSubstituteStrings Performs a substitution process on a text string, replacing certain marker characters with application-supplied text. When the string *%n* (where "*n*" is a number from 0 through 9) is encountered in the source string, a WM_SUBSTITUTESTRING message is sent to a specified window. This message returns a text string that replaces the characters *%n* in the destination string, which is an exact copy of the source string. This function is important for dialog boxes because the WM_SUBSTITUTESTRING message is received by the dialog procedure, allowing an application to make string substitutions in dialog items that reflect the current environment.

19.6.5 Messages Sent to Dialog Boxes and Dialog Items

The following messages are sent to dialog boxes and dialog items:

WM_INITDLG Sent to a dialog procedure after a dialog box is created but before it is shown. This allows an application to perform run-time initialization for the dialog box, such as filling in default text for entry fields, static-text controls, or list boxes. If any control window in a dialog box requires text substitution, the dialog box receives a WM_SUBSTITUTESTRING message before the WM_INITDLG message. The WM_INITDLG message also contains a window handle of the control window in the dialog box that receives the keyboard focus when the dialog box is shown. An application can change the focus by calling the **WinSetFocus** function for another control window and then returning TRUE. To leave the focus as is, it should return FALSE.

WM_QUERYDLGCODE Sent by the system to control windows in a dialog box to determine the capabilities of the control. Most applications ignore this message unless they are creating custom control windows. The following are predefined result codes for the WM_QUERYDLGCODE message:

Code	**Meaning**
DLGC_BUTTON	Button item
DLGC_CHECKBOX	Check box
DLGC_DEFAULT	Default push button
DLGC_ENTRYFIELD	Entry-field item, handles EM_SETSEL messages
DLGC_MENU	Menu
DLGC_PUSHBUTTON	Non-default push button
DLGC_RADIOBUTTON	Radio button
DLGC_SCROLLBAR	Scroll bar
DLGC_STATIC	Static item
DLGC_TABONCLICK	Next-on-tab control

WM_SUBSTITUTESTRING Sent to a dialog box when it is created (before the WM_INITDLG message is sent) when the system encounters the characters *%n* (where *n* is a number from 0 through 9) in the window text of a dialog control window. This message allows an application dialog procedure to make text substitutions. For example, an application can define dialog-box entry-field text as the characters *%1,* substituting context-appropriate text in the response to the WM_SUBSTITUTESTRING message.

Chapter
20

Painting and Drawing

20.1 Introduction 265

20.2 About Painting and Drawing 265
- 20.2.1 Presentation Spaces and Device Contexts 265
- 20.2.2 Window Regions 266

20.3 Strategies for Painting and Drawing 268
- 20.3.1 When to Draw in a Window 268
- 20.3.2 The WM_PAINT Message 269
  - 20.3.2.1 Drawing the Minimized View 270
- 20.3.3 Drawing Without the WM_PAINT Message 270
- 20.3.4 Three Kinds of Presentation Spaces 270
  - 20.3.4.1 Cached-Micro Presentation Spaces 271
  - 20.3.4.2 Micro Presentation Spaces 272
  - 20.3.4.3 Normal Presentation Spaces 274

20.4 Printing 275

20.5 Summary 275
- 20.5.1 Window Styles for Painting 275
- 20.5.2 Functions 276
- 20.5.3 Messages for Painting 278

20.1 Introduction

This chapter describes presentation spaces, device contexts, and window regions, and how an application uses them for painting and drawing. (For information on functions that are specifically designed for graphics production, see Chapter 21, "Drawing in Windows," and Part 3, "Graphics Programming Interface.") You should also be familiar with the following topics:

- Standard user-interface guidelines
- Standard frame windows
- Window messages and message queues
- Presentation spaces and device contexts
- Graphics programming interface (GPI)

20.2 About Painting and Drawing

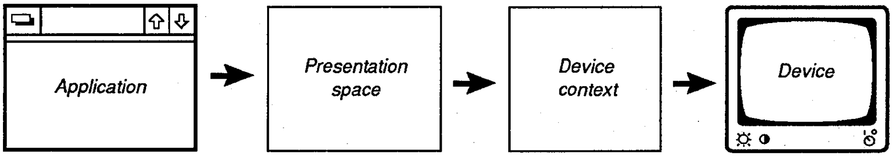
An application typically maintains an internal representation of the data that it is manipulating. The information displayed by a screen, window, or by printed copy is a visual representation of some portion of that data. MS OS/2 provides a rich environment for displaying information in a variety of ways. This chapter introduces concepts and strategies necessary to make your application function smoothly and coöperatively in the MS OS/2 display environment.

20.2.1 Presentation Spaces and Device Contexts

A presentation space is a data structure maintained by the operating system that describes the drawing environment for an application. An application can create and hold several presentation spaces, each describing a different drawing environment. All drawing in an MS OS/2 application must be directed into a presentation space.

Each presentation space is associated with a device context that describes the physical device where graphics commands are displayed. The device context translates graphics commands made to the presentation space into commands that cause the physical device to display information. Typical device contexts are the screen (windows), printers and plotters, and off-screen memory bitmaps. Figure 20.1 shows how graphics commands from an application go through a presentation space, to a device context, and then to the physical device.

Figure 20.1
Application-to-Device Path



By creating presentation spaces and associating them with particular device contexts, an application can control where its graphics output appears. Because a presentation space and device context isolate the application from the physical details of displaying graphics, the same graphics commands can typically be used for many types of displays. This virtualization of output can reduce the amount of display code that an application needs to support.

This chapter discusses how an application sets up its presentation spaces and device contexts before drawing and how to use window-drawing functions. Other chapters in this manual discuss the individual graphics routines available in MS OS/2.

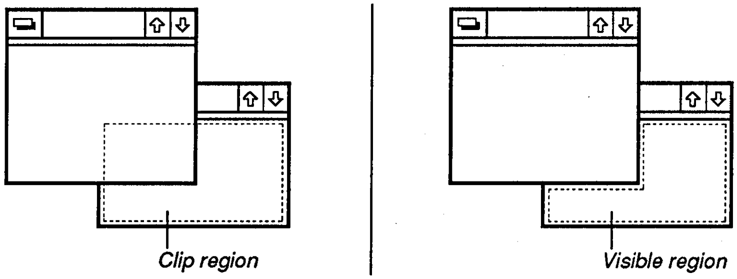
20.2.2 Window Regions

A window and the presentation space associated with it have three regions that control where drawing takes place in the window. These regions ensure that the application does not draw outside the boundaries of the window or intrude into the space of an overlapping window.

Region	Description
Update region	This region represents the area of the window that needs to be redrawn. This region changes when overlapping windows change their Z order or when an application explicitly adds an area to the update region to force a window to be painted.
Clip region	This region and the visible region determine where drawing takes place. Applications can change the clip region to limit drawing to a particular portion of a window. Typically, a presentation space is created with a clip region equal to NULL, which means that there is no clipping supplied by this region.
Visible region	This region and the clip region determine where drawing takes place. The system changes the visible region to represent the portion of a window that is visible. Typically, the visible region is used to mask out overlapping windows. When the application calls the **WinBeginPaint** function in response to a WM_PAINT message, the system sets the visible region to the intersection of the visible region and the update region to produce a new visible region. Applications cannot change the visible region directly.

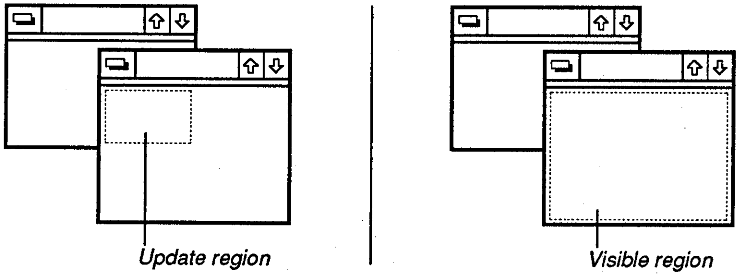
Whenever drawing occurs in a window's presentation space, the output is clipped to the intersection of the visible region and the clip region. Figure 20.2 shows how the intersection of the visible region and the clip region of a window that is behind another window prevents drawing in the back window from intruding into the front window. The clip region includes the overlapped part of the back window, but the visible region excludes that portion of the back window. The system maintains the visible region to protect other windows on the screen and the application maintains the clip region to specify the portion of the window in which it draws. Together, these two regions provide safe and controllable clipping.

Figure 20.2
Clip Region and Visible Region



The update region is manipulated by both the system and the application to further control drawing. For example, if the windows shown in Figure 20.2 switch positions front to back, several changes occur in the regions of both windows. The system adds the lower-right corner of the new front window to that window's visible region. The system also adds that corner area to the window's update region, as shown in Figure 20.3. Adding an area to this window's update region causes the window's window procedure to receive a WM_PAINT message. During the processing of the WM_PAINT message, the system sets the new visible region to be the intersection of the previous visible region and the update region. With this restricted visible region, only the appropriate part of the window is redrawn—the lower-right corner.

Figure 20.3
Update Region and Visible Region



20.3 Strategies for Painting and Drawing

The following sections discuss drawing strategies for an MS OS/2 application. Because an application shares the screen with other windows and applications, drawing must not interfere with other applications and windows. When these strategies are followed, your application will coexist with other applications and still take full advantage of the graphics capabilities of MS OS/2.

20.3.1 When to Draw in a Window

Ideally, all drawing in a window should be done during the processing of the WM_PAINT message. Applications maintain an internal representation of what should be displayed in the window, such as text or a linked list of graphics objects, and use the WM_PAINT message as a cue to display a visual representation of that data in the window.

To route all display output through the WM_PAINT message, an application should not draw on the screen at the time its data changes. Instead, the application should update the internal representation of the data and then call the **WinInvalidateRect** or **WinInvalidateRegion** function to invalidate the portion of the window that needs to be redrawn. Of course, it is often much more efficient to draw directly in a window without relying on the WM_PAINT message—for example, when drawing and redrawing an object for a user who is dragging or sizing with the mouse.

If the window has the WS_SYNCPAINT or CS_SYNCPAINT style, invalidating a portion of the window causes a WM_PAINT message to be sent to the window immediately. Because sending a message is essentially like making a function call, the actions corresponding to the WM_PAINT message are carried out before the call that caused the invalidation returns—that is, the painting is synchronous.

If the window does not have the WS_SYNCPAINT or CS_SYNCPAINT style, invalidating a portion of the window causes the invalidated region to be added to the window's update region. The next time the application calls the **WinGetMsg** or **WinPeekMsg** function when there are no other messages in the queue and the update region for the window is not empty, the application is sent a WM_PAINT message. If there are many messages in the queue the painting occurs after the invalidation—that is, the painting is asynchronous. Painting for windows that do not have the WS_SYNCPAINT or CS_SYNCPAINT style is a low-priority operation; all other messages are processed first. Because a WM_PAINT message is not posted to the queue in this case, all invalidation operations since the last WM_PAINT message are consolidated into a single WM_PAINT message the next time the application has no messages in the queue.

There are advantages to both synchronous and asynchronous painting. Windows that have simple painting routines should be painted synchronously. Most of the system-defined control windows, such as buttons and frame controls, are painted synchronously because they can be painted quickly without interfering with the responsiveness of the program. Windows with more time-consuming painting operations should be painted asynchronously so that the painting can be initiated only when there are no other pending messages that might otherwise be blocked while waiting for the window to be painted. Also, windows that use an incremental approach to invalidating small portions of the window should usually allow those operations to consolidate into a single asynchronous WM_PAINT message, rather than a series of synchronous WM_PAINT messages.

If necessary, an application can call the **WinUpdateWindow** function to cause an asynchronous window to update itself without going through the event loop. **WinUpdateWindow** sends a WM_PAINT message directly to the window if the window's update region is not empty.

20.3.2 The WM_PAINT Message

A window receives a WM_PAINT message whenever its update region is not NULL. A window procedure should respond to a WM_PAINT message by calling the **WinBeginPaint** function, drawing to fill in the update areas, and then calling the **WinEndPaint** function.

The **WinBeginPaint** function returns a handle to a presentation space that is associated with the device context for the window and that has a visible region equal to the intersection of the window's update region and its visible region. This means that only those portions of the window that need to be redrawn are drawn. Attempts to draw outside this region are clipped and do not appear on the screen.

If the application maintains its own presentation space for the window, it can pass that handle of the presentation space to the **WinBeginPaint** function, which modifies the visible region of the presentation space and passes the handle of the presentation-space back to the caller. If the application does not have its own presentation space, it can pass a NULL presentation-space handle and the system will return a cached-micro presentation space for the window. In either case, the application can use the presentation space to draw in the window.

The **WinBeginPaint** function takes a pointer to a **RECTL** structure that it fills in with the coordinates of the rectangle enclosing the area to update. The application can use this rectangle to optimize drawing, by drawing only those portions of the window that intersect with the rectangle. If an application passes a NULL pointer for the rectangle argument, the application draws the entire window and relies on the clipping mechanism to filter out the unneeded areas.

After the **WinBeginPaint** function sets the update region of a window to NULL, the application does the drawing necessary to fill the update areas. If an application handles a WM_PAINT message and does not call **WinBeginPaint** or otherwise empty the update region, the application continues to receive WM_PAINT messages as long as the update region is not empty.

Once the application has finished drawing, it should call the **WinEndPaint** function to restore the presentation space to its former state. When a cached-micro presentation space is returned by the **WinBeginPaint** function, the presentation space is returned to the system for reuse. If the application supplies its own presentation space to **WinBeginPaint**, the presentation state is restored to its previous state.

20.3.2.1 Drawing the Minimized View

When an application creates a standard frame window, it has the option of specifying an icon that the system will use to represent the application in its minimized state. Typically, if an icon is supplied, the system draws the icon in the minimized window and labels the icon with the name of the window. If the application does not specify the FS_ICON style for the window, the window receives a WM_PAINT message when it is minimized. The code in the window procedure that handles the WM_PAINT message can determine if the frame window is currently minimized and then draw accordingly. Notice that because the WS_MINIMIZED style is relevant only for the frame window, not for the client, the window procedure checks the frame window rather than the client window. The following code fragment shows how to draw a window in the minimized state and the normal state:

```
case WM_PAINT:
    hps = WinBeginPaint(hwnd, NULL, &rect);

    /* See if the frame window (client's parent) is minimized. */

    ulStyle = WinQueryWindowULong(WinQueryWindow(hwnd, QW_PARENT,
        FALSE), QWL_STYLE);

    if (ulStyle & WS_MINIMIZED) {
        .
        . /* paints the minimized state */
        .
    }
    else {
        .
        . /* paints the normal state    */
        .
    }
    WinEndPaint(hps);
    return 0L;
```

20.3.3 Drawing Without the WM_PAINT Message

An application can draw in a window's presentation space if it has not received a WM_PAINT message. As long as there is a presentation space for the window, an application can draw into the presentation space and avoid intruding into other windows or the desktop. Applications that draw without using the WM_PAINT message typically call the **WinGetPS** function to obtain a cached-micro presentation space for the window and call the **WinReleasePS** function when they have finished drawing. An application can also use any of the other types of presentation spaces described in the following sections.

20.3.4 Three Kinds of Presentation Spaces

All drawing must take place within a presentation space. MS OS/2 provides three kinds of presentation spaces for drawing: the normal presentation space, the micro presentation space, and the cached-micro presentation space.

The normal presentation space provides the most functionality, allowing access to all the graphics functions of MS OS/2 and allowing the application to draw to all device types. The normal presentation space is more difficult to use than the other two kinds of presentation spaces and it uses more memory. It is created by using the **GpiCreatePS** function and it is destroyed by using the **GpiDestroyPS** function.

The micro presentation space allows access to only a subset of the MS OS/2 graphics functions, but it uses less memory and is faster than a normal presentation space. The micro presentation space also allows the application to draw to all device types. It is created by using the **GpiCreatePS** function and destroyed by using the **GpiDestroyPS** function.

The cached-micro presentation space provides the least functionality of the three kinds of presentation spaces, but it is the most efficient and easiest to use. The cached-micro presentation space draws only to the screen. It is created and destroyed by using either the **WinBeginPaint** and **WinEndPaint** functions or the **WinGetPS** and **WinReleasePS** functions.

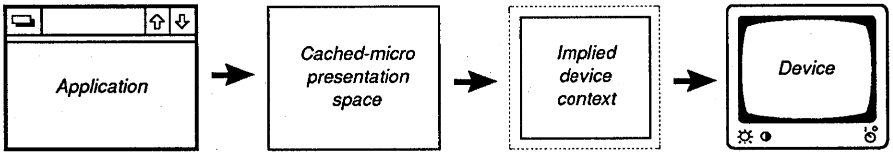
The following sections describe each of the three types of presentation spaces in detail and discuss strategies for using each type in an application. (For more information, see Chapter 30, "Presentation Spaces and Device Contexts.") All three kinds of presentation spaces can be used in a single application. Some windows, especially if they will never be printed, are best served by cached-micro presentation spaces. Other windows may require the more flexible services of micro or normal presentation spaces.

20.3.4.1 Cached-Micro Presentation Spaces

The cached-micro presentation space provides the simplest and most efficient drawing environment. It can be used only for drawing on the screen, typically in the context of a window. It is most appropriate for application tasks that need simple window-drawing functions that do not need to be printed. Cached-micro presentation spaces do not support retained graphics.

After an application draws to a cached-micro presentation space, the drawing commands are routed through an implied device context to the current display. The application does not need information about the actual device context, since it is assumed to be the display. This process makes cached-micro presentation spaces easy for applications to use. Figure 20.4 illustrates this process:

Figure 20.4
Cached-Micro Presentation Space



There are two common strategies for using cached-micro presentation spaces in an application. The simplest is to call the **WinBeginPaint** function during the WM_PAINT message, use the resulting cached-micro presentation space to draw in the window, and then return the presentation space to the system by calling the **WinEndPaint** function. By using this method, the application only interacts with the presentation space when it needs to draw in the presentation space. This method is most appropriate for simple drawing. A disadvantage of this method is that the application must set up any special attributes for the presentation space, such as line color and font, each time a new presentation space is obtained.

A second strategy is for the application to allocate a cached-micro presentation space during initialization, by calling the **WinGetPS** function and saving the resulting presentation-space handle in a static variable. The application can then set attributes in the presentation space that persist for the life of the program. The presentation-space handle can be used as an argument to the **WinBeginPaint** function each time the window gets a WM_PAINT message; the system modifies the visible region and returns the presentation space to the application with its attributes intact. This strategy is appropriate for applications that need to customize their window-drawing attributes.

A presentation space that is obtained by calling the **WinGetPS** function should be released by calling **WinReleasePS** when the application has finished using it. (Typically, this will be during program termination.) A presentation space that is obtained by calling the **WinBeginPaint** function should be released by calling the **WinEndPaint** function, typically as the last part of processing a WM_PAINT message.

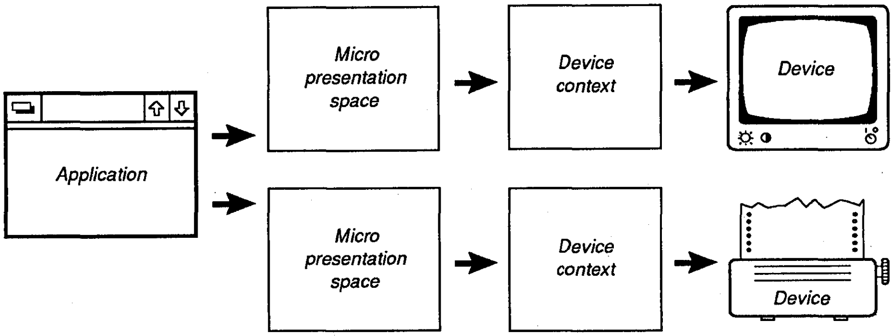
20.3.4.2 Micro Presentation Spaces

The main advantage of a micro presentation space over a cached-micro presentation space is that it can be used for printing as well as for painting in a window. An applications that uses a micro presentation space must explicitly associate it with a device context. This makes the micro presentation space useful for painting to a printer, plotter, or an off-screen memory bitmap.

A micro presentation space does not support the full set of MS OS/2 graphics functions. Unlike a normal presentation space, a micro presentation space does not support retained graphics.

An application that needs to display in a window and print to a printer or plotter typically maintains two presentation spaces: one for the window and one for the printing device. Figure 20.5 shows how an application's graphics output can be routed through separate presentation spaces to produce a screen display and printed copy:

Figure 20.5
Micro Presentation Space



An application creates a micro presentation space by calling the **GpiCreatePS** function. Because a device context must be supplied at the time the micro presentation space is created, an application typically creates a device context and then a presentation space. The following code fragment demonstrates this by obtaining a device context for a window and associating it with a new micro presentation space:

```
hdc = WinOpenWindowDC(...);
hps = GpiCreatePS(...,hdc,...,GPIA_ASSOC);
```

To create a micro presentation space for a device other than the screen, replace the call to the **WinOpenWindowDC** function with a call to the **DevOpenDC** function, which obtains a device context for a device that is not the screen. The device context that is obtained by this call can be used as an argument to the **GpiCreatePS** function.

An application typically creates a micro presentation space during initialization and uses it until termination. Each time the application receives a WM_PAINT message, it should pass the handle of the micro presentation space as an argument to the **WinBeginPaint** function; this prevents the system from returning a cached-micro presentation space. The system modifies the visible region of the supplied micro presentation space and returns the presentation space to the application. This method allows the application to use the same presentation space for all drawing in a specified window.

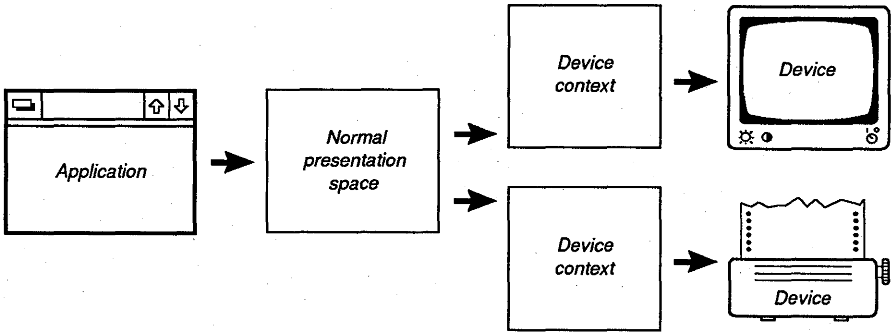
Micro presentation spaces created by using the **GpiCreatePS** function should be destroyed by calling the **GpiDestroyPS** function before the application terminates. Do not call the **WinReleasePS** function to release a presentation space obtained by using the **GpiCreatePS** function. Before terminating, applications should also use the **DevCloseDC** function to close any device contexts opened by using the **DevOpenDC** function. No action is necessary for device contexts obtained with the **WinOpenWindowDC** function, since the system automatically closes these device contexts when destroying the associated windows.

20.3.4.3 Normal Presentation Spaces

The normal presentation space supports the full power of MS OS/2 graphics, including retained graphics. The main advantages of a normal presentation space over the other two presentation-space types are its support of all graphics functions, including retained graphics, and its ability to be associated with many kinds of device contexts.

A normal presentation space can be associated with many different device contexts. Typically, this means that an application creates a normal presentation space and associates it with a window device context for screen display. When the user asks to print, the application associates the same presentation space with a printer device context. Later, the application can reassociate the presentation space with the window device context. A presentation space can be associated with only one device context at a time, but the normal presentation space allows the application to change the device context whenever necessary. Figure 20.6 shows how an application typically routes graphics through one normal presentation space into another device context:

Figure 20.6
Normal Presentation Space



A normal presentation space can be associated with a device context when the normal presentation space is created, or association can be deferred to a later time. The **GpiAssociate** function associates a device context with a normal presentation space after the presentation space has been created. An application typically associates the normal presentation space with a device context when calling the **GpiCreatePS** function and later associates the presentation space with a different device context by calling **GpiAssociate**. To obtain a device context for a window, call the **WinOpenWindowDC** function. To obtain a device context for a device other than the screen, call the **DevOpenDC** function.

An application typically creates a normal presentation space during initialization and uses it until termination. Each time the application receives a WM_PAINT message, it should pass the handle of the normal presentation space as an argument to the **WinBeginPaint** function; this prevents the system from returning a cached-micro presentation space. The system modifies the visible region of the supplied normal presentation space and returns the presentation space to the application. This method allows the application to use the same presentation space for all drawing in a specified window.

Normal presentation spaces created by using the **GpiCreatePS** function should be destroyed by calling the **GpiDestroyPS** function before the application terminates. Do not call the **WinReleasePS** function to release a presentation space obtained by using the **GpiCreatePS** function. Before terminating, applications should also use the **DevCloseDC** function to close any device contexts opened by using the **DevOpenDC** function. No action is necessary for device contexts obtained with the **WinOpenWindowDC** function, since the system automatically closes these device contexts when destroying the associated windows.

20.4 Printing

Although a detailed discussion of printing is beyond the scope of this chapter, printing should be seen as a variation of screen painting. To draw in a window, an application issues graphics calls to a presentation space associated with a screen device context. To print, the application makes graphics calls to a presentation space associated with a printer device context. In an application that supports a what-you-see-is-what-you-get window display, the printing code should be the same as or very similar to the window-display code, as though the printed page were an 8½-by-11-inch window. (Of course, many applications will optimize printing code to take advantage of such properties of the output device as high-resolution page-description languages.)

An application achieves greater device-independence if it does not use pels as its drawing unit. For example, if an application does all its drawing into a presentation space with PU_LOENGLISH units (.01 inch), a 100-unit line is certain to be one inch long on any printing device. The presentation space and device context automatically scale a drawing to compensate for the resolution of the output device.

20.5 Summary

This section summarizes the window styles related to window painting; the functions that control presentation spaces, device contexts, and window regions; and the messages related to window painting.

20.5.1 Window Styles for Painting

Most of the styles relating to window drawing can be set either for the window class (CS prefix) or for an individual window (WS prefix). The following styles control how the system manipulates the window's regions and how the window is notified to paint itself:

WS_CLIPCHILDREN, CS_CLIPCHILDREN All of the child windows of a window with this style are excluded from the window's visible region. This style protects child windows but more time is required when calculating the visible region. This style is normally not necessary, since if the parent and child overlap and are both invalidated, the parent is drawn before the child. If the child is invalidated independently from the parent, only the child is redrawn. If the update region of the parent does not intersect the child, drawing the parent should not disturb the child.

WS_CLIPSIBLINGS, CS_CLIPSIBLINGS Any windows that have the same parent as a window with this style and that are in front of the window are excluded from the window's visible region. This style protects windows with the same parent from being drawn in accidentally but requires more time when calculating the visible region. This style is appropriate for windows that overlap and that have same parent.

WS_PARENTCLIP, CS_PARENTCLIP The visible region for a window with this style is the same as the visible region of the parent window. This style simplifies the calculation of the visible region but is potentially dangerous, since the parent window's visible region is usually larger than the child window. Windows with this style should not draw outside their boundaries.

WS_SAVEBITS, CS_SAVEBITS The system saves the bits underneath a window with this style when the window is displayed. When the window moves or is hidden, the uncovered bits are simply restored by the system; there is no need to add the area to the uncovered window's update region. Because this operation can consume a great deal of memory, it is recommended only for transient windows such as menus and dialog boxes, not for main application windows.

WS_SYNCPAINT, CS_SYNCPAINT Windows that have this style receive WM_PAINT messages as soon as their update regions are not empty and are updated immediately (synchronously). For more details on synchronous painting, see Section 20.3.1.

CS_SIZEREDRAW A window with this class style receives a WM_PAINT message and is completely invalidated whenever the window is resized, even if it is made smaller. (Typically, only the uncovered area of a window is invalidated when a window is resized.) This class style is useful when an application scales graphics to fill the current window.

20.5.2 Functions

The following functions control presentation spaces, device contexts, and window regions:

DevCloseDC Closes a device context created by using the **DevOpenDC** function. Do not use this function to close a device context that was created by using the **WinOpenWindowDC** function.

DevOpenDC Creates a device context for a specified device type.

GpiAssociate Creates an association between a presentation space and a device context. Any subsequent drawing to the presentation space goes to the specified device. This function is typically used only with normal presentation spaces, since micro presentation spaces must be associated with a device context during the call to the **GpiCreatePS** function.

GpiCreatePS Creates a handle of a normal or micro presentation space. The presentation space can be associated with a specified device context at the time of creation (this is mandatory for micro presentation spaces), or it can be associated later by using the **GpiAssociate** function.

GpiDestroyPS Destroys a presentation space. Do not attempt to destroy a cached-micro presentation space by using this function.

WinBeginPaint Returns a handle of a presentation space that has a visible region equal to the intersection of the window's update region and the visible region of the window's presentation space. This function is called when a WM_PAINT message is received. If the *hps* parameter is a valid presentation-space handle, the visible region of the presentation space is modified and the handle is returned. If the *hps* parameter is NULL, the system returns the handle of a cached-micro presentation space. The update region of the window is set to NULL by this function because the system assumes that the caller will do all drawing necessary to fill the invalid region of the window.

WinEnableWindowUpdate Sets the window-visibility state for subsequent drawing, without causing any visible change to the window. This function can be used to defer drawing when making a series of changes to a window.

WinEndPaint Indicates that a drawing operation that was started by using the **WinBeginPaint** function is finished. The *hps* parameter is the presentation-space handle returned by **WinBeginPaint**. A cached-micro presentation space is returned to the system for reuse and the drawing environment of a previous presentation space is restored.

WinExcludeUpdateRegion Removes the update region from the clip region of the specified window. This can be useful for optimization, since it prevents drawing in the update region. The application must restore the clip region when the exclusion is no longer needed.

WinGetClipPS Returns a clipped presentation space for the specified window. The type of clipping depends on the flag settings of the function's parameters. The presentation space should be released by using the **WinReleasePS** function.

WinGetPS Returns the handle of a cached-micro presentation space for the specified window. The presentation space should be released by using the **WinReleasePS** function.

WinGetScreenPS Returns a presentation-space handle that can be used to draw anywhere on the screen.

WinInvalidateRect Adds the specified rectangle to the update region of the window. If the specified window has the WS_SYNCPAINT style, it receives a WM_PAINT message before **WinInvalidateRect** returns. This function can be used to force part of a window to be repainted. If the *fIncludeChildren* parameter is TRUE, any child windows that intersect the invalid rectangle are also invalidated.

WinInvalidateRegion Adds the specified region to the update region of the specified window. If the window has the WS_SYNCPAINT style, it receives a WM_PAINT message before **WinInvalidateRegion** returns. This function can be used to force a portion of a window to be repainted. If the *fIncludeChildren* parameter is TRUE, any child windows that intersect the invalid region are also invalidated.

WinLockWindowUpdate Prevents updates to the specified window and its child windows. This is useful if you need to delay updating during incremental data changes, such as adding items to a list box. Calling **WinLockWindowUpdate**

with NULL for the *hwndLockUpdate* parameter invalidates the windows that were previously locked and causes WM_PAINT messages to be sent to those windows.

WinLockVisRegions Locks or unlocks the visible regions of all windows on the screen. This function is useful to threads because it prevents the visible regions of windows from changing while the thread performs a screen operation, such as copying screen pels into a memory bitmap.

WinOpenWindowDC Returns a device-context handle for the specified window. Attempting to open more than one device context for a given window is an error. The returned device context is automatically closed when the window is destroyed; it must not be closed by using the **DevCloseDC** function.

WinQueryUpdateRect Returns the coordinates of the smallest rectangle that encloses the window's update region.

WinQueryUpdateRegion Obtains the update region of the specified window.

WinQueryWindowDC Returns the device-context handle associated with the specified window.

WinReleasePS Releases the handle of a cached-micro presentation space that was obtained by using the **WinGetPS** or **WinGetClipPS** function. This function should not be used for presentation-space handles that are not cached—that is, **WinReleasePS** should not be used for presentation spaces obtained by using the **GpiCreatePS** function.

WinValidateRect Removes the specified rectangle from the update region of the window. This function can be used to avoid duplicate drawing, by signaling that part of the window has been drawn without using the WM_PAINT message.

WinValidateRegion Removes the specified region from the update region of the window. This function can be used to avoid duplicate drawing, by signaling that part of the window has been drawn without using the WM_PAINT message.

WinWindowFromDC Returns the handle of the window associated with the specified device-context.

20.5.3 Messages for Painting

The following message is used in the management of window painting:

WM_PAINT This message is sent when some portion of the window needs to be repainted. The window procedure should respond to the message by painting the relevant portion of the window.

Chapter

21

Drawing in Windows

21.1 Introduction 281

21.2 Window-Drawing Functions 281

21.2.1 Points and Rectangles 281

21.3 Using Window-Drawing Functions 282

21.3.1 Working with Points and Rectangles 282

21.3.2 Scrolling Window Contents 283

21.3.3 Drawing Bitmaps 284

21.3.4 Drawing Text 285

21.4 Summary 285

21.1 Introduction

This chapter describes the functions that are specifically designed to help you draw in windows. (For information on the complete set of drawing functions, see Part 3, "Graphics Programming Interface.") You should also be familiar with the following topics:

- Standard user-interface guidelines
- Standard frame windows
- Presentation spaces and device contexts
- Graphics programming interface (GPI)

21.2 Window-Drawing Functions

The functions described in this chapter overlap the functionality of similar drawing functions provided by the GPI sections of MS OS/2. The difference between the **Gpi** drawing functions and these window-drawing functions is that the functions described in this chapter are designed specifically for drawing in windows. Because these window-drawing functions are less general than the **Gpi** functions, they are somewhat easier to use, but they also offer fewer capabilities than the complete set of **Gpi** functions. The functions described in this chapter offer a quick and easy way to create simple graphics. Programmers seeking more functionality should use the **Gpi** functions of MS OS/2.

21.2.1 Points and Rectangles

All drawing in a window takes place in the context of the window's coordinate system. Locations in the window are described by **POINTL** structures that contain an *x*- and a *y*-coordinate for the location. The lower-left corner of a window always has the coordinates (0,0). The **POINTL** structure has the following form:

```
typedef struct _POINTL  {
    LONG  x;
    LONG  y;
} POINTL;
```

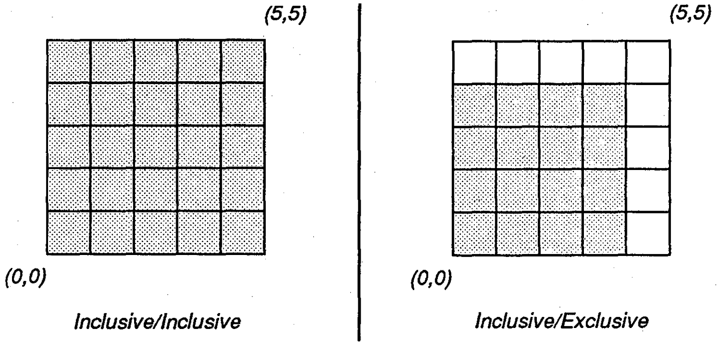
The **RECTL** structure defines a rectangular area in a window. This structure is made up of two points that define the lower-left and upper-right corners of the rectangle. The **RECTL** structure has the following form:

```
typedef struct _RECTL {
    LONG  xLeft;
    LONG  yBottom;
    LONG  xRight;
    LONG  yTop;
} RECTL;
```

An empty rectangle is a rectangle that has no area: the right coordinate is less than or equal to the left coordinate or the top coordinate is less than or equal to the bottom coordinate.

There are two types of rectangles in MS OS/2: inclusive-exclusive and inclusive-inclusive. In inclusive-exclusive rectangles the lower-left coordinate of the rectangle is included in the rectangle area while the upper-right coordinate is excluded from the rectangle area. In an inclusive-inclusive rectangle, both the lower-left and the upper-right coordinates are included in the rectangle. Figure 21.1 shows both types of rectangles:

Figure 21.1
Rectangle Types



The dimensions of an inclusive-exclusive rectangle can be calculated as follows:

```
cx = rcl.xRight - rcl.xLeft;
cy = rcl.yTop - rcl.yBottom;
```

The dimensions of an inclusive-inclusive rectangle can be calculated as follows:

```
cx = (rcl.xRight - rcl.xLeft) + 1;
cy = (rcl.yTop - rcl.yBottom) + 1;
```

In general, graphics operations involving device coordinates (such as regions, bitmaps and bit blits, and window management) use inclusive-exclusive rectangles. All other graphics operations, such as **Gpi** functions that define paths, use inclusive-inclusive rectangles.

21.3 Using Window-Drawing Functions

The functions described in this chapter are intended for simple drawing. The rectangle functions manipulate and combine rectangles. The drawing and scrolling functions perform within a presentation space's coordinate system. For more advanced drawing you should use the **Gpi** functions of MS OS/2.

21.3.1 Working with Points and Rectangles

MS OS/2 includes many functions for manipulating rectangles. Many of these functions change the coordinates of a rectangle. Other functions draw in a presentation space, using a rectangle to position the drawing operation.

The **WinFillRect** function fills a rectangle with a specified color. For example, to fill an entire window with blue in response to a WM_PAINT message, you could use the following code fragment, which is taken from a window procedure:

```
case WM_PAINT:
    hps = WinBeginPaint(hwnd, NULL, NULL);
    WinQueryWindowRect(hwnd, &rect);
    WinFillRect(hps, &rect, CLR_BLUE);
    WinEndPaint(hps);
    return OL;
```

A more efficient way of painting a client window is to pass a rectangle to the **WinBeginPaint** function. The rectangle will be set to the coordinates of the rectangle that encloses the update region of the window. Drawing in this rectangle updates the window; this can make drawing faster if only a small portion of the window needs to be painted. This method is shown in the following code fragment. Notice that the **WinFillRect** function uses the presentation space and a rectangle defined in window coordinates to guide the paint operation.

```
case WM_PAINT:
    hps = WinBeginPaint(hwnd, NULL, &rect);
    WinFillRect(hps, &rect, CLR_BLUE);
    WinEndPaint(hps);
    return OL;
```

Of course, you can draw the entire window during the WM_PAINT message, but the graphics output will be clipped to the update region.

The default method of indicating that a particular portion of a window has been selected is for the **WinInvertRect** function to invert the rectangle's bits.

The rest of the rectangle functions are mathematical and do not draw. They are used to manipulate and combine rectangles to produce new rectangles, which can then be used for drawing operations.

The **WinMapWindowPoints** function converts the points from one window-coordinate space to another window-coordinate space. If one of the specified windows is HWND_DESKTOP, then screen coordinates are used. This function is useful for converting from window coordinates to screen coordinates and back again.

21.3.2 Scrolling Window Contents

An application normally responds to a click in a scroll bar by scrolling the contents of the window. This operation typically has three parts. First, the application changes its internal data-representation state to show what portion of the image should now be in the window. Next, the application moves the current image in the window. Finally, the application draws in the area that has been uncovered by the scrolling operation.

For example, a simple text editor may display a small portion of several pages of text in a window. When the user clicks the down arrow of the vertical scroll bar, the application should move all the text up one line and display the next line at the bottom of the window.

When the user clicks the down arrow of the vertical scroll bar, the client window of the frame window that owns the scroll bar receives a message. The application responds to this message by changing its internal data-representation state to show which line of text is topmost in the window, scrolling the text in the window up one line, and drawing the new line at the bottom of the window. There is normally no need to completely redraw the entire window, because the scrolled portion of the image remains valid.

The **WinScrollWindow** function allows applications to scroll the contents of their windows. **WinScrollWindow** scrolls a specified rectangular area of the window by a specified *x*- and *y*-offset (in window coordinates). By setting the SW_INVALIDATERGN flag for this function, the areas uncovered by the scroll are automatically added to the window's update region, causing a WM_PAINT message to be sent to the window for those areas.

For example, in the simple text editor described earlier, the following call scrolls the text up one line (assuming that *iVScrollInc* is the height of the current font) and adds the uncovered area at the bottom of the window to the update region.

```
/* Scroll, adding new area to update region. */

WinScrollWindow (hwnd, /* window                          */
    0,                 /* x-displacement                  */
    -(iVScrollInc),    /* y-displacement                  */
    NULL,              /* scroll rectangle is entire window */
    NULL,              /* clip rectangle is entire window   */
    NULL,              /* update region                   */
    NULL,              /* update rectangle                */
    SW_INVALIDATERGN); /* flags                           */
```

When the uncovered area at the bottom of the window is added to the window's update region, a WM_PAINT message is sent to the window. When the message is received, the window draws the line of text at the bottom of the window. If the window has the WS_SYNCPAINT style, the WM_PAINT message is sent to the window before the **WinScrollWindow** function returns.

To optimize scrolling speed for repeated scrolling operations, you can omit the SW_INVALIDATERGN flag from the call to the **WinScrollWindow** function. This prevents **WinScrollWindow** from adding the invalid region uncovered by the scroll to the window's update region. If the SW_INVALIDATERGN flag is omitted, you must pass a region or rectangle to **WinScrollWindow**. The rectangle or region will contain the area that needs to be updated after scrolling.

21.0.1 Drawing Bitmaps

The **WinDrawBitmap** function draws a bitmap, specified by a bitmap handle, in a specified rectangle. This function allows you to shrink or enlarge the bitmap from the source rectangle to the destination. **WinDrawBitmap** can also draw in several different copy modes, including using the OR operator to combine source and destination pels.

21.3.4 Drawing Text

There are many ways to draw text in a window in an MS OS/2 application. The simplest way is to use the **WinDrawText** function, which draws a single line of text in a specified rectangle, using a variety of alignment methods.

The **WinDrawText** function allows you to set a flag so that the function does not draw any text; instead, it returns the number of characters in the string that will fit in the specified rectangle. For a section of running text, an application could alternate between computation and calls to **WinDrawText** to draw successive lines of text. The DT_WORDBREAK flag in the **WinDrawText** function can be set, when performing this kind of repetitive operation, to put line breaks on word boundaries rather than between arbitrary characters.

21.4 Summary

The following functions are provided specifically for drawing in windows and manipulating rectangles:

WinCopyRect Copies the coordinates of one rectangle to another.

WinDrawBitmap Draws a bitmap in a rectangle, scaling the bitmap to fit, if necessary.

WinDrawBorder Draws a rectangle.

WinDrawText Draws a single line of formatted text in a specified rectangle.

WinEqualRect Returns TRUE if two rectangles have the same coordinates or FALSE if the coordinates are different.

WinFillRect Fills a rectangle with a specified color.

WinInflateRect Enlarges or shrinks the rectangle horizontally and vertically by the specified amounts. If the specified values are negative, the rectangle is inset.

WinIntersectRect Calculates the intersection of two source rectangles and returns the result in a destination rectangle, or returns FALSE if the result is an empty rectangle.

WinInvertRect Inverts the pels in a rectangle.

WinIsRectEmpty Returns TRUE if the rectangle is empty (that is, if the right coordinate is less than or equal to the left coordinate or if the top coordinate is less than or equal to the bottom coordinate).

WinMapWindowPoints Converts points from one window-coordinate system to another.

WinOffsetRect Changes the left and right coordinates of a rectangle by the specified offsets.

WinPtInRect Returns TRUE if the point is inside the rectangle or FALSE if it is not.

WinScrollWindow Scrolls the contents of a rectangular area of a window. If the proper flags are set, the area uncovered by scrolling is added to the window's update region, causing a WM_PAINT message to be sent. The application can then respond to the message by drawing in the invalidated region of the window.

WinSetRect Sets the rectangle's coordinates.

WinSetRectEmpty Sets the coordinates of a rectangle to (0,0,0,0).

WinSubtractRect Subtracts the coordinates of one rectangle from those of another rectangle. This function returns FALSE if the result is an empty rectangle.

WinUnionRect Calculates a rectangle that encloses two source rectangles. This function returns FALSE if the result is an empty rectangle.

Chapter

22

Mouse Pointers and Icons

22.1 Introduction 289

22.2 About Mouse Pointers 289

22.2.1 Mouse Pointers and Icon Bitmaps 289

22.2.2 Mouse-Pointer Hot Spot 290

22.3 Using a Mouse Pointer in an Application 290

22.3.1 Creating or Loading a Mouse Pointer 290

22.3.2 Changing the Mouse Pointer 290

22.3.3 Drawing an Icon 291

22.3.4 System Bitmaps 291

22.4 Summary 291

22.4.1 Predefined Mouse Pointers 292

22.4.2 Mouse-Pointer Functions 293

22.1 Introduction

This chapter describes how to use mouse pointers and icons in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Resources and using the MS OS/2 Resource Compiler (**rc**)
- Window messages and message queues
- Bitmaps

22.2 About Mouse Pointers

Mouse pointers are special bitmaps that MS OS/2 uses to show a user the current location of the mouse on the screen. The mouse pointer moves around the screen in response to user manipulation.

Mouse pointers are also used to draw icons on the screen, such as graphics in message boxes and icons that represent minimized windows on the desktop. The data structures for mouse pointers and icon bitmaps are identical.

22.2.1 Mouse Pointers and Icon Bitmaps

Mouse pointers and icons are made up of monochrome bitmaps that MS OS/2 uses to paint an image of the pointer or icon on the screen. A monochrome bitmap is a series of bytes. Each bit corresponds to a single pel in the image (the bitmap representing the display typically has four bits for each pel).

A mouse pointer or icon bitmap is always twice as tall as it is wide. The top half of the bitmap is an AND mask, where the bits are combined using the AND operator with the screen bits where the pointer is being drawn. The lower half of the bitmap is the XOR mask, which is combined using the XOR operator with the destination screen bits.

The combination of the AND and XOR masks results in four possible colors in the bitmap. The pels of an icon or pointer can be black, white, transparent (the screen color beneath the pel), or inverted (inverting the screen color beneath the pel). Figure 22.1 shows the relationship of the bit values in the AND and XOR masks:

Figure 22.1
Bit Values in the AND and XOR Masks

AND mask	0	0	1	1
XOR mask	0	1	0	1
Result	Black	White	Transparent	Inverted

22.2.2 Mouse-Pointer Hot Spot

Each mouse pointer has a hot spot defined as an *x*- and *y*-offset from the lower-left corner of the mouse-pointer bitmap. The hot spot defines the single point that represents the mouse-pointer location. For the arrow-shaped pointer, the hot spot is at the tip of the arrow. For the cross-hairs pointer, the hot spot is at the center of the cross. Each pointer has its own hot spot.

22.3 Using a Mouse Pointer in an Application

Applications typically use mouse-pointer resources to control the appearance of the mouse pointer and to draw icons in windows. The following sections show how to use mouse pointers in applications.

22.3.1 Creating or Loading a Mouse Pointer

Before an application can use a mouse pointer, it must first receive a handle to the pointer. Most applications load mouse pointers from the system or their own resource file. MS OS/2 maintains many predefined mouse pointers that an application can use by calling the **WinQuerySysPointer** function. System mouse pointers include all the standard mouse-pointer shapes and message-box icons.

You can also load mouse pointers that are defined in the resource file for your application. Typically, you define the mouse-pointer resource using Icon Editor or a similar program. The resulting mouse pointer or icon can then be included in your resource file. This is done by pulling the Icon Editor data file into your resource file using the key word **POINTER**, a resource ID number, and a filename for the mouse-pointer data created by the Icon Editor. When the mouse-pointer resource is included in the resource file, you can use it by calling the **WinLoadPointer** function, specifying the pointer-resource ID and the resource-module handle. Typically, the resource resides in the executable file of the application, so you can supply NULL for the module handle to indicate the current application resource file.

Finally, you can create mouse pointers at run time by constructing a bitmap for the pointer and calling the **WinCreatePointer** function. The bitmap must be twice as tall as it is wide, with the first half defining the AND mask and the second half defining the XOR mask. You also must specify the hot spot when you create a mouse pointer. The handle returned can be used to set or draw the mouse pointer.

22.3.2 Changing the Mouse Pointer

Once you create or load a mouse pointer, you can change its shape by calling the **WinSetPointer** function. The following are three main situations where an application typically changes the shape of the mouse pointer:

- When an application receives a WM_MOUSEMOVE message, there is an opportunity to change the mouse pointer based on its location the window. If you want the standard arrow pointer, pass this message on to the **WinDefWindowProc** function.
- When an application is about to start a time-consuming process during which it will not accept user input, the application should display the “system-wait”

mouse pointer. This pointer is shaped like an hourglass, indicating that the user must wait. Once this process is complete, the application should reset the mouse pointer to its former shape.

- The following code fragment shows how to save the current mouse pointer, set the hourglass pointer, and then restore the original mouse pointer. Notice that the hourglass pointer is also saved in a global variable so that the application can return it when responding to a WM_MOUSEMOVE message during a time-consuming process.

```
/* Get current pointer. */

hptrOld = WinQueryPointer(HWND_DESKTOP);

/* Get wait mouse pointer. */

hptrWait = WinQuerySysPointer(HWND_DESKTOP, SPTR_WAIT, FALSE);

/* Save wait pointer to use in WM_MOUSEMOVE processing.*/

hptrCurrent = hptrWait;

/* Set mouse pointer to wait pointer.*/

WinSetPointer(HWND_DESKTOP, hptrWait);

/* Do time-consuming operation; restore original mouse pointer.*/

WinSetPointer(HWND_DESKTOP, hptrOld);
```

- The mouse pointer can be used to indicate the current operational mode of an application. For example, a paint program with a palette of drawing tools should change the mouse pointer shape to indicate which drawing tool is currently in use.

22.3.3 Drawing an Icon

You can use mouse-pointer resources to draw icons. The **WinDrawPointer** function draws a specified mouse pointer in a specified presentation space. Many of the predefined system mouse pointers are standard icons displayed in message boxes.

22.3.4 System Bitmaps

In addition to mouse pointers and icons defined by the system, you can use standard system bitmaps by calling the **WinGetSysBitmap** function. This function returns a bitmap handle that is passed to the **WinDrawBitmap** function or one of the **Gpi** bitmap calls. The system uses standard bitmaps to draw portions of control windows such as the system menu, the minimize/maximize box, and scroll-bar arrows.

22.4 Summary

The following sections describe the system mouse pointers and the functions available for mouse pointers, icons, and system bitmaps.

22.4.1 Predefined Mouse Pointers

MS OS/2 provides many predefined mouse-pointer shapes. You receive a handle to these pointers by using one of the following constants as an argument to the **WinQuerySysPointer** function:

SPTR_APPICON Square icon.

SPTR_ARROW Arrow that points to the upper-left corner of the screen.

SPTR_ICONERROR Icon containing an exclamation point, used in a warning dialog box.

SPTR_ICONINFORMATION Octagon-shaped icon containing the image of a human hand, used in a warning dialog box.

SPTR_ICONQUESTION Icon containing a question mark, used in a query dialog box.

SPTR_ICONWARNING Icon containing an asterisk, used in a warning dialog box.

SPTR_MOVE Four-headed arrow, used when dragging an object or window around the screen.

SPTR_SIZENESW Two-headed diagonal arrow that points from the upper-right (northeast) window border to the lower-left (southwest) window border, used when sizing a window.

SPTR_SIZENS Two-headed arrow that points from top to bottom (north to south), used when sizing a window.

SPTR_SIZENWSE Two-headed diagonal arrow that points from the upper-left (northwest) window border to the lower-right (southeast) window border, used when sizing a window.

SPTR_SIZEWE Two-headed arrow that points from left to right (west to east), used when sizing a window.

SPTR_TEXT Text-insertion and selection pointer, often called the I-beam pointer.

SPTR_WAIT Hourglass, used to indicate that a time-consuming operation is in progress.

MS OS/2 contains a second set of predefined mouse pointers that are used as icons in Presentation Manager. The resulting mouse pointer must be explicitly destroyed using the **WinDestroyPointer** function before the application terminates. These icons are available for application use by supplying one of the following constants to the **WinQuerySysPointer** function:

SPTR_FILE Icon representing a single file (in the shape of a single sheet of paper). It must be explicitly destroyed before the application terminates.

SPTR_FOLDER Icon representing a file folder. It must be explicitly destroyed before the application terminates.

SPTR_ILLEGAL Circular icon containing a slash, used to indicate an illegal operation. It must be explicitly destroyed before the application terminates.

SPTR_MULTFILE Icon representing multiple files. It must be explicitly destroyed before the application terminates.

SPTR_PROGRAM Icon representing an executable file. It must be explicitly destroyed before the application terminates.

22.4.2 Mouse-Pointer Functions

The following mouse-pointer functions can be used in your application:

WinCreatePointer Creates a mouse pointer from a bitmap.

WinDestroyPointer Destroys a pointer or an icon. A pointer can only be destroyed by the thread that created it. This function decrements the use count of processes that have accessed the pointer. The pointer is deleted when the use count reaches zero.

WinDrawPointer Draws the specified mouse pointer (or icon) in a presentation space.

WinGetSysBitmap Returns a handle to one of the standard bitmaps provided by the system. The bitmap returned can be used for any routine bitmap operations. The **WinGetSysBitmap** function returns a new copy of the system bitmap each time it is called.

The following bitmaps are available:

Bitmap	Description
SBMP_BTNCORNERS	Push-button corners.
SBMP_CHECKBOXES	Check-box or radio-button check mark.
SBMP_CHILDSYSMENU	Smaller version of the system-menu bitmap, used in child windows.
SBMP_DRIVE	Symbol used by File System to indicate a disk drive.
SBMP_FILE	Symbol used by File System to indicate a file.
SBMP_FOLDER	Symbol used by File System to indicate subdirectories.
SBMP_MAXBUTTON	Maximize button.
SBMP_MENUATTACHED	Symbol used to indicate that a menu item has an attached, hierarchical menu.
SBMP_MENUCHECK	Menu check mark.
SBMP_MINBUTTON	Minimize button.
SBMP_PROGRAM	Symbol used by File System to indicate that a file is an executable program.
SBMP_RESTOREBUTTON	Restore button.

Bitmap	Description
SBMP_SBDNARROW	Scroll-bar down arrow.
SBMP_SBLFARROW	Scroll-bar left arrow.
SBMP_SBRGARROW	Scroll-bar right arrow.
SBMP_SBUPARROW	Scroll-bar up arrow.
SBMP_SIZEBOX	Symbol used to indicate an area of a window that a user can click to resize the window.
SBMP_SYSMENU	System menu.
SBMP_TREEMINUS	Symbol used by File System to indicate that an entry in the directory tree is empty.
SBMP_TREEPLUS	Symbol used by File System to indicate that an entry in the directory tree contains more files.

WinLoadPointer Loads a pointer resource from a resource file into the system and returns a mouse-pointer handle. A new copy of the pointer is created each time this function is called. Once used, the pointer created by this function must be explicitly destroyed by using the **WinDestroyPointer** function.

WinQueryPointer Returns a mouse-pointer handle for the current mouse pointer and can be used to restore the mouse pointer after any temporary changes.

WinQueryPointerInfo Retrieves information about a specific mouse pointer, including its bitmap and its hot-spot coordinates.

WinQueryPointerPos Retrieves the current mouse-pointer position in screen coordinates.

WinQuerySysPointer Returns a handle to the specified system mouse pointer. You can specify whether you want a handle to the system's copy of the mouse pointer or if you want a separate copy for modification.

WinSetPointer Sets the current mouse pointer.

WinSetPointerPos Sets the mouse-pointer position in screen coordinates.

WinShowPointer Shows or hides the mouse pointer.

Chapter

23

Cursors

23.1 Introduction 297

23.2 About Cursors 297

23.3 Using Cursors in an Application 297

23.3.1 Creating a Cursor 297

23.3.2 Destroying a Cursor 298

23.3.3 Showing the Cursor 298

23.3.4 Positioning the Cursor 298

23.4 Summary 298

23.1 Introduction

This chapter describes the functions that allow you to use cursors in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window activation and deactivation
- Keyboard focus

23.2 About Cursors

A cursor is a rectangle that can be shown at any location in a window. It is usually used to mark a text-insertion point or to indicate when a control window has the keyboard focus. For example, entry-field controls use a flashing vertical bar to show the insertion point when the control has the keyboard focus. A button control, on the other hand, appears as a halftone rectangle the size of the button when the button has the keyboard focus. MS OS/2 draws and flashes the cursor, freeing the application from handling these details. Note that the cursor has no direct relationship with the mouse pointer.

23.3 Using Cursors in an Application

Typically, applications use cursors to mark the text-insertion point in a text window or to indicate that a window has the keyboard focus.

There can be only one cursor in use by the system at one time, so windows must create and destroy cursors as they gain and lose the keyboard focus. The following code fragment shows how an application should respond to a WM_SETFOCUS message when using a cursor in a particular window:

```
case WM_SETFOCUS:
    if (SHORT1FROMMP(mp2)) {

        /* gain focus */

        WinCreateCursor(...);
        WinShowCursor(hwnd, TRUE);
    } else {

        /* lose focus */

        WinDestroyCursor(hwnd);
    }
    return 0L;
```

23.3.1 Creating a Cursor

An application creates a cursor by calling the **WinCreateCursor** function. Generally, this is done when a window gains the keyboard focus. An application specifies the window in which the cursor will be displayed. This window may be HWND_DESKTOP, an application window, or a control window.

An application specifies the cursor position, in window coordinates, and the cursor height and width. It also specifies whether the cursor rectangle should be filled, framed, flashing, or halftone.

The cursor width is usually zero (nominal border width is used) for text-insertion cursors. This is preferable to a value of 1, since such a fine width is almost invisible on a high-resolution monitor. The cursor width may also be related to the window size; for example, when a button control uses a dotted-line cursor around the button text to indicate focus.

Finally, an application can specify a clipping rectangle, in window coordinates, that controls the cursor clipping region. Typically, the most efficient strategy is to specify NULL, causing the rectangle to clip the cursor to the window rectangle.

23.3.2 Destroying a Cursor

Applications should destroy cursors by calling the **WinDestroyCursor** function when they lose the keyboard focus. It is important that they be destroyed when they lose the focus. Manipulating two cursors in MS OS/2 simultaneously can have unpredictable results and affect other applications.

23.3.3 Showing the Cursor

MS OS/2 maintains a "show level" for the cursor. The cursor is visible when the show level is zero. Each time the cursor is hidden, its show level is incremented. Each time the cursor is shown, its show level is decremented. Because the show/hide relationship is one for one, the show level can never go below zero.

An application should show the cursor when it is first created since the cursor is created with a show level of 1.

MS OS/2 automatically hides the cursor when the **WinBeginPaint** function is called, and shows the cursor when the **WinEndPaint** function is called. Therefore, there is no conflict with the cursor during WM_PAINT processing.

23.3.4 Positioning the Cursor

An application can set the position of an existing cursor by calling the **WinCreateCursor** function with the CURSOR_SETPOS flag set. This function can also be used to move a cursor around a window. Position arguments are given in window coordinates. To change the cursor size, destroy the current cursor and then create a new one with the desired size.

23.4 Summary

The following section summarizes the functions related to cursor management.

WinCreateCursor Creates a new cursor or changes the position of an existing cursor. The cursor is created when the window gains the keyboard focus (receives a WM_SETFOCUS message with the *fFocus* parameter set to TRUE).

WinDestroyCursor Destroys the cursor. The cursor is destroyed when the window loses the keyboard focus (receives a WM_SETFOCUS message with the *fFocus* parameter set to FALSE).

WinShowCursor Shows or hides the cursor. The cursor is visible if its show level is zero. Hiding the cursor increments its show level. Showing the cursor decrements the show level. The show level cannot go below zero (so the cursor can be shown an infinite number of times).

WinQueryCursorInfo Fills in a supplied **CURSORINFO** data structure with information about the cursor, including its size, position, and current show level.

Chapter

24

Printing

24.1 Introduction 303

24.2 About Printing 303

- 24.2.1 The Print Queue and the Spooler 304

24.3 Printing 305

- 24.3.1 Specifying the Default Printer 305
- 24.3.2 Opening a Device Context for a Printer 306
  - 24.3.2.1 Logical Address 306
  - 24.3.2.2 Printer-Driver Name 307
  - 24.3.2.3 The Driver-Data Field 307
  - 24.3.2.4 The Data-Type Field 307
  - 24.3.2.5 Using the Print Queue 307
  - 24.3.2.6 Creating a Device Context 308
- 24.3.3 Starting a Print Job 308
- 24.3.4 Associating a Presentation Space 309
- 24.3.5 Drawing for Printing 310
  - 24.3.5.1 Determining Page Size 310
  - 24.3.5.2 Printing a Page 312
  - 24.3.5.3 Finishing a Print Job 312
- 24.3.6 Destroying the Printer Device Context 312

24.4 Special Printing Topics 313

- 24.4.1 Page Setup 313
- 24.4.2 Using a Thread to Print 315
- 24.4.3 Printing to a File 315
- 24.4.4 Printing a Bitmap 316
- 24.4.5 Optimizing Printing for a Particular Printer 317

24.5 Summary 317

24.1 Introduction

This chapter describes how to print graphics and text to printers and plotters. You should also be familiar with the following topics:

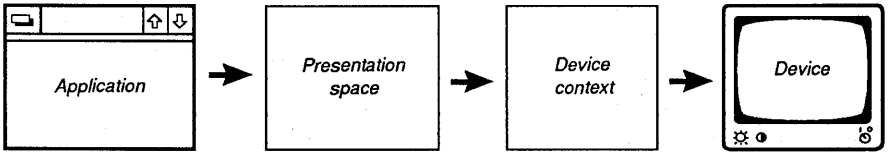
- Standard user-interface guidelines
- Device contexts and presentation spaces
- Window painting
- Graphics programming interface (GPI)
- Extracting information from the *os2.ini* file

24.2 About Printing

The graphics model in MS OS/2 is based on presentation spaces and device contexts. Applications draw in a presentation space by using **Gpi** graphics functions. The presentation space is associated with a device context that translates graphics commands into device-specific operations that display graphics on a device. By associating a presentation space with different device contexts, an application can direct output to different devices, ranging from screen displays to printers and plotters.

The central concept when printing in MS OS/2 is device independence. The same graphics commands used to draw on the screen can be used to draw graphics. For example, a word processor can display its text in a window by calling **Gpi** character and string-drawing functions. When printing, the same application can use the same **Gpi** functions to draw the text, the only difference being that the presentation space is associated with a printer instead of with a window. Printing can be thought of as drawing in a window the size of a sheet of paper. Figure 24.1 shows how output goes through a presentation space to a device context and finally to an output device.

Figure 24.1
Application to Device Path



Choosing the appropriate presentation-space units is an important consideration in achieving device independence for an application. If you use device-independent units, such as LOENGLISH, HIENGLISH, LOMETRIC, or HIMETRIC, your graphics commands will produce nearly the same results on all devices. If you use the PELS unit, you must explicitly allow for the pel size and aspect ratio of the output device.

For example, if you use LOENGLISH units (each unit is 0.01 inch), a call to the **GpiBox** function asking for a 100-unit box produces a 1-inch box on any output device, regardless of the pel size or aspect ratio. In contrast, if you issue the same command using the PELS unit, the box will be one size on the screen (approximately 72 pels per inch) and much smaller on a laser printer (300 pels per inch). Additionally, the box will not be square on an EGA display because its pels are not square.

Of course, there are limits to the device-independence of MS OS/2 graphics, depending on the physical limitations of the output device. Output typically looks better on a laser printer with 300 pels per inch than on a dot-matrix printer with lower resolution. The value of device independence is that you do not have to be concerned with the different capabilities of each device. Instead, images are drawn in a virtual page and the device context makes the best use of the available resolution.

If you use the PELS units, you can do scaling by querying the device context to determine the horizontal and vertical resolution. But it is much better to use a device-independent measurement unit and allow MS OS/2 to scale your drawings to the selected device.

24.2.1 The Print Queue and the Spooler

MS OS/2 provides the means for applications to spool printing jobs so that applications do not wait for printing to finish before proceeding with other processing. The main components of the spooling capability of MS OS/2 are the spooler (*pmspool.exe*) and the queue processor (*pmprint.qpr*). When an application submits a queued printing job, the graphics commands that comprise the print job are output in a device-independent metafile format. The queue processor takes the metafile output and sends it to the printer, translating the contents into printer-specific commands. (The queue processor calls the printer driver to help translate the metafile commands into printer-specific commands.)

The spooler may or may not be involved in this process. If it is active, the spooler manages the metafile output as a spool file and coordinates a queue of spool files waiting to be processed by the queue processor. If the spooler is not active, the metafile is passed directly from the application to the queue processor. The spooler is an optional intermediary between the application and the queue processor.

It is irrelevant to an application whether or not the spooler is active. The user determines if the spooler is running by using Control Panel. An application should always queue its printing output because this takes advantage of the device-independent features of metafiles and the queue processor. If the spooler is active, the queued output will be managed by the spooler. If the spooler is not active, the spooled output will go directly to the queue processor. An application might need to wait longer before printing finishes when the spooler is not active.

24.3 Printing

The following sections describe the typical steps for printing from an MS OS/2 application. The procedures described here allow you to print to the widest range of output devices. Special printing strategies are described later in this chapter. The following topics will be discussed:

- Specifying the default printer
- Opening a device context for a printer
- Starting a print job
- Associating a presentation space for printing
- Drawing the print job in the presentation space
- Finishing a print job
- Destroying the printer device context

24.3.1 Specifying the Default Printer

A user specifies each printer attached to a particular system by making choices in Control Panel that is part of the user shell. A user can install new printer drivers and associate printers with print queues. Information about available printers can be found in the *os2.ini* file. You can access this information by calling the **WinQueryProfileString** function, specifying the appropriate sections and keywords.

An application should not set the default printer directly. Applications should use the printer specified as the default in the *os2.ini* file.

Before using a printer with an application, you should know its driver name and the logical address. To find this information, find the name of the default printer by calling the **WinQueryProfileString** function for the "PM_SPOOLER" section and the "PRINTER" keyword, as shown in the following code fragment:

```
/* Get the default printer name, for example, "PRINTER1." */

cb = WinQueryProfileString(hab,
    "PM_SPOOLER",          /* section name          */
    "PRINTER",             /* keyname               */
    "",                    /* default               */
    szPrinter,             /* profile string        */
    32);                   /* maximum characters    */

szPrinter[cb-2] = 0;       /* remove terminating ";" */
```

The call to the **WinQueryProfileString** function fills the supplied string variable with the name of the installed printer. A typical printer name is "PRINTER1". You use this name of the printer as the keyword specifier and "PM_SPOOLER_PRINTER" as the section name, and then call the **WinQueryProfileString** function again to get a string, called the "printer details," containing several substrings. The substrings contain the name of the printer driver and the name of the logical port that the printer is attached to. The following code fragment shows how to use the **WinQueryProfileString** function to extract this information. The code fragment assumes that the name of the default printer in the variable szPrinter has been filled in by a call to **WinQueryProfileString**.

```
/* Get the printer details.
 *   Fill in a supplied string with substrings:
 *   <physical port>;<driver name>;<queue port>;<network params>;
 *   typically "LPT1;IBM4201;LPT1Q;;"
 */
cb = WinQueryProfileString(hab,
    "PM_SPOOLER_PRINTER", /* section name       */
    szPrinter,            /* keyname            */
    "",                   /* default            */
    szDetails,            /* profile string     */
    256);                 /* maximum characters */
```

Once you extract the long string of substrings from *os2.ini*, you must parse the string to find the substrings. The string contains four substrings: the name of the physical-printer port (for example, LPT1), the name of the printer driver (for example, IBM4201), the name of the logical port (for example, LPT1Q), and network information (if any). Each substring is separated from the next by a semicolon (;), so you can use the library function **strchr** to search for semicolons and return pointers to the positions in the string.

There can be more than one driver name. For example, the string might look like the following:

```
LPT1;IBM4201,PSCRIPT;LPT1Q;;
```

When more than one name is included, the first name is the default name. You should always check for a comma (,) in the driver name to make sure only one name is returned. Control Panel sometimes appends other information to the driver name. You should always parse the driver name and strip off text beginning at the period.

Once you determine the name of the driver and the logical address of the printer, you can open a device context and a presentation space for the printer.

24.3.2 Opening a Device Context for a Printer

Calls to the **WinQueryProfileString** function to produce the name of the installed-printer driver and logical address. Using this information, you can open a device context for a printer. The main role of the device context is to translate graphics calls from an application into device-specific operations. The device context is associated with a presentation space. All drawing in the presentation space is directed to the device rather than to the screen.

You open a device context for a printer by calling the **DevOpenDC** function and passing it a pointer to a **DEVOPENSTRUC** data structure. There are eight pointers in **DEVOPENSTRUC**, but typically only the first four fields must be filled (the logical address, the printer-driver name, driver data, and data type) to open a device context.

24.3.2.1 Logical Address

The logical address of a printer is the destination for the print data. Generally, you use the third substring of the printer-detail string returned by the **WinQueryProfileString** function, as explained earlier in this chapter. You can also print directly to a physical port, such as LPT1, by specifying the name of the physical port as the logical address. You can also supply a filename to direct print output to a file.

24.3.2.2 Printer-Driver Name

The printer-driver name identifies the driver that controls the printing device. The printer-driver name is usually extracted from the printer-detail string in the *os2.ini* file, as explained earlier in this chapter. MS OS/2 adds a filename extension (*.drv*) to the name you supply.

24.3.2.3 The Driver-Data Field

The driver-data field points to a printer-specific data structure that describes aspects of the page, such as the page layout (portrait or landscape) and the default data format (PM_Q_STD or PM_Q_RAW). If you set this field to NULL, the printer driver uses the default settings established by the user when the printer driver was installed with Control Panel. The user can also use Control Panel at any time to change printer settings.

If NULL is passed for the driver-data pointer, the settings most recently set in Control Panel will be used. Because Control Panel is always available, it is not necessary for an application to provide the means to change these settings. However, it is possible to change the driver data for a particular print job from an application by calling the **DevPostDeviceModes** function.

24.3.2.4 The Data-Type Field

The data-type field is a string that specifies the print-data format. The two possibilities supported by MS OS/2 version 1.1 are PM_Q_STD and PM_Q_RAW. You can supply NULL for this field to obtain the default data type for the printer.

- The PM_Q_STD format contains data generated by **Gpi** graphics calls, including graphics calls to draw text. This format is generally used and is the most versatile and printer-independent.
- The PM_Q_RAW format contains a printer-specific data stream generated by the application rather than by the translated graphics commands of the PM_Q_STD format. You should use the PM_Q_RAW format only if you know the exact capabilities of the printer. For example, this format might be useful for an application that produces its own PostScript output directly rather than relying on **Gpi** commands to be translated by the device context. It might also be useful for sending a text stream to a printer that does not support graphics. If you use the PM_Q_RAW format, you can send the data to the printer by calling **DevEscape** with the DEVESC_RAWDATA control code.

24.3.2.5 Using the Print Queue

One of the arguments to the **DevOpenDC** function specifies the type of device context to open. The possibilities that are applicable to printing-device contexts are OD_QUEUED, OD_DIRECT, and OD_INFO. Generally, OD_QUEUED is used to take advantage of the spooling capabilities of MS OS/2. If a user has the spooler turned on, graphics calls are captured as a spool file and placed in the print queue. The spooler passes the spool file to the queue processor for actual printing. Once a queued file has been written to its spool file, the application is free to continue with other tasks. This means that a user can continue working without waiting for a document to print.

Even if the spooler is turned off, you should use a OD_QUEUED device context. If the spooler is absent, printing data is passed directly from the application to the queue processor where it is processed for final output to the device. It does not matter if the spooler is running or not. You should always queue your printing output, except when you want to bypass the print queue or when you want to open a device context for information only. You can use OD_DIRECT when you want to bypass the print queue, such as when printing to a file.

You can use OD_INFO to open a device context for information only, such as during program initialization when determining the page size of the current printer. Knowing the page size of the printer can be used to provide on-screen pagination information. However, an application should always check the page dimensions before printing because the user may have changed the default printer.

24.3.2.6 Creating a Device Context

Once the device-specific information described earlier in this chapter is obtained, you can open the device context. In the **DEVOPENSTRUC** structure, the first three fields point to the driver name, the logical-address name, and the **DRIVDATA** structure, respectively. The device-context type is set to OD_QUEUED so that the output will be queued. The following code fragment shows how to open a device context, assuming that the printer-driver name and the logical-address name have been obtained:

```
/*
 * Fill in the DEVOPENSTRUC structure
 *
 * The name of the driver and logical address were obtained by calling
 * WinQueryProfileString. The driver data came from DevPostDeviceModes.
 */


DEVOPENSTRUC dop;
PSZ          pszDriverName;
PSZ          pszLogAddress;

/*
 * Use WinQueryProfileString to fill in pszDriverName
 * and pszLogAddress.
 */

dop.pszDriverName = pszDriverName; /* from os2.ini */
dop.pszLogAddress = pszLogAddress; /* from os2.ini */
dop.pdriv = NULL;
dop.pszDataType = "PM_Q_STD";

/* Now open the device context. */
hdcPrinter = DevOpenDC(hab,
                       OD_QUEUED,
                       "*",
                       4L,        /* use first four fields */
                       (PDEVOPENDATA) &dop,
                       (HDC) NULL);
```

24.3.3 Starting a Print Job

Once a device context is successfully opened using a printer driver, a "start doc" message must be issued to the device context to tell it a new document is starting. The start-doc escape call includes a string that is displayed by the queue manager as the print-job name. Typically, a filename is supplied as the document name, as shown in the following code fragment:

```
LONG lrc;

/* Start a document. */

lrc = DevEscape(hdcPrinter,
                DEVESC_STARTDOC,
                strlen(pszDocName),
                pszDocName,
                NULL,
                NULL);
```

24.3.4 Associating a Presentation Space

The device-context handle returned by the **DevOpenDC** function is used to create a presentation space for drawing. You specify zero for the *x*- and *y*-dimensions of the presentation space so that the system can make the presentation space large enough to include a single page using the specified device context. You cannot use a cached-micro presentation space for printing. If you use an absolute-measurement environment—for example, PU_LOENGLISH or PU_HIMETRIC, graphics will automatically be the same size on screen and when printed. If you use PU_PELS, you must scale the graphics commands to match the different resolutions of the screen and the printer.

```
SIZEL sizl;

sizl.cx = 0L;
sizl.cy = 0L;

hpsPrinter = GpiCreatePS(hab, hdcPrinter, &sizl,
    PU_LOENGLISH | GPIF_DEFAULT | GPIT_NORMAL | GPIA_ASSOC);
```

Another strategy is to use for printing the same presentation space used for window painting. This saves creating a new presentation space. In this case, the presentation space must be a GPIT_NORMAL presentation space. The presentation space must first be disassociated from the device context that it has been associated with (usually a window device context), and then associated with the printer device context, as shown in the following code fragment:

```
GpiAssociate(hpsWindow, NULL);          /* disassociate first     */
     .
     .
     .
GpiAssociate(hpsWindow, hdcPrinter);  /* associate with printer */
```

You must be cautious of the WM_PAINT message during printing operations if you use the same presentation space for printing and window drawing. The presentation space cannot be used to respond to the paint message while the presentation space is associated with the printer device context. For more information, see Section 24.4.2.

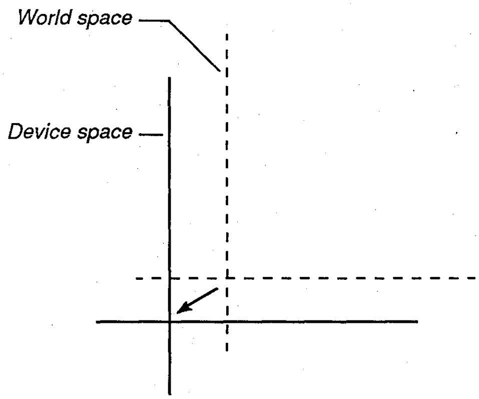
Once a printer device context has been associated with a presentation space, graphics functions can be called to draw each document page.

24.3.5 Drawing for Printing

Application data images are usually expressed in "world space." This is the coordinate system in which all graphics commands are expressed. The units of the world space are typically application-convenient units, such as 0.01-inch (LOENGLISH). Images are drawn in windows by expressing graphics commands in convenient units. The system scales and converts these units to the device-specific units of the display so that the image appears correct in the window.

If an image is larger than a window, a matrix translation is used to slide the coordinate system of the world space to match the coordinate system of the window. Figure 24.2 illustrates how a translation might be used to map a portion of the world space to a window display space.

Figure 24.2
World-Space to Device-Space Translation



The central idea behind drawing for printing is that it is very similar to drawing for a window that is the size of a piece of paper. The system performs the same scaling transformation to convert world-space units to device units. This automatic scaling allows the same imaging code to draw for both a 72-dot-per-inch impact printer and a 300-dot-per-inch laser printer.

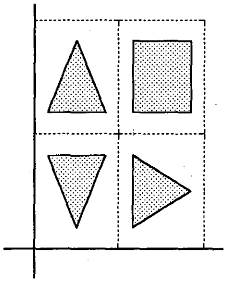
24.3.5.1 Determining Page Size

In order to print images to a printer or plotter, the page size must be expressed in convenient units. For example, if you do all your drawing in 0.01-inch (LOENGLISH) units, you will need the page size in 0.01-inch units rather than in device units (PELS). To determine the page size, call the **DevQueryCaps** function for the given device to get the width and height of the device page in device units. These units can be converted to presentation-space units by calling the **GpiConvert** function, as shown in the following code fragment:

```
SIZEL sizl;
/* page size */
DevQueryCaps(hdcPrinter, CAPS_WIDTH, 2L, (PLONG) &sizl);
GpiConvert(hpsWindow, CVTC_DEVICE, CVTC_WORLD, 1L, (PPOINTL) &sizl);
```

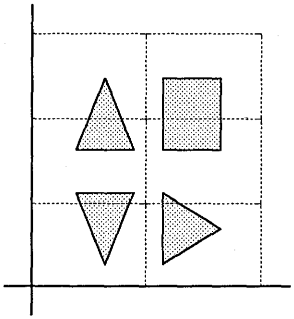
Once the page size has been determined, the image can be divided and drawn into pages for printing. Figure 24.3 shows how a multiple-page document might be broken into pages, assuming a user has selected portrait mode during printer setup. The figure shows how the page size is used to divide the world space into pages.

Figure 24.3
Portrait-Mode Pages in World Space



The page size returned reflects either portrait or landscape mode, depending on what the user selects in Control Panel. Figure 24.4 shows how these same pages are divided for landscape mode. Notice that no additional page rotation or extra support work must be done for landscape mode. As long as you use the *x*- and *y*-sizes returned by the **DevQueryCaps** and **GpiConvert** functions, a document will be correctly paginated.

Figure 24.4
Landscape-Mode Pages in World Space



24.3.5.2 Printing a Page

Each page of a print job is drawn by making **Gpi** calls to the world space of the presentation space. If a document consists of only one page, then the origin of the world space probably corresponds to the origin of the device page, so no translation is necessary. To print a multi-page document, you must translate world-space coordinates so that they line up each page with the origin before drawing.

For example, to print page 2 in Figure 24.4, you would apply a translation on the *x*- and *y*-axis to slide the desired page so that its lower-left corner sits at the origin. Assuming that a **SIZEL** structure holds the horizontal and vertical dimensions of the page, a call to the **GpiSetDefaultViewMatrix** function slides page 2 down and over to the origin. Notice that first you call the **GpiQueryDefaultViewMatrix** function to fill in the other values of the nine-element matrix, then the *x*- and *y*-translation fields are filled in, then the matrix is set.

```
MATRIXLF matrix;
SIZEL    sizl;   /* size of page, in world units */

/* First get the current transformation matrix. */

GpiQueryDefaultViewMatrix(hps, 9, &matrix);

/* Change the x- and y-translation elements. */

matrix.lM31 = -sizl.cx;
matrix.lM32 = -sizl.cy;

/* Call GpiSetDefaultViewMatrix to translate image to page. */

GpiSetDefaultViewMatrix(hps, 9, &matrix, TRANSFORM_REPLACE);
```

For a multi-page document, you must call the **DevEscape** function with the DEVESC_NEWFRAME escape at the end of each page to begin a new page. It is not necessary to send a DEVESC_NEWFRAME escape after the last page.

24.3.5.3 Finishing a Print Job

After printing all pages in a print job, you must call the **DevEscape** function with the DEVESC_ENDDOC code telling the queue processor that the print job is finished.

24.3.6 Destroying the Printer Device Context

Once drawing is finished, the presentation space should be disassociated from the printer device context. If a special presentation space was created for printing, it can be destroyed along with the device context, as shown in the following code fragment:

```
GpiAssociate(hpsPrinter, NULL);
DevCloseDC(hdcPrinter);
GpiDestroyPS(hpsPrinter);
```

However, if an existing presentation space was used for printing, you should disassociate the presentation space from the printing device context, reassociating the presentation space with its original window device context.

24.4 Special Printing Topics

The following sections describe special printing techniques.

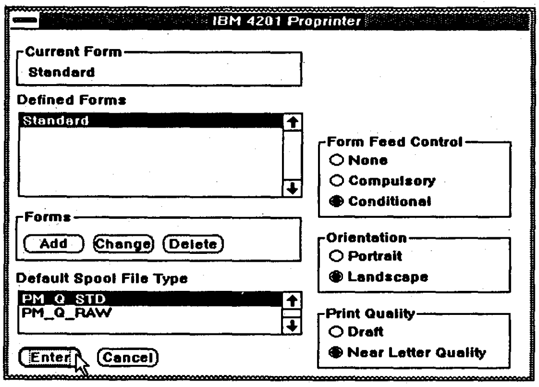
24.4.1 Page Setup

One of the fields in the **DEVOPENSTRUC** structure, used when opening a device context, is **pDriverData**, a pointer to a **DRIVDATA** structure. This structure contains device-specific information about the page configuration (landscape or portrait), page size, default font, and other page-configuration details. The size and contents of the **DRIVDATA** structure depend on the printer being used. Typically, a user specifies the default configuration information from Control Panel. Control Panel stores this information for each printer in the *os2.ini* file. If NULL is supplied for the driver data when opening the device context, the printer driver retrieves the system-wide default driver data from *os2.ini* and uses it to configure page characteristics.

An application can always use the page configuration specified by the user in Control Panel. If more control is required, calling the **DevPostDeviceModes** function displays a dialog window that can be used to specify new page characteristics for individual print jobs. Driver data retrieved by this dialog window applies to the current print job. It does not change the system default settings.

Figure 24.5 shows a dialog window that fills in the **DRIVDATA** structure for an IBM4201 printer.

Figure 24.5
IBM4201 Page Setup



IBM 4201 Proprinter

Current Form: Standard

Defined Forms: Standard

Form Feed Control
○ None
○ Compulsory
● Conditional

Orientation
○ Portrait
● Landscape

Default Spool File Type: PM_Q_STD, PM_Q_RAW

Print Quality
○ Draft
● Near Letter Quality

The **DevPostDeviceModes** function displays the page-setup dialog window. Any changes in page setup are reflected in the **DRIVDATA** structure and passed to the **DevOpenDC** function to obtain a device context matching the page characteristics.

The following code fragment shows how to call the **DevPostDeviceModes** function to fill the **DRIVDATA** structure. You call it first with a NULL structure pointer to determine the size of the device-specific structure. Using the size value returned, it is possible to allocate memory and pass a far pointer to the **DevPostDeviceModes** function, which in turn fills the device information in the structure.

```
ULONG  ulSize;
PDRIVDATA pdriv;

ulSize = DevPostDeviceModes(hab,
    NULL,            /* NULL for data size only */
    pszDriverName,   /* driver name             */
    NULL,            /* device name             */
    szPrinter,       /* printer name            */
    OL);             /* not used here           */

/* Now allocate some memory. */

usResult = DosAllocSeg(ulSize, &sel, O);

pdriv = MAKEP(sel, O);

ulSize = DevPostDeviceModes(hab,
    pdriv,                       /* buffer for drivdata */
    pszDriverName,               /* driver name         */
    NULL,                        /* device name         */
    szPrinter,                   /* printer name        */
    OL);      /* Display dialog. Do not change os2.ini. */
```

Notice that the last argument to the **DevPostDeviceModes** function is a long word that determines whether or not the function changes the contents of the *os2.ini* file. The following values are valid:

Value	Meaning
0	Displays a dialog window for driver data without changing the defaults in the *os2.ini* file. This is appropriate for an application that configures a single print job without changing system-wide settings.
1	Displays a dialog window for driver data and writes the new driver data, which becomes the default, to the *os2.ini* file. This is appropriate for Control Panel because the new default settings affect the entire system. Generally, applications should not change the driver data for the entire system.
2	Does not display a dialog window; returns the default-driver data from the *os2.ini* file. This is useful for saving default-driver data with a particular document.

If you use the **DevPostDeviceModes** function to set up the driver data for an individual document, you should save the driver data along with the document so that the settings are available to the user when the document is reopened. Be sure the printer driver is the same as it was when the driver data was created because the data format is unique to each printer driver.

24.4.2 Using a Thread to Print

An MS OS/2 application should not be unresponsive to user input for an extended period of time. Any operation that takes longer than 1 second should be carried out in a separate thread, allowing the user-interface thread to continue to respond to user input.

Printing typically begins when a user chooses a command from a menu in an application. A client window receives a WM_COMMAND message from the menu and begins the printing operation. No further mouse clicks or keystrokes are processed until the application calls the **WinGetMsg** or **WinPeekMsg** function again. Therefore, the user cannot interact with an application menu or switch to another application until one of these two functions is called.

The following are two basic ways to allow an application to remain responsive during a lengthy printing operation:

- Call the **WinGetMsg** or **WinPeekMsg** function at regular intervals during printing to handle user input.
- Create a separate thread to handle printing. The main thread can continue to call the **WinGetMsg** function while the printing thread executes.

Handling user input during printing can cause many problems with data integrity and synchronization. For example, should a user be allowed to modify a document while it is printing? Semaphores should be used to protect shared resources whenever there are potential conflicts.

One simple solution to the data synchronization problem is to create a thread for printing and display a message box in the main thread that allows a user to cancel printing. The printing thread could periodically check a semaphore controlled by a message box to determine if printing has been cancelled. This way a user cannot modify data in an application while it is printing, but can switch to another application while printing continues.

Another possible problem when handling messages during printing is receiving WM_PAINT messages. Because printing and window drawing typically use the same drawing functions and the same data, drawing routines must be reentrant. If the same presentation space is used for window painting and printing, an application should not draw in the window when the presentation space is associated with the printer device context.

24.4.3 Printing to a File

Some printer drivers allow data to be sent to a file rather than to a device. In particular, a PostScript device driver and a plotter device driver direct print data to files if a filename is supplied in the **pszLogAddress** field of the **DEVOPENSTRUC** structure when opening a device context for printing. All output from printing is sent to this file. If the file already exists, its data is overwritten. If the file does not exist, the device driver creates it. These two device drivers support this feature because their data stream is ASCII text. The IBM4201 device driver, which sends mostly binary data to the printer, does not support printing to a file.

You should open an OD_DIRECT device context when printing to a file.

24.4.4 Printing a Bitmap

Printing a bitmap requires that the bitmap be converted from its original format, which is usually compatible with the video display, to a format for the current printing device. The **GpiSetBitmap** function converts bitmaps from one device format to another.

You must perform the following steps to print a bitmap on a printer:

1 Create a device context and a presentation space for the screen.

2 Create a memory device context for the screen and associate it with a presentation space.

3 Create a bitmap and attach it to the memory presentation space and device context using the **GpiSetBitmap** function.

4 Draw the bitmap from the screen presentation space and device context to the memory presentation space and device context using the **GpiBitBlt** or **GpiWCBitBlt** function.

5 Create a device context and presentation space for the printer.

6 Create a memory device context for the printer and associate it with a presentation space.

7 Convert the bitmap from the display-memory presentation space and device context to the format of the printer-memory presentation space and device context.

8 Attach it to the printer-memory presentation space and device context by calling the **GpiSetBitmap** function.

9 Draw the bitmap from the printer-memory presentation space and device context to the printer presentation space and device context by using the **GpiBitBlt** or **GpiWCBitBlt** function. Do any scaling necessary to correct for resolution differences between the display and the printer.

The **GpiSetBitmap** function converts the different device formats, such as color to monochrome, but does not correct for differences in pel resolution between devices. For example, if you are printing a screen bitmap (typically about 72 pels per inch to a laser printer with 300 pels per inch, you must scale the image when converting from the printer-memory device context to the device context. You can determine the differences in pel resolution by calling the **DevQueryCaps** function for each device.

24.4.5 Optimizing Printing for a Particular Printer

You can optimize printing for certain printers not usually available through the API by using the **DevEscape** function. Escape calls are sent to the device driver, which must interpret them. The **DevEscape** function signals the beginning and end of documents of single pages, and sends printer-specific binary data to a printer.

24.5 Summary

The following functions can be used in setting up a printing operation:

DevCloseDC Closes a device context opened by the **DevOpenDC** function.

DevEscape Sends escape codes to the specified device. Starts and stops printing, signals the end of a page, and sends printer-specific binary data to a printer.

DevOpenDC Opens a device context for a printing device.

DevPostDeviceModes Displays a dialog window for a user to specify various printer configuration information.

DevQueryCaps Returns information about a specific device, such as the width and height of a page.

GpiAssociate Associates a presentation space with a device context for a particular printer, or disassociates the presentation space from its current device context if a NULL device context is specified.

GpiConvert Converts device units to world coordinates from one coordinate system to another.

GpiCreatePS Creates a presentation space that can be associated with a printing-device context.

GpiDestroyPS Destroys a presentation space created by the **GpiCreatePS** function.

GpiQueryDefaultViewMatrix Queries the current viewing transformation matrix and fills in a supplied **MATRIXL** structure. This function retrieves the current viewing transformation matrix so that the *x*- and *y*-translation elements can be changed.

GpiSetBitmap Connects a bitmap with a presentation space that is associated with a memory device context, converting a bitmap from one device format to another, such as converting between the screen and a printer.

GpiSetDefaultViewMatrix Sets the viewing transformation matrix so that the *x*- and *y*-translation values can be used to move portions of a world-space image to the origin for printing.

WinQueryProfileString Retrieves the name of the current default printer and printer driver from the *os2.ini* file.

Chapter

25

Heaps

25.1 Introduction 321

25.2 About Heaps 321

25.3 Using a Heap in an Application 322

25.3.1 Creating a Heap 322

25.3.2 Heaps in a Separate Data Segment 323

25.3.3 Moveable Heap Objects 324

25.3.3.1 Moveable Heaps in an Automatic Data Segment 326

25.3.3.2 Moveable Heaps in a Separate Data Segment ... 326

25.3.4 Allocating Memory from Heaps 328

25.3.5 Deallocating Memory from a Heap 328

25.3.6 Using Dedicated Free Lists 329

25.3.7 Destroying Heaps 329

25.4 Summary 330

25.1 Introduction

This chapter describes the functions that allow you to use heaps for memory management in your applications. You should also be familiar with the following topic:

- Memory management in MS OS/2

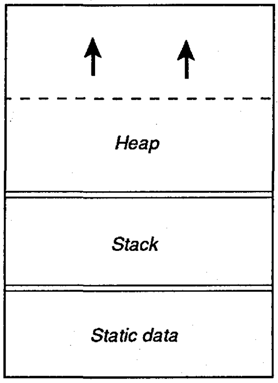
25.2 About Heaps

A heap is a memory segment containing other memory-block objects that are allocated and deallocated by the functions of the heap manager (the group of functions that manage heaps in MS OS/2). The heap functions are provided to supplement, and in some cases replace, the basic memory-management functions of MS OS/2. The heap functions provide more functionality than the basic memory-management functions, including moveable objects within a segment and faster allocation implementation.

A heap exists within a memory segment. The segment can be the automatic data segment of an application or dynamic-link module, or it can be another segment that has been allocated explicitly by using the **DosAllocSeg** function. Typically, the heap is part of an automatic data segment, and it shares that segment with the application's static data and stack. The combined size of the heap, static data, and stack cannot be larger than 64K—this is the maximum segment size in MS OS/2. Heaps in separate segments also cannot be larger than 64K.

Figure 25.1 shows how a heap can share an automatic data segment with the static data and stack:

Figure 25.1
Heap in Automatic Data Segment



A heap typically contains many memory-allocation objects. Each object is accessed by an offset (near pointer) from the beginning of the segment. Notice that the beginning of the heap is not necessarily at the beginning of the segment. For heaps in the automatic data segment, the application or dynamic-link module can use the near pointer to the memory object directly, because the data-segment selector is implicit. For heaps allocated in separate segments, the near pointer must be combined with the segment selector to make a far pointer.

A heap can be created so that objects within the heap are moveable. This allows the system to rearrange objects on the heap to make more free memory available and avoid heap fragmentation.

MS OS/2 attempts to make the heap larger if a memory-allocation request cannot be filled by using the existing heap. This growth is controlled by setting growth limits when the heap is created.

25.3 Using a Heap in an Application

Applications typically create a heap, allocate and deallocate memory blocks in the heap as needed, and destroy the heap when terminating. A heap can be created using many different memory sources and it can have moveable or nonmoveable memory objects. The following sections discuss how to use heaps in applications and dynamic-link libraries.

25.3.1 Creating a Heap

Applications and dynamic-link libraries create a heap by calling the **WinCreateHeap** function. The heap is created within an automatic data segment or in a separate segment, depending on the values of the *selHeapBase* and *cbHeap* parameters of **WinCreateHeap**. The possible values of these parameters are summarized in the following list:

selHeapBase	cbHeap	Meaning
Zero	Zero	The calling process is an application that places the heap at the end of its automatic data segment.
Selector	Nonzero	The calling process is either a dynamic-link library that places a heap at the end of its automatic data segment, or an application or dynamic-link library that has explicitly allocated a segment and places the heap at the end of the segment.
Selector	Zero	The calling process is either an application or dynamic-link library that has explicitly allocated a segment and places a heap in that segment.
Zero	Nonzero	The calling process is either an application or dynamic-link library that places a heap of a specific size in a separate segment but does not call the **DosAllocSeg** function.

In addition to the characteristics of the heap described by the preceding list, the creator of the heap may specify whether the heap contains moveable objects and whether the functions should check the validity of certain arguments to heap-manager functions. The HM_MOVEABLE attribute specifies that the heap can contain moveable objects. The HM_VALIDSIZE attribute, which can be used only in conjunction with HM_MOVEABLE, specifies that the heap manager should check the validity of size arguments in heap-deallocation calls. Moveable heap objects allow a more flexible memory-management scheme for applications with heavy memory requirements.

You must specify the minimum amount the heap will grow each time it enlarges to satisfy a memory request. The default minimum is 512 bytes.

The *cbMinDed* and *cbMaxDed* parameters of the **WinCreateHeap** function define how many dedicated free lists the heap manager should maintain for the heap. Dedicated free lists can make the allocation of fixed-size blocks significantly faster, but they are not essential to the operation of the heap. Zeros can be passed as values for these parameters to generate the default heap behavior without using dedicated free lists. More information about dedicated free lists is given later in this chapter.

For more information on the **WinCreateHeap** function, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

The following code fragment shows how to create a heap with the default behavior and the moveable attribute:

```
hHeap = WinCreateHeap(0,  /* uses automatic data segment         */
    0,                    /* uses HEAPSIZE from .def file        */
    1024,                 /* minimum size to grow heap           */
    0,                    /* minimum size of dedicated free list */
    0,                    /* maximum size of dedicated free list */
    HM_MOVEABLE);
```

The ability to share a heap depends upon the sharing attributes of the segment containing the heap. Heaps in an application's data segment are private to that application. Segments explicitly allocated with the **DosAllocSeg** function are shared or private, depending upon the value of the *fsAlloc* parameter. Segments allocated by using the **WinCreateHeap** function are shareable. Because shared segments cannot shrink, heaps within a shared segment also do not shrink.

The heap manager does not prevent multiple threads from calling the heap manager with the same heap handle. The calling process must ensure that this does not occur.

25.3.2 Heaps in a Separate Data Segment

One of the options you can specify when creating a heap is that the heap will be created in a data segment that is separate from the automatic data segment. You might choose this option if there is insufficient space in the automatic data segment for the static variables, the stack, and the heap. A heap in a separate data segment can occupy up to 64K.

The pointer returned by the **WinAllocMem** function is an offset value that locates the allocated memory block relative to the beginning of the segment that contains the heap. If the heap is in a separate segment, you must use far pointers to access memory blocks that are allocated on the heap. You can determine the far pointer to a heap object by using the heap's segment selector and the offset. The **WinLockHeap** function returns a far pointer to the beginning of a specified heap. You can combine the selector from this far pointer with the offset to a memory block on the heap to produce a far pointer to the heap object, as shown in the following code fragment:

```
HHEAP   hHeap;
SEL     selHeap;
NPBYTE  npbObject;
PBYTE   pbObject;
PVOID   pvHeap;

/* Allocate a heap in a separate segment. */

hHeap = WinCreateHeap(0,  /* uses a separate segment           */
    32*1024,              /* allocates 32K for heap            */
    1024,                 /* minimum size to grow heap         */
    0,                    /* minimum size of dedicated free list */
    0,                    /* maximum size of dedicated free list */
    HM_MOVEABLE);

/* Allocate an object and retrieve a near pointer. */

npbObject = WinAllocMem(hHeap,...);

/* Retrieve a far pointer to the start of the heap. */

pvHeap = WinLockHeap(hHeap);

/* Make a far pointer to the heap object. */

pbObject = MAKEP(SELECTOROF(pvHeap), npbObject);
```

25.3.3 Moveable Heap Objects

A moveable heap allows the memory objects within the heap to move in order to reclaim fragmented heap space. All heaps are moveable in the sense that a segment that contains a heap can move as a result of the mapping of selectors to physical addresses that is provided by MS OS/2. Moveable heaps differ from regular heaps in that individual objects within a movable heap can change their positions relative to the beginning of the segment. The moveable-heap attribute is specified when the heap is created and lasts until the heap is destroyed.

Allocated memory blocks in a moveable heap have a header structure that is attached to the beginning of the block. This header structure contains a pointer to the variable holding the pointer to the memory block and a field containing the size of the block (not including the header structure). The near pointer returned by the **WinAllocMem** function points to the first byte after the block-header words. The C definition of the header structure is as follows:

```
typedef struct _MOVBLOCKHDR {
    NPBYTE  *ppmem;
    USHORT  cb;
} MOVBLOCKHDR;
```

The size parameter of the **WinReallocMem** and **WinFreeMem** functions is ignored for objects in a moveable heap and the value of the size word is used instead. However, if the HM_VALIDSIZE option is specified in the **WinCreateHeap** function when the heap is created, the **WinReallocMem** and **WinFreeMem** functions verify that the passed size matches the current size and return an error if it does not.

Objects in a moveable heap can move whenever the **WinAvailMem** function is called. Because this function is also called by **WinAllocMem** and **WinReallocMem**, objects can also move when these functions are called. **WinReallocMem** and **WinAvailMem** move blocks that have a back pointer. Allocated objects whose back pointer is zero are considered fixed and do not move.

When allocating memory blocks within a moveable heap, the calling process specifies that the block is moveable by altering the back pointer (**ppmem**) field of the header structure so that it points to the variable holding the pointer returned by the **WinAllocMem** function. When **WinAllocMem** creates a block on a moveable heap, it clears the back pointer to zero. As long as the back pointer remains zero, the heap manager cannot move the block. If the application alters the value of the back pointer so that it points to a valid variable address within the same segment (by using an offset from the beginning of the segment), the heap manager will move the block, when necessary, to compact the heap. Whenever the heap manager moves the moveable block it also updates the variable pointed to by the back pointer so that the variable points to the new location of the block. The back pointer ensures that the application's reference to the moveable block is updated when the block moves.

Note that MS OS/2 alters only the variable pointed to by the back pointer when it moves a moveable block. If the application has made copies of this variable, those copies will be invalid if the memory object moves.

The **SETMEMBACKPTR** macro sets the back pointer of a moveable block. (It is assumed that the **MOVBLOCKHDR** data structure described above is also defined.) **SETMEMBACKPTR** uses the variable that holds the pointer returned by the **WinAllocMem** function and sets the back pointer of that block to point to the variable. The **SETMEMPACKPTR** macro is shown below. Note that it should not be used on nonmoveable heaps.

```
#define SETMEMBACKPTR(npb) (((PMOVBLOCKHDR) npb) -1) -> ppmem = &npb
```

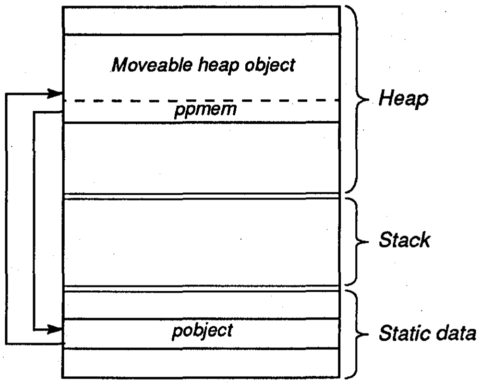
The back pointer of a moveable block should point to the variable that holds the pointer returned by the **WinAllocMem** function. Since the back pointer is a near pointer, the variable pointed to must be in the same data segment as the heap. If the heap is in the automatic data segment (the default case), you can use a static or stack-based variable to hold the pointer. If the heap is in a separate data segment, you must allocate space for the pointer variable as a nonmoveable block on the heap.

Using moveable blocks allows an application to use memory more efficiently and to avoid most memory-fragmentation problems. Using moveable heap objects, however, requires that the pointer references to the objects remain valid even when the objects move. The back pointer allows MS OS/2 to handle updating an application's pointer variables, but the application must use the macro **SETMEMBACKPTR** to set the original link between the moveable block and its pointer variable.

25.3.3.1 Moveable Heaps in an Automatic Data Segment

Figure 25.2 shows the relationship of the pointer returned by the **WinAllocMem** function and the back pointer of the memory-block header when the heap is in the automatic data segment:

Figure 25.2
Back Pointer for Moveable Heap Object
in Automatic Data Segment



The following code fragment shows how to allocate a block and then make it moveable by altering the value of the back pointer. This code works only if the heap is in the automatic data segment (the default case for most applications).

```
static NPBYTE  pObject;

/* Allocate the block for the object. */

pObject = WinAllocMem(hHeap, sizeof(YOUR_OBJECT_TYPE));

/* Make the block moveable. */

SETMEMBACKPTR(pObject);
```

You should avoid placing a pointer to a moveable heap object on the stack—that is, making it a local variable—if the heap object will continue to exist after the function has ended and the local variable has been cleared from the stack. This is dangerous because the heap manager could attempt to update the pointer variable, using the back pointer, and inadvertently write into the stack frame of another function.

25.3.3.2 Moveable Heaps in a Separate Data Segment

The variable pointed to by the back pointer must be in the same segment as the moveable block. For most applications, in which the heap is in the application's automatic data segment, the pointer variable can be in the application's static-data area. If the heap is in a separate segment, the variable must also be allocated on the same heap, and it must be in a nonmoveable block.

Figure 25.3 shows the relationship of the pointer returned by the **WinAllocMem** function and the back pointer of the memory-block header when the heap is in a separate data segment:

Figure 25.3
Back Pointer for Moveable Heap Object in Separate Data Segment

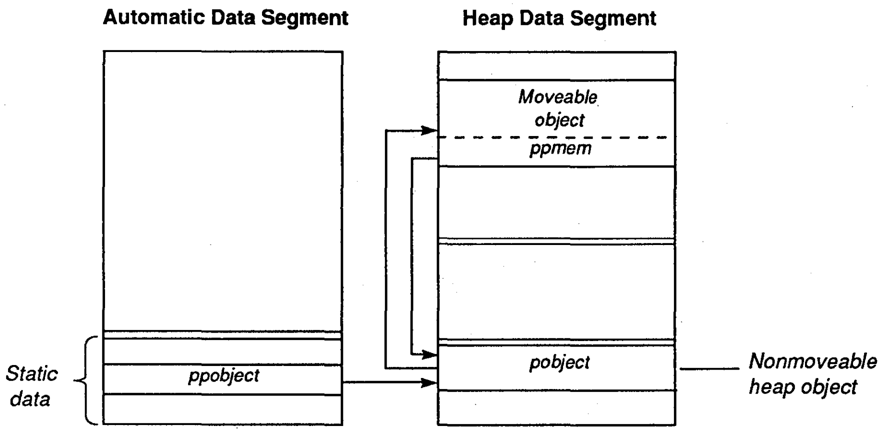


Figure 25.3 shows a variable in the application's static-data area that points to the nonmoveable block containing the pointer to the moveable memory block. The variable that the application uses in its static-data area is a pointer to a pointer. Since the pointer to a pointer is pointing to another data segment, it must be a far pointer, consisting of the selector of the heap's data segment and the offset within that segment.

The following code fragment shows how to allocate a block in a heap in a separate data segment and then make the block moveable by altering the value of the back pointer. The *ppObject* variable is declared with the **static** storage class so that it will be in the static area of the automatic data segment, rather than on the stack.

```
static NPBYTE FAR *ppObject;

/*
 * Allocate nonmoveable space for the pointer to the object
 * and make a far pointer to the pointer.
 */

ppObject = MAKEP(SELECTOROF(WinLockHeap(hHeap)),
                            WinAllocMem(hHeap, sizeof(NPBYTE)));

/* Allocate the block for the object. */

*ppObject = WinAllocMem(hHeap, sizeof(YOUR_OBJECT_TYPE));

/* Make the block moveable. */

SETMEMBACKPTR(*ppObject);
```

Once the pointer to a pointer is set up correctly and the back pointer is initialized, the *ppObject* variable should be dereferenced twice whenever the moveable memory block is accessed, as shown in the following example:

```
/* Put a value into the moveable block. */

**ppObject = 2;
```

25.3.4 Allocating Memory from Heaps

Once an application or dynamic-link library has created a heap, it can allocate blocks on the heap by calling the **WinAllocMem** function. The value returned by this function is a near pointer to the memory block, or NULL if the function is unsuccessful.

All pointers to memory objects within a heap are 16-bit offsets from the start of the heap's segment. All memory objects in the heap are aligned on a 32-bit-word boundary—this means that the contents of the 2 low-order bits of a returned pointer are unused. The **WinAllocMem** function clears these bits. (The **WinReallocMem** and **WinFreeMem** functions require that they be zero.) The application can use these two bits for any purpose. The HEAP_MASK constant can be used to clear the bits when passing a parameter for a memory-block pointer to **WinReallocMem** or **WinFreeMem.**

If the heap is created with the HM_MOVEABLE attribute, the size argument for the allocation is retained in the size word of the allocated block's header. The returned address is the address of the first byte after the header.

The heap manager searches the heap for the first free block large enough to fulfill the allocation request. If the free block that is found is larger than what is needed to satisfy the request, the extra space is added to the appropriate dedicated free list. If no free block is found that is large enough, the heap manager attempts to combine free blocks by calling the **WinAvailMem** function. If this call does not generate a large enough free block, the heap manager attempts to enlarge the heap segment by the combined values of the size of the request and the minimum-growth parameter specified in the call to the **WinCreateHeap** function. If this attempt fails, the **WinAllocMem** function returns NULL, indicating that it could not allocate the memory block.

25.3.5 Deallocating Memory from a Heap

Memory blocks on the heap can be deallocated by calling the **WinFreeMem** function. The calling process must specify the heap handle, the pointer to the block, and the size of the block. The size argument must be accurate because the heap manager does not normally validate this argument. Passing an incorrect size argument to **WinFreeMem** can damage other blocks on the heap.

The **WinFreeMem** function returns NULL if it successfully deallocates the memory block. The return value is NULL for success because of the following idiom for deallocating a memory block and invalidating the variable that contains the pointer to the block, all in one line of code:

```
pMem = WinAllocMem(...);

/* Code that uses the block. */

pMem = WinFreeMem(pMem);
```

For nonmoveable heaps, the heap manager has no way to check the size of blocks that it deallocates. For moveable blocks on a heap created with the HM_MOVEABLE and HM_VALIDSIZE attributes, the heap manager checks the size argument against the size specification in the moveable block's header and returns the pointer (instead of NULL) if the size parameter is invalid.

25.3.6 Using Dedicated Free Lists

A dedicated free list is a linked list of free blocks of a particular size on the heap. For example, the heap manager might maintain a dedicated free list of memory blocks that are 1024 bytes in length. It is much faster to search for a memory block in a dedicated free list than to do a straight linear search of all blocks on the heap. Thus, dedicated free lists are very useful if your application allocates many blocks with the same size.

The size of memory blocks that should be maintained in dedicated free lists is specified when the heap is created. Two arguments to the **WinCreateHeap** function specify the minimum block size and the maximum block size to put into dedicated free lists. All memory sizes between the minimum and maximum sizes, in four-byte increments, are maintained in separate lists.

For example, if you specify 1024 for the minimum size and 2048 for the maximum size, the heap manager creates dedicated free lists for memory blocks of 1024 bytes, 1028 bytes, 1032 bytes, and so on, through 2048 bytes. The cost of each dedicated free list is an additional two bytes in the heap-control block for each size of memory block that is maintained in the list.

Blocks that are not within the size limits of existing dedicated free lists are maintained in a single nondedicated free list. The heap manager first looks in the dedicated free lists, starting with the list whose memory-block size is greater than or equal to the requested size. It continues to look in the dedicated free lists until it either finds the smallest block that is greater than or equal to the requested size or it exhausts the dedicated free lists. If no block is found on the dedicated free lists that is large enough, the heap manager does a linear search of the nondedicated free list for the first block that satisfies the request. (This may not be the smallest free block that would satisfy the request, since the order of the nondedicated free list is implementation-dependent.) Dedicated free lists are organized in last-in, first-out (LIFO) order.

To produce dedicated free lists in a heap, pass nonzero arguments for the *cbMinDed* and *cbMaxDed* parameters of the **WinCreateHeap** function.

25.3.7 Destroying Heaps

The **WinDestroyHeap** function destroys a heap that was created by using the **WinCreateHeap** function. If **WinCreateHeap** calls the **DosAllocSeg** function to allocate space for the heap, then **WinDestroyHeap** calls **DosFreeSeg** to free the allocated segment. Otherwise, **WinDestroyHeap** frees only the heap handle that is passed to it.

The return value is zero if the **WinDestroyHeap** function is successful. Otherwise, the return value is the heap handle that is passed to the function. (A reason this function could fail is if the heap handle is invalid.) This function is not affected by allocated memory objects within the heap.

The return value is zero for success because of the following idiom for destroying a heap and invalidating the variable that contains the handle to the heap, all in one line of code:

```
hHeap = WinCreateHeap(...);

/* Code that manipulates the heap. */

hHeap = WinDestroyHeap(hHeap);
```

25.4 Summary

The following functions allow your application to manage heaps:

WinAllocMem Returns a near pointer to a memory block of the specified size on the heap. Returns NULL if the memory cannot be allocated.

WinAvailMem Returns the largest free block of memory on the heap.

WinCreateHeap Creates a heap that can be used for memory management.

WinDestroyHeap Destroys a heap. All memory objects on the heap are lost.

WinFreeMem Frees a memory block that was allocated by using the **WinAllocMem** function.

WinLockHeap Returns a far pointer to the beginning of the segment containing the heap and locks the heap. This is useful for a heap allocated in a separate segment.

WinReallocMem Reallocates a heap memory block to a new size. If the new size is larger than the previous size, a new block is allocated by using the **WinAllocMem** function and the previous block is copied to the new block.

Chapter

26

Clipboard

26.1 Introduction 333

26.2 About the Clipboard 333
26.2.1 Cutting, Copying, and Pasting Data 333
26.2.2 Clipboard-Data Formats 334
26.2.3 Shared Memory and the Clipboard 334

26.3 Using the Clipboard 335
26.3.1 Putting Data on the Clipboard 335
26.3.2 Retrieving Data from the Clipboard 336
26.3.3 Becoming the Clipboard Viewer 337
26.3.4 Becoming the Clipboard Owner 339
26.3.5 Custom Clipboard Formats 340
26.3.5.1 Assigning a Unique Format ID 341
26.3.5.2 Display Formats 341
26.3.6 Delayed Rendering 341

26.4 Summary 342
26.4.1 Standard Clipboard Formats 342
26.4.2 Clipboard Functions 342
26.4.3 Clipboard Messages 343

26.1 Introduction

This chapter describes how to use the clipboard to transfer data between applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window messages and message queues
- MS OS/2 memory management and shared memory

26.2 About the Clipboard

The clipboard is a set of functions that can be used by Presentation Manager applications for exchanging data. In particular, the clipboard provides support for the generalized cut, copy, and paste user-interface common to Presentation Manager applications. The clipboard supports data formats common to most applications, as well as allowing individual applications to define new formats for special purposes.

The data on the clipboard is maintained in memory only. Clipboard data is lost when the computer is turned off.

Data exchange on the clipboard is controlled by the user. An application should not perform any clipboard operations unless the user explicitly initiates them. Other MS OS/2 features, such as pipes, queues, and shared memory should be used when data exchange is needed without the knowledge of the user. For example, an application that continuously passes remotely collected data to a data-analysis application should not use the clipboard. Such an application should use the other interprocess data communication capabilities of MS OS/2 instead.

26.2.1 Cutting, Copying, and Pasting Data

All Presentation Manager programs should support the cut, copy, and paste data exchange in a single application and between applications. These are all user-initiated operations. Typically, a user selects data in an application, called the "current selection." The application should provide visual feedback, such as inverting the data display, to indicate the current selection. The user can then initiate a cut, copy, or paste operation on the current selection.

The standard cut, copy, and paste operations are summarized below:

Operation	Description
Cut	Copies the current selection to the clipboard and deletes the current selection from the application document. The previous contents of the clipboard are destroyed.
Copy	Copies the current selection to the clipboard. The current selection remains unchanged. The previous contents of the clipboard are destroyed.
Paste	Deletes the current selection and replaces it with the contents of the clipboard. The contents of the clipboard are not changed.

Operation	Description
Clear	Deletes the current selection without putting the data on the clipboard. The contents of the clipboard are not changed.

26.2.2 Clipboard-Data Formats

The clipboard accepts data in several formats. MS OS/2 supports three standard formats: text, bitmap, and metafile. Applications can use these predefined formats or create their own formats.

Typically, all formats on the clipboard are simply different representations of the most recent selection on the clipboard. For example, a word processor that supports multiple fonts might write a selection to the clipboard in three formats: straight text, rich text, and metafile. Another application (pasting from the clipboard) could then choose the format most applicable to its own capabilities. All of these formats refer to the same data.

26.2.3 Shared Memory and the Clipboard

Because data on the clipboard can be accessed by different applications, it is important that it be stored in shared memory. The clipboard uses two types of memory: selectors for shareable segments (allocated by the **DosAllocSeg** function), and Presentation Manager objects such as bitmaps and metafiles. Clipboard functions use two flag values, CFI_SELECTOR and CFI_HANDLE, to distinguish each memory type.

When an application writes either a bitmap or a metafile to the clipboard, it passes a bitmap or metafile handle to the clipboard. The clipboard functions make the object "shareable." The application cannot access the object once it closes the clipboard. Once an object is passed to the clipboard, it can no longer be used in the application. Likewise, when an application requests a bitmap or metafile from the clipboard, it receives a handle to a bitmap or metafile object that is good only until the application closes the clipboard. Typically, the application either uses the object immediately before closing the clipboard, or it copies the object to local memory for future use, then closes the clipboard.

To give a selector to the clipboard, an application must put data into a segment allocated by using the **DosAllocSeg** function with the SEG_GIVEABLE attribute. Once an application passes the selector for that segment to the clipboard and closes the clipboard, the clipboard owns the segment. The application cannot access the shared segment. When an application requests a selector from the clipboard, the clipboard gives the segment to the application. An application must use the selector before closing the clipboard or it must copy the data from the shared segment to a local segment before closing the clipboard.

An application must use a shared segment when writing text to the clipboard. An application must also use shared segments for any application-defined clipboard formats. In this case, it is important to specify the CFI_SELECTOR flag when sending data to the clipboard.

26.3 Using the Clipboard

Applications should use the clipboard when cutting, copying, or pasting data. Typically, an application places data on the clipboard for cut and copy operations and removes data from the clipboard for paste operations. An application can use the standard clipboard-data formats or create its own formats. An application that uses custom clipboard formats often becomes the clipboard owner, assuming control of drawing or freeing data on the clipboard.

Clipboard data does not need to be generated, or rendered, when it is placed on the clipboard. Instead, an application can delay rendering, waiting until the data is requested by another application.

Finally, an application can become the clipboard viewer, showing the clipboard contents, and receiving messages when the clipboard contents change.

26.3.1 Putting Data on the Clipboard

To put data on the clipboard, an application must first call the **WinOpenClipbrd** function to verify that other applications are not trying to retrieve or set clipboard data. The **WinOpenClipbrd** function does not return if another thread has the clipboard open. The **WinOpenClipbrd** function waits until the clipboard is free or there is a message in the calling thread's message queue. In practice, this means that the **WinOpenClipbrd** function waits until the clipboard is available or until the calling application responds to a message. If the clipboard cannot be opened before a message arrives, the application receives the message and the **WinOpenClipbrd** function continues to try to open the clipboard. The **WinOpenClipbrd** function does not return until the clipboard is open. However, the application continues receiving messages.

Once an application successfully opens the clipboard, it should remove any previously stored data on the clipboard by calling the **WinEmptyClipbrd** function. Although the clipboard supports many data formats, all the formats on the clipboard should represent the same data at any one time. For this reason, it is important to clear the clipboard of old data before writing new data. If the clipboard is not cleared, writing a format that already exists on the clipboard will replace the old data with the new data.

When the clipboard is cleared, an application should write its data to the clipboard in as many standard formats as possible. For each format, the application should pass the data to the clipboard by calling the **WinSetClipbrdData** function, specifying the data format. Because the clipboard is not cleared when a new format is written to the clipboard, all new data formats coexist with each other until the clipboard is cleared by the next clipboard user.

Data passed to the clipboard can take many forms, depending on the format of the data. For text data, the data handle is a selector to a shared segment containing the text. For bitmap data, the data handle is a bitmap handle. For a metafile format, the data handle is a metafile handle. If an application passes NULL for the data handle, it renders the data on request.

Once an application passes a selector or a handle to the clipboard, the application should not alter the contents of that segment or handle. The clipboard owns that data from then on.

Finally, when an application finishes writing the clipboard data, it should release the clipboard by calling the **WinCloseClipbrd** function so that other applications can use the clipboard.

The following code fragment shows how an application places text data on the clipboard, how it opens the clipboard, copies the text to a shared segment, empties the clipboard, and passes the selector to the clipboard:

```
if (WinOpenClipbrd(hab)) {

    /*
     * Allocate a shareable segment for the data szClipString in the
     * application's copy of the text.
     */

    if (usSuccess = DosAllocSeg(strlen(szClipString) + 1,
            &sel, SEG_GIVEABLE )) {

        /* Make a far pointer (selector:0) out of the selector. */

        pszDest = MAKEP(sel, 0);

        /* Set up the source pointer to point to text. */

        pszSrc = &szClipString[0];

        /* Copy the string to the segment. */

        while (*pszDest++ = *pszSrc++);

        /* Clear old data from the clipboard. */

        WinEmptyClipbrd(hab);

        /*
         * Pass the selector to the clipboard in CF_TEXT format. Note
         * that the selector must be a ULONG value.
         */

        fSuccess = WinSetClipbrdData(hab, (ULONG) sel,
            CF_TEXT, CFI_SELECTOR);

        /* Close the clipboard. */

        WinCloseClipbrd(hab);
    }
}
```

26.3.2 Retrieving Data from the Clipboard

To retrieve data from the clipboard, an application must first call the **WinOpenClipbrd** function to verify that no other applications are trying to retrieve or set the clipboard data.

Once an application successfully opens the clipboard, it should call the **WinQueryClipbrdData** function, specifying a preferred format. If that format is not available, indicated by a NULL return from the **WinQueryClipbrdData** function, the application should repeat calls to the **WinQueryClipbrdData** function for other possible formats until it either receives the data or runs out of format choices.

If the clipboard contains one of the requested formats, the **WinQueryClipbrdData** function returns a 32-bit integer, the meaning of which depends on the particular format. For text data, the return value is a selector (in the lower 16 bits of the long integer) to a shareable segment containing the text. For bitmap data, the return value is a bitmap handle. For metafile data, the return value is a metafile handle.

Whatever the format, the handle or selector returned is valid only while the clipboard remains open. An application can use the data while the clipboard is open or copy the data to its own memory and use it after the clipboard is closed.

It is important that an application close the clipboard as soon as possible so that other applications can access it.

The following code fragment shows how to open the clipboard, retrieve data in the requested format, copy the data to a local segment, and close the clipboard:

```
if (WinOpenClipbrd(hab)) {
   if (hText = WinQueryClipbrdData(hab, CF_TEXT)) {

      /* Turn the selector into a pointer. */

      pszClipText = MAKEP((SEL) hText, 0);

      /* Copy text from the selector segment to a local segment. */

      while (*pszLocalText++ = *pszClipText++);
   }
   WinCloseClipbrd(hab);
}
```

26.3.3 Becoming the Clipboard Viewer

A window can become a clipboard viewer and display the current contents of the clipboard. The clipboard viewer is informed whenever the clipboard contents change. Typically, the clipboard viewer is a window that can draw the standard clipboard formats. The clipboard viewer is a convenience for the user; it does not have any effect on the data-transaction functions of the clipboard.

To create a clipboard viewer, an application calls the **WinSetClipbrdViewer** function, specifying the window in which the clipboard data will be displayed. This is usually the client window of an application. There can only be one clipboard viewer at any time in the system, so setting a clipboard viewer replaces any previous clipboard viewer. The **WinQueryClipbrdViewer** function receives the handle to the current clipboard viewer so that the application can reset it when finished with the clipboard viewer.

Once a window becomes the clipboard viewer, it receives WM_DRAWCLIPBOARD messages whenever the contents of the clipboard change. The window should respond to these messages by drawing the contents of the clipboard.

The clipboard viewer displays all the standard formats and should process CFI_OWNERDISPLAY items by sending the appropriate clipboard message to the clipboard owner.

Three special formats exist for of the clipboard viewer: CF_DSPTEXT, CF_DSPBITMAP, and CF_DSPMETAFILE. Applications that write data to the clipboard in private formats should also write the data in one of these formats. These DSP formats should be a representation of the private formats. If the clipboard viewer does not find one of the standard formats (CF_TEXT, CF_BITMAP, or CF_METAFILE), it can search for one of the DSP formats. Display strategies for these formats are the same as for the corresponding standard formats.

The following code fragment shows how a sample clipboard viewer responds to the WM_DRAWCLIPBOARD message, drawing text and bitmap data in its window. Note that the code uses the data retrieved from the clipboard before closing the clipboard. An alternate strategy would be to copy the data to a local segment and then close the clipboard. In any case, the original data from the clipboard cannot be used after the clipboard is closed.

```
case WM_DRAWCLIPBOARD:
    if (!WinOpenClipbrd(hab))
        return 0L;

    if (hText = WinQueryClipbrdData(hab, CF_TEXT)) {
        pszText = MAKEP((SEL) hText, 0);

        hps = WinGetPS(hwnd);
        WinQueryWindowRect(hwnd, &rect);

        WinDrawText(hps,
            0xFFFF,                     /* null-terminated string  */
            pszText,                    /* the string              */
            &rect,                      /* where to put the string */
            CLR_BLACK,                  /* foreground color        */
            CLR_WHITE,                  /* background color        */
            DT_CENTER | DT_VCENTER | DT_ERASERECT);
        WinValidateRect(hwnd, (PRECTL) NULL, FALSE);
        WinReleasePS(hps);
    }
    else if (hBitmap = WinQueryClipbrdData(hab, CF_BITMAP)) {
        hps = WinGetPS(hwnd);
        ptlDest.x = ptlDest.y = 0;
        WinQueryWindowRect(hwnd, &rect);
        WinFillRect(hps, &rect, CLR_WHITE);
        WinDrawBitmap(hps,
            hBitmap,
            (PRECTL) NULL,              /* draws entire bitmap     */
            &ptlDest,                   /* destination             */
            CLR_BLACK,                  /* foreground color        */
            CLR_WHITE,                  /* background color        */
            DBM_NORMAL);                /* bitmap flags            */
        WinValidateRect(hwnd, (PRECTL) NULL, FALSE);
        WinReleasePS(hps);
    }

    WinCloseClipbrd(hab);               /* closes the clipboard    */
    return 0L;
```

The clipboard viewer uses a similar sequence of calls to get clipboard data when responding to a WM_PAINT message.

The clipboard viewer is also responsible for sending messages to the clipboard owner when clipboard data has the attribute CFI_OWNERDISPLAY. Typically, an application sets the attribute CFI_OWNERDISPLAY only for private clipboard formats and not for any standard formats. The clipboard viewer must send messages to the clipboard owner when the clipboard owner does not provide a standard clipboard format in addition to its private formats. In this case, the viewer sends messages to the clipboard owner of a CFI_OWNERDISPLAY format to draw, scroll, and resize the clipboard-image data.

The clipboard viewer determines the attributes of a particular clipboard format by calling the **WinQueryClipbrdFmtInfo** function. The identity of the current owner can be found by calling the **WinQueryClipbrdOwner** function.

26.3.4 Becoming the Clipboard Owner

The clipboard owner is any application window that is connected to the clipboard data. To become the clipboard owner, an application must call the **WinSetClipbrdOwner** function. The following are situations in which an application should call the **WinSetClipbrdOwner** function to become the clipboard owner:

- The application calling the **WinSetClipbrdData** function passes a NULL selector or handle to the clipboard, indicating that the application renders the data in a particular format on request. As a result, the system sends rendering requests to the current clipboard owner.
- The application calling the **WinSetClipbrdData** function passes data with the attribute CFI_OWNERFREE, indicating that the application frees memory for data when the clipboard is emptied. As a result, the system sends owner-free requests to the current clipboard owner.
- The application calling the **WinSetClipbrdData** function passes data with the attribute CFI_OWNERDISPLAY, indicating that the owner application draws the data in the clipboard viewer. As a result, the clipboard viewer sends drawing-related requests to the current clipboard owner.

The window specified in the call to the **WinSetClipbrdOwner** function should respond to the following messages:

Message	Description
WM_RENDERFMT	Sent by the system to the clipboard owner when a particular format with delayed rendering must be rendered. The receiver should render the data in the specified format and pass it to the clipboard by calling the **WinSetClipbrdData** function. For more information, see Section 26.3.6.
WM_RENDERALLFMTS	Sent by the system to the clipboard owner just before the owner application terminates. The receiver should render the clipboard data in all formats on the clipboard with delayed rendering. It should pass the data for each format to the clipboard by calling the **WinSetClipbrdData** function. For more information, see Section 26.3.6.
WM_DESTROYCLIPBOARD	Sent by the system to the clipboard owner when the clipboard is cleared by another application calling the **WinEmptyClipbrd** function. The receiver should free the memory occupied by any clipboard formats using the attribute CFI_OWNERFREE.
WM_SIZECLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains the data handle with the attribute CFI_OWNERDISPLAY and

Message	Description
	when the clipboard-viewer changes size. When the clipboard viewer is being destroyed or reduced to an icon, this message is sent with the coordinates of the opposite corners set to (0,0), which permits the owner to free its display resources.
WM_VSCROLLCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the attribute CFI_OWNERDISPLAY and when an event occurs in the clipboard-viewer scroll bars. The receiver should respond to this message by scrolling the image, invalidating the appropriate area of the clipboard viewer, and updating the scroll-bar position.
WM_HSCROLLCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the attribute CFI_OWNERDISPLAY and when an event occurs in the scroll bars of the clipboard viewer. The receiver should respond to this message by scrolling the image, invalidating the appropriate area of the clipboard viewer, and updating the scroll-bar position.
WM_PAINTCLIPBOARD	Sent by the clipboard viewer to the clipboard owner when the clipboard contains data with the attribute CFI_OWNERDISPLAY and when the clipboard-viewer client area needs repainting. The receiver should respond to this message by painting the requested format (by calling **WinGetPS** for the window handle of the clipboard viewer).

An application automatically loses ownership of the clipboard when the clipboard data is cleared by the **WinEmptyClipbrd** function. Ownership is necessary only when data is present on the clipboard. Typically, an application loses ownership when another application places data on the clipboard.

26.3.5 Custom Clipboard Formats

Applications often use custom clipboard formats when standard formats are insufficient for representing clipboard data. For example, a word processor might have a rich-text format that contains font and style information in addition to the usual text characters. Clearly, if the word processor uses the clipboard to support cut, copy, and paste operations for moving data in its documents, a standard text format would be inadequate.

Such a word processor should write at least two formats to the clipboard for each cut or copy operation: a standard text format representing the text of the current selection, and a private rich-text format representing the true state of the selection. If the word processor performs a paste operation using clipboard data, it can use the rich-text format to retain all formatting. If another application requests the same data, it can use the standard-text format if it does not recognize the private format. The word processor should also be able to render data in CF_BITMAP and CF_METAFILE formats for painting or drawing applications.

26.3.5.1 Assigning a Unique Format ID

Each private format must have an identification number when it is written to the clipboard. To obtain a unique ID number for a private clipboard format, the application should register the name of the format in the system atom table. The system assigns a unique ID number for the format name. Other applications that know the format name can query the system atom table for the format ID. An application can interpret its own private formats and can request them from the clipboard for cutting and pasting its own data. Other applications that know the private format ID can also interpret the formatted data. The following code fragment illustrates how an application obtains a unique identification number for a clipboard format. This technique can be used either by the application that creates the format or by another application.

```
hatomtbl = WinQuerySystemAtomTable();
formatID = WinAddAtom(hatomtbl, "SuperCAD_Format");
```

26.3.5.2 Display Formats

Three standard display formats exist for applications that use private formats: CF_DSPTEXT, CF_DSPBITMAP, and CF_DSPMETAFILE. These three formats correspond to the standard text, bitmap, and metafile formats with the exception that they are intended only for use by the clipboard viewer. An application that uses a private format should write one of the DSP formats that approximates the appearance of the private data so that the clipboard viewer can display the data regardless of the format. For example, a word processor using the rich-text format should also write a CF_DSPBITMAP formatted picture of the selected text that contains all the type fonts and styles. Note that you might choose delayed rendering for DSP formats because there may not always be a clipboard viewer active on the screen. With delayed rendering, an application does not actually render the format unless it is requested to do so.

26.3.6 Delayed Rendering

An application can pass NULL instead of a selector or a handle, indicating that the data is rendered only when another application requests it from the clipboard. This is useful if an application supports several clipboard formats that are time-consuming to render. With delayed rendering, an application can send NULL handles for each clipboard format that it supports, and render individual formats only when the format is actually requested from the clipboard. An application can either write data for standard formats or choose delayed rendering for more complex formats.

When an application uses delayed rendering for one or more of its clipboard formats, it must become the clipboard owner. As long as the application is the clipboard owner, it receives a WM_RENDERFMT message whenever a request is received by the clipboard for a format using delayed rendering. When the application receives such a message, it should render the data and pass the selector or handle to the clipboard by calling the **WinSetClipbrdData** function. The rules for shared-memory access for rendered data are the same as those for standard clipboard data. This is simply a delayed execution of the operation that occurs if the data does not have delayed rendering.

The clipboard owner with one or more delayed-rendering formats on the clipboard receives a WM_RENDERALLFMTS message just before the clipboard owner application terminates. This insures that the application renders all of its data before terminating.

26.4 Summary

The following sections summarize the standard clipboard data formats, the functions that control the clipboard, and the window messages associated with the clipboard.

26.4.1 Standard Clipboard Formats

The following are the standard clipboard-data formats used in MS OS/2:

CF_BITMAP The handle returned by the **WinQueryClipbrdData** function is a bitmap handle.

CF_DSPBITMAP A bitmap representation of a private-data format. The clipboard viewer uses this format to display a private format.

CF_DSPMETAFILE A metafile representation of a private-data format. The clipboard viewer uses this format to display a private format.

CF_DSPTEXT A text representation of a private-data format. The clipboard viewer uses this format to display a private format.

CF_METAFILE The handle returned by the **WinQueryClipbrdData** function.

CF_TEXT The handle returned by the **WinQueryClipbrdData** function has a selector (in the low word) to an array of text characters that can include newline characters indicating line breaks. The null character indicates the end of the text data.

26.4.2 Clipboard Functions

The following are the MS OS/2 functions that control the clipboard:

WinCloseClipbrd Closes the clipboard, allowing other applications to open and use it. This function sends a WM_DRAWCLIPBOARD message, causing the clipboard contents to be drawn in the clipboard viewer (if any). The clipboard must be open before this function is used.

WinEmptyClipbrd Empties the clipboard, removing and freeing all handles to clipboard data.

WinEnumClipbrdFmts Enumerates the available clipboard data formats. The *fmtPrev* argument specifies the index of the last clipboard-data format enumerated using this function. This index should start at zero, in which case the first available format is obtained. Subsequently, it should be set to the last format index value returned by this function. The return value is the index of the next available clipboard-data format on the clipboard. Enumeration is complete (no further formats are available) when zero is returned.

WinOpenClipbrd Opens the clipboard and prevents other threads and processes from examining or changing the clipboard contents. If another thread or process already has the clipboard open, this function does not return until the clipboard is available. However, it passes messages to the application while it waits for the clipboard.

WinQueryClipbrdData Retrieves data with a specified format from the clipboard. This function returns zero if no data with that format exists on the clipboard.

WinQueryClipbrdFmtInfo Determines whether a particular data format is present on the clipboard. If it is, this function provides information to the caller about that format.

WinQueryClipbrdOwner Returns the current clipboard owner (if any). The *fLock* argument specifies whether the clipboard-owner window should be locked. If the window is locked, the calling application must unlock the window. This window handle should be locked while being used because it may belong to another process. Locking prevents other processes from destroying the window.

WinQueryClipbrdViewer Returns the current clipboard viewer (if any). The *fLock* argument specifies whether the clipboard viewer is locked. If the window is locked, the calling application must unlock the window. This window handle should be locked while being used because it may belong to another process. Locking prevents other processes from destroying the window.

WinSetClipbrdData Puts data in a specified format on the clipboard.

WinSetClipbrdOwner Sets the current clipboard owner. An application should become the clipboard owner when it sends delayed-rendering data to the clipboard or when it has data it must draw in the clipboard viewer.

WinSetClipbrdViewer Sets the current clipboard viewer to a specified window. The clipboard viewer receives the WM_DRAWCLIPBOARD message when the clipboard contents change. This allows the clipboard viewer to display an up-to-date version of the clipboard contents. The clipboard must be open before this function is called.

26.4.3 Clipboard Messages

The following are the window messages used with the clipboard:

WM_DESTROYCLIPBOARD Sent by the system to the clipboard owner when the clipboard is emptied by the **WinEmptyClipbrd** function. If any of the formats have the CFI_OWNERFREE flag set, the clipboard owner must free the data when it receives the WM_DESTROYCLIPBOARD message.

WM_DRAWCLIPBOARD Sent by the system to the clipboard viewer when the clipboard contents change. The clipboard viewer draws the contents of the clipboard.

WM_HSCROLLCLIPBOARD Sent by the clipboard viewer to the clipboard owner when the clipboard data has the CFI_OWNERDISPLAY attribute and an event occurs in the clipboard-viewer scroll bars. The clipboard owner scrolls the clipboard image, invalidating the appropriate sections, and updates the scroll-bar values.

WM_PAINTCLIPBOARD Sent by the clipboard viewer to the clipboard owner when a clipboard format with the CFI_OWNERDISPLAY flag set must be drawn in the clipboard viewer. The owner receives a window handle for the clipboard viewer and uses it as the destination window for drawing the clipboard data.

WM_RENDERALLFMTS Sent by the system to the clipboard owner when the owner application is being destroyed. The clipboard owner should render all formats that it can generate and pass a handle or selector for each format to the clipboard by calling the **WinSetClipbrdData** function. This ensures that the clipboard contains valid data even though the application that rendered the data is destroyed.

WM_RENDERFMT Sent by the system to the clipboard owner when clipboard data must be rendered. The receiver of this message renders the data and sends it to the clipboard by calling the **WinSetClipbrdData** function.

WM_SIZECLIPBOARD Sent by the clipboard viewer to the clipboard owner when the clipboard viewer is resized and contains data with the attribute CFI_OWNERDISPLAY.

WM_VSCROLLCLIPBOARD Same as the WM_HSCROLLCLIPBOARD message.

Chapter
27

Dynamic Data Exchange

27.1 Introduction 347

27.2 About Dynamic Data Exchange 347
27.2.1 Client and Server Interaction 347
27.2.2 Sample DDE Relationship 348

27.3 Using Dynamic Data Exchange 349
27.3.1 Detailed DDE Example 349
27.3.2 DDE Message Contents 350
27.3.3 Unique Data Formats 353
27.3.4 Sample DDE Transactions 353
27.3.4.1 Initiating an Exchange Between Two Applications 353
27.3.4.2 Positive WM_DDE_ACK Response 354
27.3.4.3 Negative WM_DDE_ACK Response 355
27.3.4.4 One-Time Data Transfer Between Two Applications 355
27.3.4.5 Permanent Data Link Between Two Applications 356
27.3.4.6 Executing Commands in a Remote Application 358
27.3.4.7 Terminating an Exchange Between Two Applications 359
27.3.5 Synchronization Rules 359

27.4 Summary 360
27.4.1 Functions 360
27.4.2 Messages 360
27.4.3 DDE Status Flags 361

27.1 Introduction

This chapter describes how to use dynamic data exchange (DDE) messages to transfer data between applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window messages and message queues
- MS OS/2 memory management and shared memory
- Clipboard data-exchange model

27.2 About Dynamic Data Exchange

The dynamic data exchange (DDE) protocol is a set of messages and guidelines that allow MS OS/2 Presentation Manager applications to share data freely, using either one-time data transfers or ongoing exchanges, in which applications send updates to one another as new data becomes available.

The DDE protocol uses messages for signaling between applications that share data. The DDE protocol uses shared memory as the means of transferring data from application to application. DDE defines some structures to store the shared memory objects.

DDE is different from the clipboard data-transfer mechanism that is also part of MS OS/2. One difference is that the clipboard is almost always used as a one-time response to a specific action by the user (such as choosing Paste from a menu). DDE, on the other hand, is often initiated by a user but typically continues without the user's further involvement.

27.2.1 Client and Server Interaction

DDE transactions always consist of a client application and a server application. The client initiates the exchange by requesting data from the server. The server responds to the data requests by providing data to the client. A server can have many clients at the same time, and a client can request data from multiple servers.

An application can be both a client and a server. For instance, an application might receive data from another application as a client, and then act as a server by passing the data to another application.

The important distinction between a client and a server is that the client initiates the DDE transaction.

27.2.2 Sample DDE Relationship

There are many potential uses of DDE in real-time data-acquisition applications. This section discusses an example of one such use: a DDE-based real-time system for tracking portfolios. Two hypothetical Presentation Manager applications cooperate in this example. One application, named "Collector," is a specialized interface that draws data from an on-line data service. The other application is a spreadsheet. Both applications use the DDE protocol. In the described transactions the spreadsheet application is the client—that is, the application that initiates DDE transactions—and the on-line data-collection application is the server.

The sample spreadsheet has the following layout:

	A	B	C	D
1	Stock	Shares	Price	Extension
2	BTRX	1000	148	148000
3	HLOW	2000	26	52000
4	WRLD	200	24	4800
5	ZMXI	2000	93	186000
6				390800

Without DDE, this spreadsheet could be updated by using the clipboard to manually copy numbers from the screen display of the Collector application into the spreadsheet. This would require screen sharing or switching between applications, and would also require that the user pay attention to the price data and personally undertake the data exchange.

With DDE, this system could be much more automatic, providing the spreadsheet with the current values for multiple data items without intervention by the user. DDE would allow the user to set up an exchange between the server and client applications that would keep the spreadsheet up-to-date whenever a change occurred in the value of specified stocks. Once this connection was established, the cell values in the spreadsheet would always reflect the most current data available from the server. This system would facilitate the timely analysis of real-time data.

The usefulness of the DDE protocol is not restricted to specialized real-time data-acquisition applications. Productivity software in general can benefit significantly from the protocol. For example, suppose a monthly report is prepared using a graphics-and-text word processor, and that the report includes graphs generated in a separate business-graphics package. Without DDE, it would be necessary to manually copy and paste each month's new graphs into each month's report. With DDE, the word processor can establish a permanent link to the charting application, so that any changes made by the user to the charting document are reflected in the word-processing document, either automatically or on request. This makes the routine of document preparation much simpler for the user.

27.3 Using Dynamic Data Exchange

A DDE transaction between two applications actually takes place between two windows, one for each of the participating applications. Applications open a window for each conversation they engage in. (Note that such windows are typically not visible.) A window is identified by its handle. The window belonging to the server application is the server window; the window belonging to the client application is the client window.

After a conversation has been initiated by the client, the client interacts with the server by issuing transactions. When issuing a transaction, the client requests that the server perform a particular action. There are six types of transactions: request, advise, unadvise, poke, execute, and terminate. These transactions are permitted only within an exchange begun by using the WM_DDE_INITIATE message. DDE transactions are one-way: the client application always issues the transactions. If the server issues a transaction to the client, the server must initiate a new exchange for that purpose. The server becomes the client in this new exchange. (The only exception to the one-way rule is the terminate transaction, which can be issued by either the client or the server.)

27.3.1 Detailed DDE Example

This section presents a more detailed view of the workings of the DDE protocol. It discusses the example of the Collector and spreadsheet interaction and illustrates forwarding stock quotes from the Collector application to the spreadsheet. For the sake of simplicity, this example will be limited to the exchange of quotes for a single stock, BTRX.

The Collector DDE server application is started first. Typically, applications designed to operate as dedicated DDE servers have some user interface for initialization and then run as icons at the bottom of the Presentation Manager screen. As part of the initialization process, the Collector DDE server application goes through whatever steps are necessary (entering passwords, testing, etc.) to ensure that data can be provided to clients.

The spreadsheet is started next, and the stock-portfolio document is loaded. At this time, the spreadsheet calls the **WinDdeInitiate** function, which sends a WM_DDE_INITIATE message to all current top-level frame windows.

The WM_DDE_INITIATE message is a request to initiate an exchange with an application on a specified topic—in this case, NYSE. An application can accept this message by responding with a positive WM_DDE_INITIATEACK message, or can decline it by passing the message on to the **WinDefWindowProc** function. If no application accepts the request, the spreadsheet assigns an error value to the external reference and its DDE activity concludes.

If the Collector application acknowledges the request, the spreadsheet can use the newly established exchange to request the Collector application to provide continuous updates on a specified data item. To make this request, the spreadsheet posts a WM_DDE_ADVISE message to the Collector application (actually, to a window within the Collector application that is acting as the message recipient for DDE messages), indicating that updates should be sent every time there is a new value available for the data item named "BTRX," and that the updates should be in a particular format—for example, DDEFMT_TEXT.

Upon receiving this message, the Collector application records the request in its database and posts a WM_DDE_ACK message to the spreadsheet. From then on, the Collector application posts a WM_DDE_DATA message to the spreadsheet application (actually, to the window in the spreadsheet that initiated the exchange) whenever it receives a new BTRX stock quote from the server. Each of these messages carries a selector for a shared memory object. The object itself contains the data, rendered in the requested format. Whenever the spreadsheet receives such a message, it retrieves the data from the referenced memory object and uses the data to update the value of the cell containing the external reference.

The periodic updates continue until the spreadsheet document is closed. At that point the spreadsheet application posts a WM_DDE_UNADVISE message to the Collector application, indicating that further updating is not necessary. Upon receipt of this message, the Collector application removes the corresponding data request from its database and posts a positive WM_DDE_ACK message back to the spreadsheet.

Finally, unless the spreadsheet initiates other data exchanges under this same topic, it posts a WM_DDE_TERMINATE message to the Collector application, indicating the end of the DDE transaction. The Collector application responds with a WM_DDE_TERMINATE message.

27.3.2 DDE Message Contents

DDE uses the three-level hierarchy—application, topic, and item—to uniquely identify a unit of data. An item is a data object that can be passed in a DDE transaction. For example, an item might be a single integer, a string, several paragraphs of text, or a bitmap. A topic is a logical data context. For applications that operate on file-based documents, topics are usually filenames; for other applications they are other application-specific strings. Using the Collector and spreadsheet model described earlier, the application name is collector, the topic name is NYSE, and the item name is BTRX.

There are two data structures used for DDE transactions: the **DDEINIT** structure and the **DDESTRUCT** structure. The **DDEINIT** structure is used for the WM_DDE_INITIATE and WM_DDE_INITIATEACK messages. **DDEINIT** contains pointers to the application-name and topic-name strings. The **DDEINIT** structure has the following form:

```
typedef struct _DDEINIT {
    USHORT cb;
    PSZ    pszAppName;
    PSZ    pszTopic;
} DDEINIT;
```

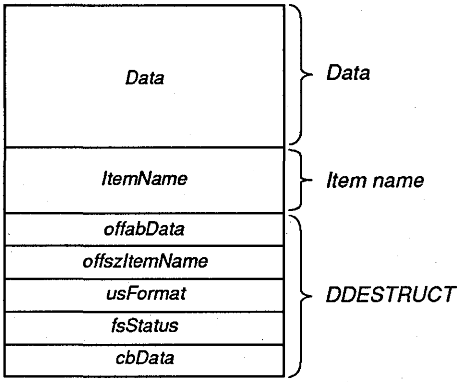
An application typically does not need to fill in a **DDEINIT** structure, since the operating system fills it in automatically when the application calls the **WinDdeInitiate** or **WinDdeRespond** function. It is important, however, to understand the organization of the **DDEINIT** structure when receiving a WM_DDE_INITIATE or WM_DDE_INITIATEACK message, so that you can extract the application name and the topic name.

The **DDESTRUCT** structure is passed with all DDE messages except WM_DDE_INITIATE and WM_DDE_INITIATEACK. It contains a byte count of the data, the format of the data, the item name, a status word, and the data being transferred. The **DDESTRUCT** structure has the following form:

```
typedef struct _DDESTRUCT {
    ULONG  cbData;
    USHORT fsStatus;
    USHORT usFormat;
    USHORT offszItemName;
    USHORT offabData;
} DDESTRUCT;
```

The data in a DDE message is contained in a shared memory segment. The sender allocates a segment large enough to hold one of the two data structures described above and the actual data to be transferred, and passes the selector for the memory as part of the message. The layout of a typical DDE segment is shown in Figure 27.1. The first part of the DDE segment is occupied by the **DDESTRUCT** structure. Next comes the item-name string. Following the name string is the actual data to be transferred. The offset fields of the **DDESTRUCT** structure must be set to point to the name string and the beginning of the data. The **cbData** field must also be set to indicate the number of bytes of data.

Figure 27.1
Typical DDE Segment



The sender must allocate the segment as SEG_GIVEABLE and call the **DosGiveSeg** function to share the segment with the receiving application. To share a segment, the sender needs to know the process identifier of the recipient. The process identifier can be obtained by calling the **WinQueryWindowProcess** function for the recipient's window handle.

The sender should call the **DosFreeSeg** function to free its copy of the segment selector as soon as it has given the shared segment selector to the recipient. The recipient should call **DosFreeSeg** when it is finished using the segment. The sender should not try to access the segment once it has been sent to the recipient in a DDE message.

The following code fragment shows a function that creates a shared segment for a DDE transaction. The function parameters include the destination window for the DDE message, the item name for the transaction, the status word, the format of the data, the actual data to be transferred (if any), and the length of the data. The segment allocated by this function must be big enough to hold the **DDESTRUCT** structure, the item name, and the actual data to be transferred. The function returns a pointer **(PDDESTRUCT)** to a shared segment that is ready to post as part of a DDE message.

```
PDDESTRUCT MakeDDESegment(hwndDest, pszItemName, fsStatus, usFormat,
    pabData, usDataLen)
HWND   hwndDest;
PSZ    pszItemName;
USHORT fsStatus;
USHORT usFormat;
PVOID  pabData;
USHORT usDataLen;

{
    PDDESTRUCT pddes;           /* pointer to DDESTRUCT         */
    USHORT     usItemLen;       /* length of item name          */
    USHORT     usTotalLen;      /* total length of segment      */
    SEL        selBuf;          /* local selector for segment   */
    SEL        selShared;       /* shared selector for segment  */
    USHORT     receiverPID;     /* process ID of server         */
    USHORT     receiverTID;     /* thread ID of server          */

    usItemLen = FarStrLen(pszItemName) + 1;

    usTotalLen = sizeof(DDESTRUCT) + usItemLen + usDataLen;

    if (! DosAllocSeg(usTotalLen, &selBuf, SEG_GIVEABLE)) {

        /* Initialize DDESTRUCT. */

        pddes = SELTOPDDES(selBuf);
        pddes->cbData = usTotalLen;
        pddes->fsStatus = fsStatus;
        pddes->usFormat = usFormat;
        pddes->offszItemName = sizeof(DDESTRUCT);
        if ((usDataLen) && (pabData))
            pddes->offabData = sizeof(DDESTRUCT) + usItemLen;
        else
            pddes->offabData = 0;

        /* Copy item name immediately following DDESTRUCT. */

        FarStrCopy(DDES_PSZITEMNAME(pddes), pszItemName);

        /* Copy data immediately following item name. */

        FarStructCopy(DDES_PABDATA(pddes), pabData, usDataLen);

        /* Get process identifier of server. */

        WinQueryWindowProcess(hwndDest, &receiverPID, &receiverTID);

        /* Give the segment away. */

        if (!DosGiveSeg(selBuf, receiverPID, &selShared)) {
            pddes = SELTOPDDES(selShared);
            return (pddes);
        }
    }

    /* Else could not allocate or share segment. */

    return (NULL);
}
```

This function is used in many examples in the following sections to demonstrate the creation of DDE shared segments. You may want to define a similar function in your own programs as well.

27.3.3 Unique Data Formats

Whenever you exchange data by using the DDE protocols you must specify the format of the data in the **usFormat** field of the **DDESTRUCT** structure. The system-defined standard format is DDEFMT_TEXT, which indicates text data.

Applications can define their own data formats. Each nonstandard DDE format must have a unique identification number. The application should register the name of the format in the system atom table, receiving an identification number for that format name. Other applications that have the name of the format can also query the system atom table for the format's identification number. This method ensures that all applications use the same atom to identify a format.

The following code fragment shows how an application can obtain a unique identification number for a DDE format. This technique can be used by the application that creates the format and by an application that is able to use the format.

```
hatomtbl = WinQuerySystemAtomTable();
formatID = WinAddAtom(hatomtbl, "SuperCAD_Format");
```

27.3.4 Sample DDE Transactions

This section discusses beginning and ending a DDE transaction and the five basic types of interchange supported by DDE. Each of the following subsections provides a detailed description of the message protocols that are associated with the transactions it discusses.

27.3.4.1 Initiating an Exchange Between Two Applications

To initiate a DDE transaction, the client calls the **WinDdeInitiate** function, specifying the server application-name and topic-name strings. This function sends a WM_DDE_INITIATE message to all frame windows whose parent is HWND_DESKTOP. Because the message is sent rather than posted, **WinDdeInitiate** requires all of the message's recipients to respond to the message before control is returned. Either the application name or the topic name can be a null string, in which case the server ignores that name. For example, a client could send a valid application name with a null topic name to request an exchange on all available topics for that application.

The server applications that respond to the WM_DDE_INITIATE message will call the **WinDdeRespond** function, as shown in the following pseudocode:

```
If ((specific app requested and server is instance of app) or
    (specific app not requested){

     If (specific topic requested)
          If (server can support topic)

               acknowledge the requested topic

     Else
          acknowledge each supported topic
}
```

To acknowledge a specific topic, the server responds with the following code fragment:

```
WinDdeRespond(hwndClient, hwndServer, pszAppName, pszTopicName);
```

To acknowledge more than one topic, the server makes one such response for each topic. This initiates an exchange on each topic. The client should post a WM_DDE_TERMINATE message for all unneeded transactions.

The System Topic

Applications are encouraged to support the "System" topic at all times. This topic provides a context for information that may be of general interest to any partners in a DDE transaction. DDE applications should request an exchange on the System topic with a NULL application name when they start up, to find out what kinds of information other DDE-capable programs can provide.

The System topic should support the following terms, as well as any other items the application may use:

Item	Description
SysItems	A list of the items supported under the System topic by this application.
Topics	A list of the topics supported by the application at the current time (this may vary from moment to moment).
ReturnMessage	Supporting detail for the most recently issued WM_DDE_ACK message. (This is useful when more than eight bits of application-specific return code are required.)
Status	An indication of the current status of the application.
Formats	A list of DDE format numbers that the application can render.

Individual elements of lists should be delimited by tabs (the DDEFMT_TEXT format).

27.3.4.2 Positive WM_DDE_ACK Response

A client or server often must positively acknowledge a DDE message that it receives by posting a WM_DDE_ACK message with the DDE_FRESPONSE flag set in the status word of the **DDESTRUCT** structure. Sending a positive

WM_DDE_ACK message means that the sender will respond to the previous message. The following code fragment is an example of a positive acknowledgment message:

```
pddeStruct = MakeDDESegment(hwndDest,   /* handle of destination  */
    "BTRX",                             /* item name              */
    DDE_FACKREQ,                        /* status flags           */
    DDEFMT_TEXT,                        /* data format            */
    NULL,                               /* no data for request    */
    0);                                 /* data length            */

WinDdePostMsg(hwndDest,                 /* handle of destination  */
    hwndSource,                         /* handle of source       */
    WM_DDE_ACK,                         /* message                */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                  */
```

27.3.4.3 Negative WM_DDE_ACK Response

When an application receives a DDE message that it cannot respond to (such as a request for data in a format that it does not support), the application must post a WM_DDE_ACK message with the DDE_NOTPROCESSED flag set in the status word of the **DDESTRUCT** structure. The following code fragment is an example of a negative acknowledgment message:

```
pddeStruct = MakeDDESegment(hwndDest,   /* handle of destination  */
    "BTRX",                             /* item name              */
    DDE_NOTPROCESSED,                   /* status flags           */
    DDEFMT_TEXT,                        /* data format            */
    NULL,                               /* no data for request    */
    0);                                 /* data length            */

WinDdePostMsg(hwndDest,                 /* handle of destination  */
    hwndSource,                         /* handle of source       */
    WM_DDE_ACK,                         /* message                */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                  */
```

If an application is busy when it receives a DDE message, it can post a WM_DDE_ACK message with the DDE_FBUSY flag set.

27.3.4.4 One-Time Data Transfer Between Two Applications

A client application can use the DDE protocol to obtain a data item from a server (WM_DDE_REQUEST), or to submit a data item to a server (WM_DDE_POKE). In either case, the client must have already initiated an exchange with the server, as described earlier.

The client posts a WM_DDE_REQUEST message to the server, specifying an item and format by allocating a shared segment and filling in a **DDESTRUCT** structure and passing the structure to the **WinDdePostMsg** function. For example, if a DDE exchange has been started on the NYSE topic, the client could request data for the BTRX item by using the following code fragment. (For an example of how to allocate and initialize a shared memory segment, see Section 27.3.2.)

```
pddeStruct = MakeDDESegment(hwndServer, /* handle of server       */
    "BTRX",                             /* item name              */
    0,                                  /* status flags           */
    DDEFMT_TEXT,                        /* data format            */
    NULL,                               /* no data for request    */
    0);                                 /* data length            */

WinDdePostMsg(hwndServer,               /* handle of server       */
    hwndClient,                         /* handle of client       */
    WM_DDE_REQUEST,                     /* message                */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                  */
```

If the server is unable to satisfy the request, it sends the client a negative WM_DDE_ACK message. If the server can satisfy the request, it renders the item in the requested format and includes it with a **DDESTRUCT** structure in a shared memory object and posts a WM_DDE_DATA message to the client, as shown in the following code fragment:

```
pddeStruct = MakeDDESegment(hwndClient, /* handle of client       */
    "BTRX",                             /* item name              */
    0,                                  /* status flags           */
    DDEFMT_TEXT,                        /* data format            */
    pabData,                            /* pointer to data        */
    usDataLen);                         /* data length            */

WinDdePostMsg(hwndClient,               /* handle of client       */
    hwndServer,                         /* handle of server       */
    WM_DDE_DATA,                        /* message                */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                  */
```

Upon receiving a WM_DDE_DATA message, the client processes the data item. The **DDESTRUCT** structure at the beginning of the shareable segment contains a status word indicating whether the sender has requested an acknowledgment message. If the DDE_FACKREQ bit of the status word is set, the client should send the server a positive WM_DDE_ACK message.

Upon receiving a negative WM_DDE_ACK message, the client can ask for the same item again, specifying a different DDE format. Typically, a client will first ask for the most complex format it can support, then step down, if necessary, through progressively simpler formats until it finds one the server can provide.

27.3.4.5 Permanent Data Link Between Two Applications

A client application can use DDE to establish a link to an item in a server application. When such a link is established, the server sends periodic updates about the linked item to the client (typically, whenever the data that is associated with the item in the server application has changed). A permanent "data stream" is established between the two applications and remains in place until it is explicitly disconnected.

The client sends the server a WM_DDE_ADVISE message to set up the data link. (Of course, the client must have first initiated an exchange by using the WM_DDE_INITIATE message, as described previously.) The advise message contains a shared-memory pointer containing a **DDESTRUCT** structure with the item name, format information, and status information, as shown in the following code fragment:

```
pddeStruct = MakeDDESegment(hwndServer, /* handle of server        */
    "BTRX",                             /* item name               */
    DDE_FACKREQ,                        /* status flags            */
    DDEFMT_TEXT,                        /* data format             */
    NULL,                               /* no data for advise      */
    0);                                 /* data length             */

WinDdePostMsg(hwndServer,               /* handle of server        */
    hwndClient,                         /* handle of client        */
    WM_DDE_ADVISE,                      /* message                 */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                   */
```

If the server has access to the requested item and can render it in the desired format, the server records the new link and then sends the client a positive WM_DDE_ACK message. Until the client issues a WM_DDE_UNADVISE message, the server sends data messages to the client every time the source data changes that is associated with the item in the server application.

If the server is unable to satisfy the request, it sends the client a negative WM_DDE_ACK message.

When a link is established with the DDE_FNODATA status bit cleared, the client is sent the data each time the data changes. In such cases, the server renders the new version of the item in the previously specified format and posts a WM_DDE_DATA message to the client, as shown in Section 27.3.4.4.

When the client receives a WM_DDE_DATA message, it extracts data from the shared memory segment by using the **DDESTRUCT** structure at the beginning of the segment. If the DDE_FACK status bit is set in the status word of the **DDESTRUCT** structure, the client must post a positive WM_DDE_ACK message to the server.

When a link is established with the DDE_FNODATA status flag set, a notification, not the data itself, is posted to the client each time the data changes. In this case, the server does not render the new version of the item when the source data changes, but simply posts a WM_DDE_DATA message with zero bytes of data and the DDE_FNODATA status flag set, as shown in the following code fragment:

```
pddeStruct = MakeDDESegment(hwndClient, /* handle of client        */
    "BTRX",                             /* item name               */
    DDE_FNODATA,                        /* status flags            */
    DDEFMT_TEXT,                        /* data format             */
    NULL,                               /* no data                 */
    0);                                 /* data length             */

WinDdePostMsg(hwndClient,               /* handle of client        */
    hwndServer,                         /* handle of server        */
    WM_DDE_DATA,                        /* message                 */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                   */
```

The client can request the latest version of the data by performing a regular one-time WM_DDE_REQUEST transaction, or it can simply ignore the data-change notice from the server. In either case, if the DDE_FACK status bit is set, the client should send a positive WM_DDE_ACK message to the server.

To terminate a specific item link, the client posts a WM_DDE_UNADVISE message to the server, as shown in the following code fragment:

```
pddeStruct = MakeDDESegment(hwndServer, /* handle of server        */
    "BTRX",                             /* item name               */
    DDE_FACKREQ,                        /* status flags            */
    DDEFMT_TEXT,                        /* data format             */
    NULL,                               /* no data for unadvise    */
    0);                                 /* data length             */

WinDdePostMsg(hwndServer,               /* handle of server        */
    hwndClient,                         /* handle of client        */
    WM_DDE_UNADVISE,                    /* message                 */
    pddeStruct,                         /* shared-segment pointer */
    1);                                 /* retry                   */
```

The server checks that the client currently has a link to the specified item in this exchange. If the link exists, the server sends a positive WM_DDE_ACK message to the client and no longer sends updates on the item in this exchange. If the server has no such link, it sends a negative WM_DDE_ACK message.

To terminate all links for a particular exchange, the client application posts a WM_DDE_UNADVISE message with a null item name to the server. The server checks that the exchange has at least one link currently established. If so, the server posts a positive WM_DDE_ACK message to the client, and no longer sends any updates in the exchange. If the server has no links in the exchange, it posts a negative WM_DDE_ACK message.

27.3.4.6 Executing Commands in a Remote Application

A Presentation Manager application can use the DDE protocol to cause a command or series of commands to be executed in another application. Such remote executions are performed by means of the WM_DDE_EXECUTE transaction.

To execute a remote command, the client application posts to the server a WM_DDE_EXECUTE message containing a selector for a shared-memory object that contains a **DDESTRUCT** structure and a command string, as shown in the following code fragment:

```
pddeStruct = MakeDDESegment(hwndServer,   /* handle of server      */
    "BTRX",                          /* item name                  */
    DDE_FACKREQ,                     /* status flags               */
    DDEFMT_TEXT,                     /* data format                */
    pabData,                         /* pointer to command string */
    usDataLen);                      /* data length                */

WinDdePostMsg(hwndServer,            /* handle of server           */
    hwndClient,                      /* handle of client           */
    WM_DDE_EXECUTE,                  /* message                    */
    pddeStruct,                      /* shared-segment pointer     */
    1);                              /* retry                      */
```

The server attempts to execute the specified string according to some agreed-upon protocol. If successful, the server posts a positive WM_DDE_ACK message to the client; if unsuccessful, a negative WM_DDE_ACK message is posted.

27.3.4.7 Terminating an Exchange Between Two Applications

At any time, either the client or the server may terminate an exchange by using the following procedure to issue a WM_DDE_TERMINATE message. Similarly, both the client application and server application should be able to receive a WM_DDE_TERMINATE message at any time.

An application must end its exchanges before terminating. The application posts a WM_DDE_TERMINATE message with a NULL shared-segment pointer, as shown in the following code fragment. A WM_DDE_TERMINATE message stops all transactions for a given exchange.

```
WinDdePostMsg(hwndDest,         /* handle of destination     */
    hwndSource,                 /* handle of source          */
    WM_DDE_TERMINATE,           /* message                   */
    NULL,                       /* no shared-segment pointer */
    1);                         /* retry                     */
```

The WM_DDE_TERMINATE message means that the sender will send no further messages in that exchange and that the recipient may destroy its DDE window. The recipient must always send a WM_DDE_TERMINATE message promptly in response; it is not permissible to send a negative, busy, or positive WM_DDE_ACK message instead.

If the sender of the original termination request receives any other message before the WM_DDE_TERMINATE message arrives from the recipient of the request, no response should be sent to this other message; the sender of the other message may already have destroyed the window to which the response would be sent.

27.3.5 Synchronization Rules

A window processing DDE requests from another window must process them strictly in the order in which the requests were received.

A window does not need to apply this first-in, first-out rule between requests from different windows—that is, it may provide asynchronous support for multiple processes. For example, a window might have the following requests in its queue:

- 1: Request Message from window *x*
- 2: Request Message from window *y*
- 3: Request Message from window *x*

The window must process request 1 before 3, but it does not need to process 2 before 3. If *y* has a lower priority than *x*, the window follows the order 1, 3, 2.

If a server is unable to process an incoming request because it is waiting for an external process, it must post a busy WM_DDE_ACK message to the client, to prevent deadlock. A busy WM_DDE_ACK message can also be sent if the server is unable to process an incoming request quickly.

27.4 Summary

This section describes the functions, messages, and status flags associated with the DDE protocol.

27.4.1 Functions

Three functions simplify the use of DDE messages:

WinDdeInitiate Sends a WM_DDE_INITIATE message containing the specified application name and topic name to all top-level frame windows in the system.

WinDdePostMsg Posts a DDE message to the specified recipient window.

WinDdeRespond Sends a WM_DDE_INITIATEACK message in response to a WM_DDE_INITIATE message.

27.4.2 Messages

The predefined DDE messages are summarized below:

WM_DDE_ACK Sent as acknowledgment to many DDE messages.

WM_DDE_ADVISE Sent from the client to the server, requesting the server to provide a data update whenever the specified data item changes.

WM_DDE_DATA Sent from the server to the client to notify the client that the data is available.

WM_DDE_EXECUTE Sent from the client to the server, containing a text string that the server should execute as a command or series of commands.

WM_DDE_INITIATE Sent by a client application to initiate an exchange with one or more server applications. This message is often sent to all current applications by calling the **WinBroadcastMsg** function.

WM_DDE_INITIATEACK Sent by a server application as a positive response to a WM_DDE_INITIATE message. This message specifies the server application name and the topic on which the server will open a DDE transaction.

WM_DDE_POKE Sent as an unsolicited data message for the recipient, which should reply with a WM_DDE_ACK message to indicate whether or not it accepted the data.

WM_DDE_REQUEST Sent from the client to the server to request that a data item be sent to the client.

WM_DDE_TERMINATE Sent by either the client or the server to terminate the exchange.

WM_DDE_UNADVISE Sent from the client to the server to indicate that the specified item should no longer be updated. This message requests the server to remove the link to the data item set up by the WM_DDE_ADVISE message.

27.4.3 DDE Status Flags

The following constant values control various aspects of a DDE transaction. They can be combined in the *fsStatus* word of the **DDESTRUCT** structure by using the OR operator.

DDE_FACK Set for positive acknowledgment.

DDE_FACKREQ Set to acknowledge DDE messages for the application.

DDE_FAPPSTATUS Upper eight bits of status word are reserved for the application-specific data.

DDE_FBUSY Set if application is busy.

DDE_FNODATA Set if there is no data transfer for the WM_DDE_ADVISE message.

DDE_FRESERVED Reserved; must be zero.

DDE_FRESPONSE Set if there is a response to a WM_DDE_REQUEST message.

DDE_NOTPROCESSED Set if the message is not supported.

Chapter

28

Hooks

28.1 Introduction 365

28.2 About Hooks 365

28.3 Types of Hooks 365

28.3.1 Input Hook 365

28.3.2 Send-Message Hook 366

28.3.3 Message-Filter Hook 367

28.3.4 Journal-Record Hook 368

28.3.5 Journal-Playback Hook 370

28.3.6 Help Hook 370

28.4 Using Hooks 372

28.5 Hook Example 373

28.5.1 Installing a System Hook 373

28.5.2 System-Hook Code 374

28.6 Summary 374

28.6.1 Functions 374

28.6.2 Hook Types 375

28.1 Introduction

This chapter describes how to use hooks in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window messages and message queues
- Focus window and input guidelines

28.2 About Hooks

MS OS/2 is based on a message-passing model. The behavior of most programs depends on the messages that the program receives. Messages can be generated by input devices, such as the keyboard and mouse, or they can originate within the system as a way of managing and communicating between system resources.

MS OS/2 provides hooks to allow applications to monitor and modify the message stream. Hooks can be installed in either the system queue, so that they affect all applications, or in an individual thread's message queue, so that only messages for that queue are affected.

Because many applications may install hooks at the same time, most hooks are arranged in chains. The system passes a message to the first hook in the chain, and then to the next hook in the chain, and so on until the message is delivered to the destination application. Each hook may modify the message or stop its progress through the chain, preventing it from reaching the application. Hooks in a chain are called in last-installed, first-called order.

28.3 Types of Hooks

There are six different types of hooks. You can install the different types of hooks in any combination, although some of the hook types can be installed only in the system queue.

The following sections describe the available types of hooks. Each type of hook is expressed as a function with a unique syntax.

28.3.1 Input Hook

This hook monitors the input queue and is called whenever a message is about to be returned by the **WinGetMsg** or **WinPeekMsg** function. Typically, the input hook is used to monitor mouse and keyboard input and other messages that are posted to a queue.

The syntax for the input hook is as follows:

BOOL CALLBACK InputHook(HAB *hab*, **PQMSG** *pQmsg*, **USHORT** *fs*)

The *pQmsg* parameter is a far pointer to a **QMSG** structure that contains information about the message. The **QMSG** structure has the following form:

```
typedef struct _QMSG {
    HWND   hwnd;
    USHORT msg;
    MPARAM mp1;
    MPARAM mp2;
    ULONG  time;
    POINTL ptl;
} QMSG;
```

The *fs* parameter of the **InputHook** function can contain the following flags from the **WinPeekMsg** function, indicating whether or not the message is removed from the queue:

PM_NOREMOVE
PM_REMOVE

If this hook function returns TRUE, the message is not passed to the rest of the hook chain or to the application—effectively ending the message. If the function returns FALSE, the message is passed to the next hook in the chain, or to the application if no other hooks exist.

The input hook can modify a message by changing the contents of the **QMSG** structure, then return FALSE to pass the modified message to the rest of the chain. The following problems may occur when a hook modifies a message:

- If the caller uses the **WinPeekMsg** or **WinGetMsg** function with a message filter range (*msgFilterFirst* through *msgFilterLast*), the message is checked before the hooks are called, not after the hooks are called. This means the caller may receive messages that are not in the range of the caller's message filter.
- If the hook changes a WM_CHAR message from one character into another—for example, if the hook modifies all TAB messages into F6 messages—a program that depends on the key state will be unable to interpret the result. (When the TAB key is translated into the F6 key, the application receives the F6 keystroke and enters a process loop, waiting for the F6 key to be released; the application calls the **WinGetKeyState** function with the HWND_DESKTOP and VK_F6 arguments).

28.3.2 Send-Message Hook

This hook is called whenever a message is sent by using the **WinSendMsg** function. The hook chain is called before the message is delivered to the recipient window. Typically, the send-message hook is used to monitor messages that are not posted to a queue. By installing an input hook and a send-message hook, you can effectively monitor all window messages.

The syntax for the send-message hook is as follows:

VOID CALLBACK SendMsgHook(HAB *hab*, **PSMHSTRUCT** *psmh*,
BOOL *fInterTask*)

The *psmh* parameter is a far pointer to an **SMHSTRUCT** structure that contains information about the message. The **SMHSTRUCT** structure has the following form:

```
typedef struct _SMHSTRUCT {
    MPARAM mp2;
    MPARAM mp1;
    USHORT msg;
    HWND   hwnd;
} SMHSTRUCT;
```

The *fInterTask* parameter of the **SendMsgHook** function is TRUE if the message is sent between two threads, or FALSE if the message is sent within a thread.

The send-message hook does not return a value, and the next hook in the chain is always called. This hook can modify values in the **SMHSTRUCT** structure before returning.

28.3.3 Message-Filter Hook

This hook is called during system modal loops, which include tracking the window size and window movement, displaying a dialog box or message box, scroll-bar tracking, menu-selection tracking, and window-enumeration operations. The message-filter hook is typically used to provide input-message filtering (such as monitoring hot keys) during modal dialog processing.

The syntax of the message-filter hook is as follows:

BOOL CALLBACK MsgFilterHook(HAB *hab*, **USHORT** *msgf*, **PQMSG** *pQmsg*)

The *msgf* parameter has the following three values:

Value	Meaning
MSGF_DIALOGBOX	Message originated while processing a modal dialog window.
MSGF_MESSAGEBOX	Message originated while processing a message box.
MSGF_TRACK	Message originated while tracking a control (such as a scroll bar).

The *pQmsg* parameter of the **MsgFilterHook** function is a far pointer to a **QMSG** structure containing information about the message.

If this hook returns TRUE, the message is not passed to the rest of the hook chain or to the application. If it returns FALSE, the message is passed to the next hook in the chain, or to the application if no other hooks exist.

This hook allows applications to perform message filtering during modal loops that is equivalent to the typical filtering for the main message loop. For example, applications often examine a new message in the main event loop between the time they retrieve the message from the queue and the time they dispatch it, performing special processing as appropriate. The following code fragment shows a main message loop that tests for the ENTER key and the ESC key before dispatching an event:

```
while (WinGetMsg(hab, (PQMSG) &qmsg, (HWND) NULL, 0, 0)) {

   if ((qmsg.msg == WM_CHAR) && !(LOUSHORT(qmsg.mp1) & KC_KEYUP)
            && !hwndQueue) {
        switch (HIUSHORT(qmsg.mp2)) {
            case VK_ESC:

                /* Use the ESC key.       */

                continue;

            case VK_NEWLINE:

                /* Use the newline event. */

                continue;

            default:
                break;
        }
    }
    WinDispatchMsg(hab, (PQMSG) &qmsg);
}
```

You cannot usually do this sort of filtering during a modal loop, since the loop created by the **WinGetMsg** and **WinDispatchMsg** functions is executed by the system. If you install a message-filter hook, the hook is called by the system between **WinGetMsg** and **WinDispatchMsg** in the modal processing loop.

Your application can also call the message-filter loop directly by calling the **WinCallMsgFilter** function. By using this function, you can use the same code to filter messages in your main message loop and during modal loops. In the example shown previously for processing main message loops, you would encapsulate the filtering operations in a message-filter hook and call **WinCallMsgFilter** between the calls to the **WinGetMsg** and **WinDispatchMsg** functions, as shown in the following code fragment:

```
while (WinGetMsg(hab, (PQMSG) &qmsg, (HWND) NULL, 0, 0)) {
    if (!WinCallMsgFilter(hab, (PQMSG) &qmsg, 0))
        WinDispatchMsg(hab, (PQMSG) &qmsg);
}
```

The last argument of the **WinCallMsgFilter** function is simply passed to the hook; the application can enter any value. This value can be used by the hook to determine where the hook was called from, by defining a constant such as MSGF_MAINLOOP.

28.3.4 Journal-Record Hook

This hook monitors the system queue and allows the application to record input events. Typically, it is used to record a set of mouse and keyboard events that can be played back later by using the journal-playback hook. The journal-record hook can be installed only in the system queue.

The syntax for the journal-record hook is as follows:

VOID CALLBACK JournalRecordHook(HAB *hab*, **PQMSG** *pQmsg*)

The *pQmsg* parameter is a far pointer to a **QMSG** structure containing information about the message. The hook is called after the raw input has been processed enough to create valid WM_CHAR or double-click mouse messages and the window-handle field of the **QMSG** structure has been set.

The journal-record hook does not return a value, and the next hook in the chain is always called. Typically, this hook saves the input event to the disk, to be played back later. The **hwnd** field of the **QMSG** structure is not important and is ignored when the message is read back.

Character messages are passed to the journal-record hook as WM_VIOCHAR messages. The format of the WM_VIOCHAR message is as follows:

```
fsKeyFlags = (USHORT) SHORT1FROMMP(mp1);     /* key flags    */
uchRepeat = (UCHAR) CHAR3FROMMP(mp1);        /* repeat count */
uchScanCode = (UCHAR) CHAR4FROMMP(mp1);      /* scan code    */
usChr = LOBYTE(LOWORD(mp2));                 /* character    */
uschKbdScan = HIBYTE(LOWORD(mp2));           /* virtual key  */
```

The WM_VIOCHAR message was chosen over the WM_CHAR message because it preserves that raw information from the keyboard driver that is needed for video input and output.

The following mouse messages are passed to the journal-record hook:

WM_MOUSEMOVE
WM_BUTTON1DOWN
WM_BUTTON1UP
WM_BUTTON2DOWN
WM_BUTTON2UP
WM_BUTTON3DOWN
WM_BUTTON3UP

The positions stored in the mouse messages are in screen coordinates. The system does not combine mouse clicks into double clicks before calling the hook, since there is no guarantee that both clicks will be in the same window when they are read back.

A WM_JOURNALNOTIFY message is passed to the journal-record hook whenever a program calls the **WinGetPhysKeyState** or **WinQueryQueueStatus** function. This message is necessary because the system is reduced to a queue that is only one message deep while a playback hook is active. For example, the user might press the A, B, and C keys while in record mode. While the program is processing the "A" character message, the B key might be down; **WinGetPhysKeyState** will return this information. However, during playback mode, the system only knows that it is currently processing the A key.

The format of the WM_JOURNALNOTIFY message is as follows:

For the **WinGetPhysKeyState** function:

```
ulCmd = LONGFROMMP(mp1);               /* calling function    */
sc = SHORT1FROMMP(mp2);                /* virtual key         */
fsPhysKeyState = SHORT2FROMMP(mp2);    /* physical-key state  */
```

For the **WinQueryQueueStatus** function:

```
ulCmd = LONGFROMMP(mp1);               /* calling function    */
ulQueStatus = LONGFROMMP(mp2);         /* queue status        */
```

28.3.5 Journal-Playback Hook

The journal-playback hook allows an application to insert messages into the system queue. Typically, this hook is used to play back a series of mouse and keyboard events that were recorded earlier by using the journal-record hook. This hook can be installed only in the system queue.

Regular mouse and keyboard input is disabled as long as a journal-playback hook is installed. It is important to note that, since mouse and keyboard input are disabled, this hook can easily hang the system.

The syntax for the journal-playback hook is as follows:

ULONG CALLBACK JournalPlaybackHook(HAB *hab***, PQMSG** *pQmsg***,**
BOOL *fSkip***)**

The *pQmsg* parameter is a far pointer to a **QMSG** structure that the hook fills in with the message to be played back. If the *fSkip* parameter is FALSE, the hook should fill in the **QMSG** structure with the current recorded message. The same message should be returned each time the hook is called, until *fSkip* is TRUE. The same message may be returned many times if an application is examining the queue but not removing the message. If *fSkip* is TRUE, the hook should advance to the next message without attempting to fill in the **QMSG** structure, since the *pQmsg* parameter is NULL when *fSkip* is TRUE.

The journal-playback hook returns a **ULONG** time-out value. This value tells the system how many milliseconds to wait before processing the current message from the playback hook. This allows the hook to control the timing of the events it plays back.

The **time** field of the **QMSG** structure is filled in with the current time before the playback hook is called. The hook should use the time stored in this field instead of the system clock to set up the delays.

The discussion of different types of messages in Section 28.3.4 is also applicable to the journal-playback hook. The only exception has to do with character processing. The journal-record hook is always passed a WM_VIOCHAR message for each character processed by the system. The journal-playback hook, on the other hand, has the option of playing back either WM_VIOCHAR or WM_CHAR messages. The preferred method is to play back WM_VIOCHAR messages, since they contain the information required to function in a video-input-and-output window. If WM_CHAR messages are returned, the appropriate KC bits must be set or cleared.

28.3.6 Help Hook

The help hook is called during the default processing of the WM_HELP message. Help processing is done in two stages: creating the WM_HELP message and calling the help hook. The WM_HELP message can come from the following sources:

- From a WM_CHAR message, after translation by an **ACCEL** structure with the AF_HELP style. The default system accelerator table translates the F1 key into a help message. The WM_HELP message is posted to the current focus window, which can be a menu, a button, a frame, or your client window.

- From a menu-bar selection, when the MIS_HELP style is specified for the menu-bar item. The WM_HELP message is posted to the current focus window.
- From a dialog-box push button, when the BS_HELP style is specified for the push button. The WM_HELP message is posted to the button's owner window, which is normally the dialog window.
- From a message box, when the MB_HELP style is specified for the message box. The WM_HELP message is posted to the message-box dialog window.

The WM_HELP message is posted to the current focus window. The default processing in the **WinDefWindowProc** function is to pass the message up to the parent window. If the message reaches the client window, it can be processed there. If the message reaches a frame window, the default frame-window procedure calls the help hook. The help hook is also called if a WM_HELP message is generated while the application is in menu mode—that is, while a selection is being made from a menu.

The syntax for the help hook is as follows:

BOOL CALLBACK HelpHook(HAB *hab***, USHORT** *usMode***, USHORT** *idTopic***,**
USHORT *idSubTopic***, PRECTL** *prcPosition***)**

If this hook function returns TRUE, the message is not passed to the rest of the hook chain or to the application. If it returns FALSE, the message is passed to the next hook in the chain. The arguments of the help hook provide context information, such as the screen coordinates of the focus window and whether the message originated in a message box or a menu.

The WM_HELP message often goes to a frame window instead of to the client window. The frame window (including dialog boxes and message boxes) processes a WM_HELP message as follows:

- If the window with the focus is the FID_CLIENT window, the frame window passes the WM_HELP message to the FID_CLIENT window.
- If the parent of the window with the focus is the FID_CLIENT frame-control window, the frame window calls the help hook, specifying the following:

  Mode = HLPM_FRAME
  Topic = frame-window identifier
  Subtopic = focus-window identifier
  Position = screen coordinates of focus window

- If the parent of the focus window is not an FID_CLIENT window (it could be the frame window or a second-level dialog window), the frame window calls the help hook, specifying the following:

  Mode = HLPM_WINDOW
  Topic = identifier of parent of focus window
  Subtopic = focus-window identifier
  Position = screen coordinates of focus window

An application receives the WM_HELP message in its dialog-window procedure. The application can ignore the message, in which case the frame-manager action occurs as described, or the application can handle the WM_HELP message directly.

Menu windows receive a WM_HELP message when the user presses the help accelerator key (F1 by default) while a menu is displayed. Menu windows process WM_HELP messages by calling the help hook, specifying the following:

Mode = HLPM_MENU
Topic = identifier of pull-down menu
Subtopic = identifier of selected item in pull-down menu
Position = screen coordinates of selected item

The help hook should respond to the help message by displaying information about the selected menu item.

The **WinDefWindowProc** function processes WM_HELP messages by passing the message to the parent window. The message typically moves up the parent chain until it arrives at a frame window.

28.4 Using Hooks

Applications install hooks by calling the **WinSetHook** function, specifying the type of hook, whether it should go into the system queue or into the queue of a particular thread, and a far pointer to a function entry point.

Procedures installed as hooks in a thread's queue are called only in the context of that thread. This kind of hook is typically a locally defined function.

Procedures installed as hooks in the system queue can be called in the context of any application. System-queue hooks must be defined in separate dynamic-link-library (DLL) modules, because it is not possible to call application-module procedures from other applications. For a sample system-queue DLL module, see Section 28.5.

A hook can be released by calling the **WinReleaseHook** function with the same arguments used when installing the hook. All hooks should be released before the application terminates, although the system automatically releases them if the application does not.

A system hook can be released by using the **WinReleaseHook** function, but the DLL function containing the hook procedure is not freed. This is because system hooks are called in the process context of every Presentation Manager application in the system, causing an implicit call to the **DosLoadModule** function for all of those processes. Since a call to the **DosFreeModule** function cannot be made for another process, there is no way to free the dynamic-link libraries. (However, since the hook procedure is no longer called, the DLL segments may be discarded or swapped.)

28.5 Hook Example

This section shows the main elements of installing and using a system-input-queue hook, although many of the details of the hook are omitted. The example code comes from a larger program that uses a hook to monitor the input queue and display all input messages in an application window on the screen. This example has two main parts: the installing application and the hook DLL. The installing application identifies a hook procedure in the DLL and installs it in the system-queue hook chain. The application then controls the hook through other function calls to the DLL, performing such actions as turning the hook on and off and asking it for the most recent messages.

The system hook is more than a single hook procedure; it is typically a DLL with several support procedures and entry points. This allows the installing application to control and communicate with the hook procedure.

28.5.1 Installing a System Hook

Since this example is a system hook, the hook procedure must be in a separate DLL from the application that installs it. The installing application needs the module handle of the DLL before it can install the hook. The **DosLoadModule** function, given the name of the DLL, returns the handle of the module. Once the module handle is known, the application calls the **WinSetHook** function, passing the module handle, a far pointer to the hook-procedure entry point, and NULL for the message-queue argument, indicating that the hook should go into the system queue. This sequence is shown in the following code fragment:

```
CHAR        achFailName[128];
HMODULE     hmodSpy;
BOOL        fRet;

/*
 * This example assumes that there is a hook procedure named
 * "SpyInputHook" that resides in a DLL file named "spyhook.dll,"
 * and that "spyhook.dll" is in a directory defined by the LIBPATH
 * command in the config.sys file.
 */

if (DosLoadModule(achFailName,          /* failure-name buffer     */
                  sizeof(achFailName),  /* size of failure buffer  */
                  "spyhook",            /* module name             */
                  &hmodSpy) == 0) {     /* address of handle       */

    fRet = WinSetHook(hab,              /* anchor-block handle     */
                  (HMQ) NULL,           /* system queue            */
                  HK_INPUT,             /* hook type               */
                  (PFN) SpyInputHook,   /* far pointer to function */
                  hmodSpy);             /* module handle for DLL   */
}
```

An alternative method for installing a system queue is to provide an installation function in the DLL along with the hook procedure. With this method, the installing application does not need the module handle for the DLL. By linking with the DLL, the application gains access to the installation function. The installation function can supply the module handle and other details in the call to the **WinSetHook** function. The DLL can also contain a function that removes the system-hook procedure; the application can call this function when it terminates.

28.5.2 System-Hook Code

The hook procedure is part of the DLL module. As such, it has access to the resources of the module: its global variables, other support procedures, and any memory allocated by the module. The following code template shows the structure of a typical input-hook procedure. It uses a static variable to control whether or not it should examine the input messages. Although it is not required, the DLL would typically provide another procedure to toggle the state of the variable, so that an application that installs the hook can turn the hook on and off.

This example could save each message in a buffer allocated during the initialization of the DLL module. The DLL module would also provide a calling interface, so that the installing application could periodically request the messages from the buffers.

```
BOOL CALLBACK PASCAL SpyInputHook(hab, lpqmsg)
HAB hab;
PQMSG lpqmsg;
{
    static BOOL fRecording;

    /*
     * The lpqmsg parameter points to a queue-message structure.
     * Returns FALSE to let the message through to the rest of the
     * hook chain. Returns TRUE to prevent further message processing.
     */

    if (!fRecording)
        return FALSE; /* pass message to rest of chain  */

    /* Else examine message and process as appropriate. */

    return FALSE;

}
```

28.6 Summary

This section summarizes the six types of hooks and the functions related to hooks.

28.6.1 Functions

The following functions are used with hooks:

WinCallMsgFilter Calls the currently installed message-filter chain of hooks. This call is useful if the same message filtering is used in the main message loop and the modal system-message loops.

WinReleaseHook Removes a specified procedure from the input-hook chain.

WinSetHook Installs a specified procedure in either the system or queue chain of hooks.

28.6.2 Hook Types

The following constants are used to identify the type of hook being installed or released by the **WinSetHook** or **WinReleaseHook** function:

HK_HELP Monitors the WM_HELP message. Returns **BOOL**. If FALSE, the next hook in the chain is called. If TRUE, the next hook in the chain is not called.

HK_INPUT Monitors messages in the specified message queue. Returns **BOOL**. If FALSE, the next hook in the chain is called. If TRUE, the message is not passed to the next hook in the chain.

HK_JOURNALPLAYBACK Allows applications to insert events into the system-input queue. Returns a **LONG** time-out value. This value is the number of milliseconds to wait before processing the current message. This type never calls the next hook in the chain.

HK_JOURNALRECORD Allows applications to record system-input-queue events. Returns **VOID**. The next hook in the chain is always called.

HK_MSGFILTER Monitors input events during system modal loops. Returns **BOOL**. If FALSE, the next hook in the chain is called. If TRUE, the message is not passed to the next hook in the chain.

HK_SENDMSG Monitors messages sent by using the **WinSendMsg** function. Returns **VOID**. The next hook in the chain is always called.

Chapter
29

Help

29.1 Introduction 379

29.2 About Help 379

29.3 Using Help in an Application 381
29.3.1 Help Menu Item 381
29.3.2 Help in a Dialog Box 381
29.3.3 Help in a Message Box 382
29.3.4 Help for a Menu Item 382
29.3.5 The Help Hook 382
29.3.5.1 Installing a Help Hook 384
29.3.5.2 Help Hook for a Menu Item..................... 384

29.4 Summary 386
29.4.1 Functions 386
29.4.2 Messages 386
29.4.3 Syntax of the Help Hook.................................... 386

29.1 Introduction

This chapter describes how to use the WM_HELP message in your applications. You should also be familiar with the following topics:

- Standard user-interface guidelines
- Window messages and message queues
- Focus window and input guidelines
- Help hook

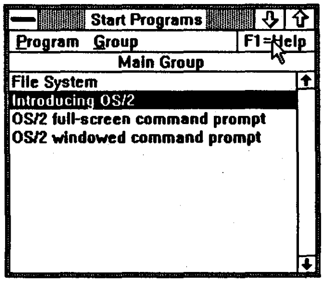
29.2 About Help

One of the key elements of user-friendly software is readily available on-line help. MS OS/2 provides functions and messages to support the implementation of a help system in your application. This chapter discusses the features of MS OS/2 that support on-line help and suggests techniques for implementing a help system. The two main components of the system support for help are the WM_HELP message and the help hook.

The WM_HELP message is defined by the system to notify an application when the user wants help. The WM_HELP message contains context information that allows the application to respond with information appropriate to current conditions. The system posts a WM_HELP message in the following situations:

- During accelerator-table translation for a keyboard event, if the keystroke corresponds to an accelerator-table entry with the AF_HELP attribute. The system default accelerator table translates the F1 key into a WM_HELP message. The WM_HELP message is posted to the focus window. If the F1 key is pressed while a menu has the focus, the help message goes to the menu, which calls the help hook.
- When a menu item with the MIS_HELP style is chosen. All applications should define at least one menu item with the MIS_HELP style in the menu bar. This menu item is typically labeled “F1=Help,” as shown in Figure 29.1. The WM_HELP message is posted to the focus window.
- When a push button with the BS_HELP style is pressed. (Note that this includes the case in which the Help button is pressed in a message box that has the MB_HELP style.) The WM_HELP message is posted to the button’s owner window, which is typically a dialog frame window.

Figure 29.1
Help Menu Item



The Help menu item and the F1 accelerator key provide a standard user-interface to the help system. They allow the user to get general help about the application. The BS_HELP push button style allows applications to provide help within dialog boxes. You should provide a Help push button in every dialog box; the help message should explain the purpose and consequence of any action that could be taken by using that dialog box.

Your application should process the WM_HELP message to provide context-appropriate help to the user. In the case of the Help menu item or the F1 accelerator key, the WM_HELP message is posted to the focus window. The focus window can process the message or it can pass it to the **WinDefWindowProc** function, which sends the message to the parent window. If you have a client window with several child windows that can accept the focus, you can intercept the WM_HELP message in the client window procedure as the message is passed up from the child window with the focus. The state of the application when the help message is received determines the appropriate help, which may be a list of help topics or information about the window that has the focus.

There are limits to what can be accomplished by intercepting the WM_HELP message. In particular, the application does not receive WM_HELP messages generated while a menu item is selected. For example, if the user presses the F1 key while selecting a menu item, the WM_HELP message goes to the menu window. The menu window does not pass the message on; instead, it calls the help hook.

The help hook allows you to install code that is called by menu windows and frame windows when they receive WM_HELP messages. You must install a help hook for your application if you want to provide help for selected menu items, since there is no way to intercept the WM_HELP message that originates while a menu item is selected. In addition, if you don't process a WM_HELP messages in your client window, the message is passed to the frame window, which calls the help hook.

The following sections describe in more detail how to handle WM_HELP messages and when to install a help hook.

29.3 Using Help in an Application

There are two methods of providing help in an application. The simplest method provides general information to the user by processing the WM_HELP message in the client window. The second method, installing a help hook, provides help in situations in which the WM_HELP message is not available to the client window. Since installing a help hook is not difficult, most applications use both methods.

29.3.1 Help Menu Item

Every application should have a Help menu item in the menu bar of its main window. The text of the Help item should be "F1=Help" and it should have a menu-item identifier of zero. The Help menu item should have the MIS_HELP and MIS_BUTTONSEPARATOR styles. The following resource-definition file shows how to place a Help item in the menu bar.

```
MENU ID_MENU_RSRC
BEGIN
    SUBMENU "~File", IDM_FILE
        BEGIN
            MENUITEM "~Quit",      IDM_FI_QUIT
            MENUITEM "",           IDM_FI_SEP3,      MIS_SEPARATOR
            MENUITEM "~About Sample...", IDM_FI_ABOUT
        END
    SUBMENU "~Edit", IDM_EDIT
        BEGIN
            MENUITEM "~Undo",      IDM_ED_UNDO,      0,MIA_DISABLED
            MENUITEM "",           IDM_ED_SEP1,      MIS_SEPARATOR
            MENUITEM "~Cut",       IDM_ED_CUT,       0,MIA_DISABLED
            MENUITEM "C~opy",      IDM_ED_COPY,      0,MIA_DISABLED
            MENUITEM "~Paste",     IDM_ED_PASTE,     0,MIA_DISABLED
            MENUITEM "C~lear",     IDM_ED_CLEAR,     0,MIA_DISABLED
        END
    MENUITEM "F1=Help", 0x00, MIS_HELP | MIS_BUTTONSEPARATOR
END
```

If the user clicks the Help item in the menu bar, the system posts a WM_HELP message to the current focus window. If the client window has the focus, it can process the WM_HELP message to provide general help information for the user. If a child window of the client has the focus, the child window passes the message up the parent-window chain to the client window (by default). In this case, the client window can process the WM_HELP message appropriately for the child window that has the focus.

If the client window passes the WM_HELP message on to the **WinDefWindowProc** function, this function passes the message up the parent-window chain until it finds a frame window. The default frame window processes the WM_HELP message by calling the help hook.

29.3.2 Help in a Dialog Box

You should provide a Help button in each dialog box that your application displays. The Help button should have the BS_HELP style. When the user presses the Help button, a WM_HELP message is posted to the button's owner, which is usually the dialog window. The dialog procedure can process the message and give the user help pertaining to the dialog box. Alternatively, if you pass the WM_HELP message on to the **WinDefDlgProc** function, it will call the help hook; this means you can choose whether to process dialog help in the dialog procedure or in the help hook.

29.3.3 Help in a Message Box

A message box can be given a Help button by specifying MB_HELP when creating the message box. Message boxes are like dialog boxes, except that you do not specify your own dialog procedure. If a message box contains a Help button, the program does not process the WM_HELP message. The default window procedure for the message box responds to the WM_HELP message by calling the help hook. To provide help for message boxes, you must install a help hook. Because the window identifier that you specify for the message box is passed to the help hook, you must use unique identifiers if you want to provide help for multiple message boxes.

29.3.4 Help for a Menu Item

One of the most important functions of a help system is to provide information about individual menu items. The user requests help for a menu item by pressing the F1 key while the menu item is selected. The menu receives the WM_HELP message, since the menu has the focus while the item is selected. The menu responds to the message by calling the help hook, supplying the identifier of the selected item.

You must install a help hook to provide this kind of help for individual menu items. For an example of a help hook that supplies information about menu items, see Section 29.3.5.2.

A method of supplying less comprehensive help for menu items is to use the WM_MENUSELECT message. The menu system sends a WM_MENUSELECT message every time the menu selection changes. The low word of the *mp1* parameter of this message contains the identifier of the item that is changing state, and the high word is a 16-bit Boolean value that describes whether or not the item is chosen; the *mp2* parameter contains the handle of the menu.

If the Boolean value is FALSE, the menu item is selected but not chosen—for example, if the user moves the cursor or mouse pointer over the item while the button is down. An application can use this message to display brief help information at the bottom of the application window.

29.3.5 The Help Hook

You must install a help hook if you want to provide help for message boxes or menu items. You can also use the help hook to provide help when WM_HELP messages are passed up to a frame window. You can install more than one help hook; the system will call them in last-installed, first-called order.

The parameters for a sample help hook are shown in this code fragment:

```
BOOL CALLBACK HelpHook(HAB hab,
    USHORT usMode,
    USHORT idTopic,
    USHORT idSubTopic,
    PRECTL prcPosition);
```

The *usMode* parameter has three possible values:

Value	Description
HLPM_MENU	The help message originated in a menu window when the user pressed the F1 key while a menu item was selected.
HLPM_FRAME	The frame window received a WM_HELP message, and the parent of the focus window is the client window of the frame window.
HLPM_WINDOW	The frame window received a WM_HELP message, and the parent of the focus window is not the client window of the frame window.

The value of the *idTopic* parameter depends on the value of the *usMode* parameter. The following list shows the values of the *idTopic* parameter:

Value	Description
HLPM_MENU	Identifier of submenu containing selected item.
HLPM_FRAME	Identifier of frame window that called the help hook.
HLPM_WINDOW	Identifier of parent of focus window.

The value of the *idSubTopic* parameter depends on the value of the *usMode* parameter. The following list shows the values of the *idSubTopic* parameter:

Value	Description
HLPM_MENU	Identifier of selected menu item (-1 if menubar item is selected).
HLPM_FRAME	Identifier of focus window.
HLPM_WINDOW	Identifier of focus window.

The *prcPosition* parameter of the help hook indicates the screen area from which the help was requested. This argument may be the bounding rectangle of a menu item or the bounding rectangle of the focus window. You should use this information when displaying your help window to avoid covering up the corresponding screen area.

The following code fragment shows a template for a help hook that handles the three possible calling modes:

```
BOOL CALLBACK HelpHook(HAB hab,
    USHORT usMode,
    USHORT idTopic,
    USHORT idSubTopic,
    PRECTL prcPosition);
{
    CHAR szMessage[256];
```

```
        switch(usMode){
            case HLPM_MENU:
                /*
                 * idTopic: submenu identifier
                 * idSubTopic: item identifier
                 * prcPosition: boundary of item
                 */

                break;

            case HLPM_FRAME:
                /*
                 * idTopic: frame identifier
                 * idSubTopic: focus-window identifier
                 * prcPosition: boundary of focus window
                 */

                break;

            case HLPM_WINDOW:
                /*
                 * idTopic: identifier of parent of focus window
                 * idSubTopic: focus-window identifier
                 * prcPosition: boundary of focus window
                 */

                break;
        }
    }
```

29.3.5.1 Installing a Help Hook

Typically, you define a help hook as a function in your program's source code. At run time you install the help hook by calling the **WinSetHook** function, specifying the message queue and a pointer to the help function, as shown in the following code fragment:

```
/* Install the help hook. */

WinSetHook(hab,       /* anchor block                    */
    hmq,              /* message queue                   */
    HK_HELP,          /* hook type                       */
    (PFN) HelpHook,   /* function pointer                */
    NULL);            /* module = NULL => in our .exe */
```

If you install a help hook you must release it by using the **WinReleaseHook** function before your program terminates. The following code fragment shows how to release a help hook:

```
/* Release the help hook. */

WinReleaseHook(hab,   /* anchor block                    */
    hmq,              /* message queue                   */
    HK_HELP,          /* hook type                       */
    (PFN) HelpHook,   /* function pointer                */
    NULL);            /* module = NULL => in our .exe */
```

29.3.5.2 Help Hook for a Menu Item

If the user presses the F1 key while a menu item is selected, the menu receives a WM_HELP message. The menu responds to the help message by calling the help hook with the HLPM_MENU argument to the *usMode* parameter of the HelpHook function. The *idTopic* parameter contains the identifier of the submenu containing the selected item. The *idSubTopic* parameter contains the identifier of the selected item. If the selected item is in the menu bar instead of a submenu, *idTopic* contains the identifier of the menu-bar item and *idSubTopic* contains 0xFFFF.

Your help hook can use this information to determine which menu item was selected when the F1 key was pressed. Using this information, the help hook should display information that explains the function of the menu item. The following code fragment shows a help hook that is capable of giving information about a file menu with two items and an edit menu with five items:

```
BOOL CALLBACK HelpHook(HAB hab,
    USHORT usMode,
    USHORT idTopic,
    USHORT idSubTopic,
    PRECTL prcPosition);
{
    /*
     * idTopic: submenu identifier
     * idSubTopic: item identifier
     */
    switch(usMode){

        case HLPM_MENU:

            switch(idTopic){

                case IDM_FILE:
                    /* One of the File menu items is selected.   */
                    switch(idSubTopic){
                         case OxFFFF:
                           /*
                            * A menu-bar item is selected; display
                            * information on all items in File menu.
                            */
                           break;

                         case IDM_FI_QUIT:
                           /* Display information on Quit item.  */
                           break;

                         case IDM_FI_ABOUT:
                           /* Display information on About item. */
                           break;

                    }      /* ends idSubTopic switch             */
                    break;

                case IDM_EDIT:
                    /* One of the Edit menu items is selected.   */
                    switch(idSubTopic) {
                         case OxFFFF:
                           /*
                            * A menu-bar item is selected; display
                            * information on all items in Edit menu.
                            */
                           break;

                         case IDM_ED_UNDO:
                           /* Display information on Undo item.  */
                           break;

                         case IDM_ED_CUT:
                           /* Display information on Cut item.   */
                           break;

                         case IDM_ED_COPY:
                           /* Display information on Copy item.  */
                           break;

                         case IDM_ED_PASTE:
                           /* Display information on Paste item. */
                           break;
                         case IDM_ED_CLEAR:
                           /* Display information on Clear item. */
                           break;

                    }      /* ends idSubTopic switch             */

                    break; /* ends IDM_EDIT case                 */

            }              /* ends idTopic switch                */

            break;         /* ends HLPM_MENU case                */
    }
}
```

29.4 Summary

This section lists the functions and messages associated with providing help to the user, as well as a syntax description of the help hook.

29.4.1 Functions

The following functions work with the help hooks:

WinSetHook Installs a help hook in an application's message queue.

WinReleaseHook Releases a hook installed by using the **WinSetHook** function.

29.4.2 Messages

The following message is sent to notify an application that the user wants help:

WM_HELP Sent when a user requests help by pressing the F1 key or choosing a menu item or button with the help style.

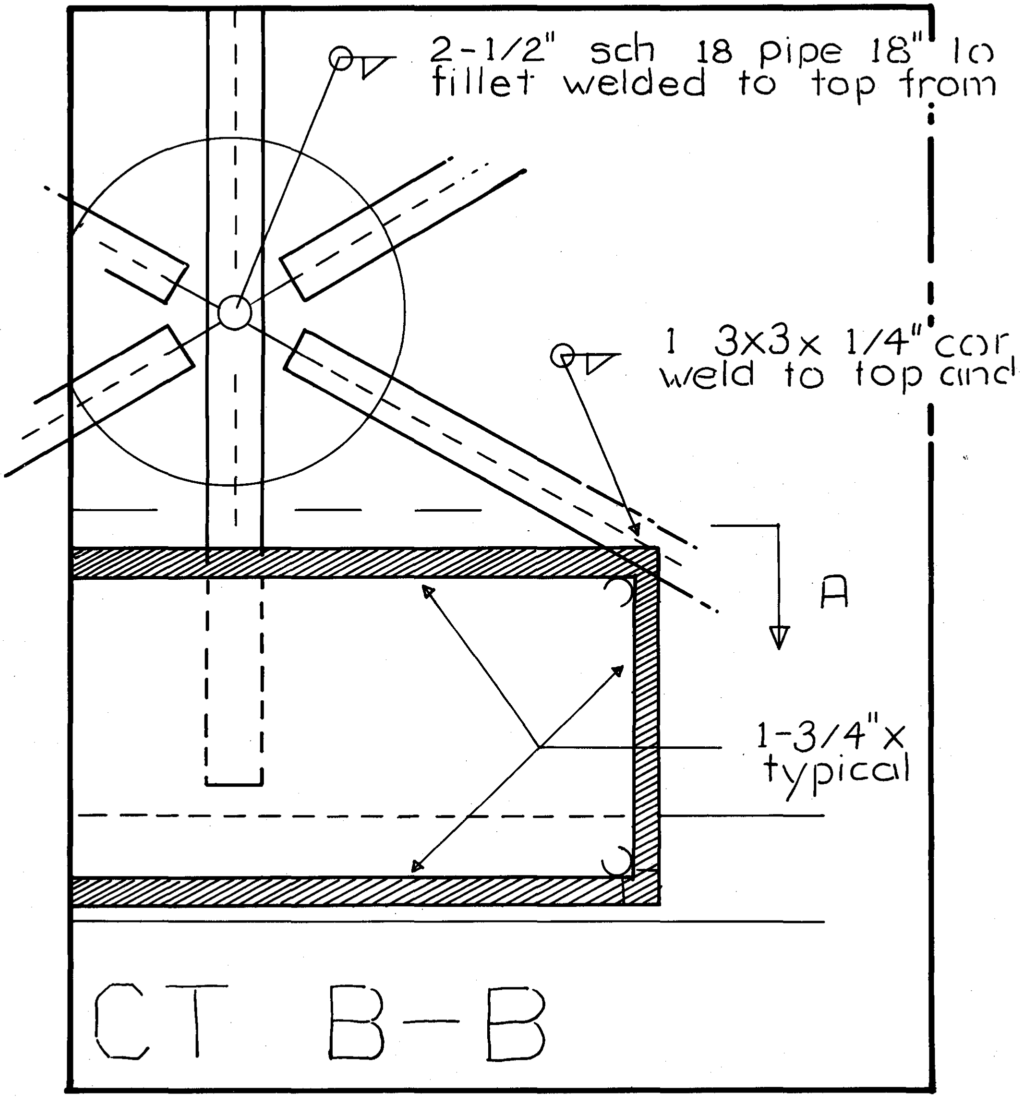
29.4.3 Syntax of the Help Hook

The following syntax description is for the help hook, which is used to provide help for message boxes and menu items:

BOOL CALLBACK HelpHook (HAB *hab***, USHORT** *usMode***,**
USHORT *idTopic***, USHORT** *idSubTopic***, PRECTL** *iprcPosition***)**

Part 3

Graphics Programming Interface



Part 3

Graphics Programming Interface

Chapter 30 Presentation Spaces and Device Contexts 391

Chapter 31 Coordinate Spaces and Transformations 403

Chapter 32 Line and Arc Primitives 425

Chapter 33 Fonts and Character Primitives 439

Chapter 34 Color and Mix Modes 467

Chapter 35 Paths 479

Chapter 36 Area Primitives 493

Chapter 37 Marker Primitives 505

Chapter 38 Bitmaps 511

Chapter 39 Regions 527

Chapter 40 Clipping 539

Chapter 41 Metafiles 549

Chapter 42 Segments and Retained Graphics 561

Chapter

30

Presentation Spaces and Device Contexts

30.1 Introduction 393

30.2 About Presentation Spaces and Device Contexts 393

30.2.1 Presentation Spaces 393

30.2.1.1 Normal Presentation Spaces 393

30.2.1.2 Micro Presentation Spaces 394

30.2.1.3 Saving and Restoring Presentation Spaces 395

30.2.1.4 Destroying Presentation Spaces 396

30.2.2 Device Contexts 396

30.2.2.1 Normal Device Contexts 396

30.2.2.2 Cached Device Contexts 397

30.2.2.3 Closing Device Contexts 397

30.2.3 Linking Presentation Spaces with Devices 397

30.2.4 The Presentation Page 398

30.2.5 Determining Device Capabilities 398

30.3 Using Presentation Spaces and Device Contexts 399

30.3.1 Creating a Normal Presentation Space 399

30.3.2 Creating a Standard Device Context for a Printer 399

30.3.3 Creating a Standard Device Context for a Metafile 400

30.3.4 Reassociating a Normal Presentation Space 400

30.3.5 Associating a Device Context with a Presentation Space 400

30.4 Summary 400

30.1 Introduction

This chapter describes Microsoft Operating System/2 presentation spaces and device contexts. You should also be familiar with the following topics:

- Coordinate spaces and transformations
- Segments

30.2 About Presentation Spaces and Device Contexts

A presentation space is a data structure maintained by MS OS/2 that describes an application's device-independent drawing environment. A device context links a presentation space to a device and gives an application access to important device information.

30.2.1 Presentation Spaces

There are two kinds of presentation spaces: normal and micro. You must use a normal presentation space to display and print output on multiple devices (a video display and a printer, for instance) or if your application uses the segment and retained-drawing functions to generate complex drawings. You should use a micro presentation space only to display and print output on a single device or if your application's output is a simple drawing. For a list of the segment and retained-drawing functions that are not supported by a micro presentation space, see Section 30.2.1.2.

There are two kinds of micro presentation spaces: standard and cached. Use a standard micro presentation space to send output to a printer, a plotter, or any device; use a cached-micro presentation space to send output to a window on the display device.

Table 30.1 summarizes the features and restrictions of each type of presentation space:

Table 30.1 Presentation-Space Features and Restrictions

Presentation-space type	Retained-drawing support	Multiple-device support
Normal	Yes	Yes
Micro	No	No
Cached-micro	No	No (restricted to a video-display window)

30.2.1.1 Normal Presentation Spaces

Because normal presentation spaces use more memory than micro presentation spaces, you should use them only when they are required. To create a normal presentation space, use the **GpiCreatePS** function. You can then associate the normal presentation space with a device and reassociate it later with a new device when you need to direct output to that second device. To associate a normal presentation space with a device when you create the presentation space, use the GPIA_ASSOC flag when you call **GpiCreatePS**. To associate a normal presentation space with a second or a third device later while your application is running, call the **GpiAssociate** function.

30.2.1.2 Micro Presentation Spaces

You can create a standard micro presentation space by calling the **GpiCreatePS** function and specifying the GPIT_MICRO flag. You must associate a standard micro presentation space with a device when you create the presentation space by specifying the GPIA_ASSOC flag as one of the options to **GpiCreatePS** and by supplying a handle that identifies a device context. You cannot reassociate a micro presentation space with another device.

The window manager maintains a cache of micro presentation spaces for windows on a video display. You can access a cached-micro presentation space by calling the **WinGetPS** or **WinBeginPaint** function. You need not associate a cached-micro presentation space with the display; the window manager takes care of this for you.

The following functions are *not* allowed in a micro presentation space:

- **GpiAssociate**
- **GpiBeginElement**
- **GpiCallSegmentMatrix**
- **GpiCloseSegment**
- **GpiCorrelateChain**
- **GpiCorrelateFrom**
- **GpiCorrelateSegment**
- **GpiDeleteElement**
- **GpiDeleteElementRange**
- **GpiDeleteElementsBetweenLabels**
- **GpiDeleteSegment**
- **GpiDeleteSegments**
- **GpiDrawChain**
- **GpiDrawDynamics**
- **GpiDrawFrom**
- **GpiDrawSegment**
- **GpiErase**
- **GpiElement**
- **GpiEndElement**
- **GpiErrorSegmentData**
- **GpiGetData**
- **GpiLabel**
- **GpiOffsetElementPointer**
- **GpiOpenSegment**

- **GpiPutData**
- **GpiQueryBoundaryData**
- **GpiQueryDrawControl**
- **GpiQueryDrawingMode**
- **GpiQueryEditMode**
- **GpiQueryElement**
- **GpiQueryElementPointer**
- **GpiQueryElementType**
- **GpiQueryInitialSegmentAttrs**
- **GpiQueryPickAperturePosition**
- **GpiQueryPickApertureSize**
- **GpiQuerySegmentAttrs**
- **GpiQuerySegmentNames**
- **GpiQuerySegmentPriority**
- **GpiQueryStopDraw**
- **GpiQueryTag**
- **GpiRemoveDynamics**
- **GpiResetBoundaryData**
- **GpiSetDrawControl**
- **GpiSetDrawingMode**
- **GpiSetEditMode**
- **GpiSetElementPointer**
- **GpiSetElementPointerAtLabel**
- **GpiSetInitialSegmentAttrs**
- **GpiSetPickAperturePosition**
- **GpiSetPickApertureSize**
- **GpiSetSegmentAttrs**
- **GpiSetSegmentPriority**
- **GpiSetStopDraw**
- **GpiSetTag**

30.2.1.3 Saving and Restoring Presentation Spaces

You can save the contents of a presentation space, modify its fields, draw in the modified presentation space, and then restore it to its original state. The **GpiSavePS** function saves the contents of a presentation space and the **GpiRestorePS** function restores them. When you call **GpiSavePS**, MS OS/2 copies the following items from the current presentation space onto a special stack:

- Primitive attributes
- Transformation matrices
- Viewing limit
- Clip path
- Clip region
- Current position
- Loaded logical color table
- Loaded logical font

You can push the contents of a presentation space on the stack, and you can do so as many times as is necessary. The **GpiRestorePS** function pops the contents of a presentation space off the stack.

30.2.1.4 Destroying Presentation Spaces

Because presentation spaces consume a considerable amount of memory, you should destroy them whenever your application no longer needs them. To destroy a normal or micro presentation space, call the **GpiDestroyPS** function. Once you finish using a cached-micro presentation space that was accessed by using the **WinGetPS** function, you can release it by calling the **WinReleasePS** function. You need not destroy a presentation space that you accessed by using the **WinBeginPaint** function; MS OS/2 does this for you when you call the **WinEndPaint** function.

30.2.2 Device Contexts

Device contexts link presentation spaces to devices by converting device-independent presentation-space information into device-dependent information. (This conversion occurs in the device driver, a low-level program that is transparent at the API level.) Device contexts also give applications access to important device information such as screen dimensions or printer capabilities.

There are two kinds of device contexts: normal and cached. A normal device context links a presentation space with any type of device. A cached device context can link a presentation space only with a window on a video display.

30.2.2.1 Normal Device Contexts

There are five types of normal device contexts:

- Queued
- Direct
- Information
- Memory
- Metafile

Each of these device contexts serves a specific purpose. A queued device context links a presentation space with a printer or plotter shared by multiple users. Queued devices store print jobs by using a program called a print spooler that keeps track of the order in which jobs arrive at the printer and the order in which they are printed. A direct device context links a presentation space with a printer or a plotter controlled by a single user and does not queue print jobs or other output. An information device context is a special device context that forms a one-way link to a device. An application can use an information device context to query a device but cannot send output to the device. A memory device links a presentation space with system memory. Memory device contexts are useful in applications that keep a "shadow" bitmap of the client area in a window. A metafile device context links a presentation space with a metafile. Applications that create complex drawings use metafiles to store the drawings.

You can create a normal device context by calling the **DevOpenDC** function. This function requires you to specify one of the five types. It also requires that you pass important device-initialization data, including a logical address, the device-driver name, device-driver data, a description of the device type, and information about the queue (if the device is a queued device). The device-initialization data is passed in a **DEVOPENSTRUC** structure. This structure has the following form:

```
typedef struct _DEVOPENSTRUC {      /* dop                          */
    PSZ        pszLogAddress;       /* logical-device address       */
    PSZ        pszDriverName;       /* device-driver name           */
    PDRIVDATA  pdriv;               /* pointer to add'l driver data */
    PSZ        pszDataType;         /* type of queued data          */
    PSZ        pszComment;          /* optional spooler info        */
    PSZ        pszQueueProcName;    /* queue-processor name         */
    PSZ        pszQueueProcParams;  /* queue-processor arguments    */
    PSZ        pszSpoolerParams;    /* spooler arguments            */
    PSZ        pszNetworkParams;    /* network arguments            */
} DEVOPENSTRUC;
```

The last six fields in this structure apply only to queued devices. For more information about the **DEVOPENSTRUC** structure, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

30.2.2.2 Cached Device Contexts

You can obtain a handle to one of the cached device contexts by calling the **WinOpenWindowDC** function. A cached device context is a direct device context that links a presentation space with a window in a video display. You should use a cached device context whenever a task will send output only to a window.

30.2.2.3 Closing Device Contexts

To close a device context that your application opened by calling the **DevOpenDC** function, you can call the **DevCloseDC** function. However, you should not try to close a device context that you opened by using the **WinOpenWindowDC** function; MS OS/2 will do this for you automatically when you destroy the associated window.

30.2.3 Linking Presentation Spaces with Devices

When you call the **GpiCreatePS** function and pass it the GPIA_ASSOC flag or when you call the **GpiAssociate** function, MS OS/2 requires that you pass the device-context handle returned by the **DevOpenDC** or **WinOpenWindowDC** function. This handle identifies the device context that links a presentation space to a device. Once you have established this link, any drawing operations performed using the presentation-space handle will also be performed on the associated device. Once an application is through using a particular device, it should disassociate the presentation space and the device by calling **GpiAssociate** and passing NULL as the second argument. A presentation space can be associated with a new device only after it has been disassociated from the original device, since it is not possible to associate a presentation space with more than one device at a time.

30.2.4 The Presentation Page

One of the arguments to the **GpiCreatePS** function is a **SIZEL** structure that specifies the dimensions of the presentation space's presentation page. A presentation page is a representation of a page on a printer or a plotter or a representation of a maximized window on a video display. MS OS/2 uses presentation pages to scale and position output on a device. For more information about presentation pages, see Chapter 31, "Coordinate Spaces and Transformations."

The page-unit constant PS_UNITS, which is another argument to **GpiCreatePS**, is related to the **SIZEL** structure. This constant can be one of seven values:

- PU_ARBITRARY
- PU_PELS
- PU_LOMETRIC
- PU_HIMETRIC
- PU_LOENGLISH
- PU_HIENGLISH
- PU_TWIPS

The page-unit constant defines the dimensions of each unit on the presentation page. For example, if you specify PU_PELS, the length of each page unit in the *x*-direction will be identical to the width of a pel, and the length of each page unit in the *y*-direction will be identical to the height of a pel on the selected output device. If you specify PU_LOENGLISH, each page unit will measure 0.01 inches in the *x*- and *y*-directions.

If you link a presentation space to a device when you call **GpiCreatePS**, MS OS/2 automatically assigns page dimensions to your presentation page. If you do not link a presentation space to a device when you call **GpiCreatePS**, you must assign the page dimensions.

30.2.5 Determining Device Capabilities

Once you have created a device context for a particular output device, you can determine the capabilities of that device by calling the **DevQueryCaps** function. This function retrieves the following information:

- Device technology (whether the device is a raster or vector device)
- Maximized window dimensions (if the device is a video display)
- Page dimensions (if the device is a printer or plotter)
- Character-box dimensions
- Marker-box dimensions
- Pel resolution

- Color capabilities
- Mix-mode capabilities

You can use this information to, for example, select fonts, set up the presentation page, or create a new logical color table.

30.3 Using Presentation Spaces and Device Contexts

You can use presentation-space and device-context functions to perform the following tasks:

- Create a normal, micro, or cached-micro presentation space.
- Delete, save, or restore a presentation space.
- Create a standard or cached device context.
- Associate a presentation space or device context.
- Destroy a device context.
- Retrieve information about a device's capabilities.

30.3.1 Creating a Normal Presentation Space

The following code fragment shows how to create a presentation space with page units of 0.01 inches and associate it with a printer device context:

```
HAB hab;          /* anchor-block handle         */
HPS hpsNormal;    /* presentation-space handle */
HDC hdcPrinter;   /* device-context handle      */
SIZEL sizlPage;   /* page structure             */
      .
      .
      .
hpsNormal = GpiCreatePS(hab, hdcPrinter, &sizlPage,
   PU_LOENGLISH | GPIA_ASSOC);
```

30.3.2 Creating a Standard Device Context for a Printer

The following code fragment shows how to create a printer device context:

```
HDC hdcPrinter;              /* handle of printer device context */
HAB hab;                     /* anchor-block handle               */
DEVOPENSTRUC dop;            /* device information                */

dop.pszLogAddress = "lpt1";       /* logical-device address */
dop.pszDriverName = "PSCRIPT";    /* device-driver name     */
dop.pdriv = NULL;                 /* pointer to driver data */
dop.pszDataType = "PM_Q_STD";     /* standard queued data   */

hdcPrinter = DevOpenDC(hab,
    OD_DIRECT,                        /* direct device type */
    "*",                              /* no data in os2.ini */
    4L,
    (PDEVOPENDATA) &dop,
    (HDC) NULL);
```

30.3.3 Creating a Standard Device Context for a Metafile

The following code fragment shows how to create a standard device context:

```
HDC hdcMeta, hdcWin; /* handles of metafile and window DC */
HAB hab;             /* anchor-block handle              */
DEVOPENSTRUC dop;    /* device information               */

dop.pszLogAddress = NULL;          /* logical-device address */
dop.pszDriverName = "DISPLAY";     /* device-driver name     */
dop.pdriv = NULL;                  /* pointer to driver data */
dop.pszDataType = NULL;            /* no queued data         */

hdcMeta = DevOpenDC(hab,
    OD_METAFILE,            /* metafile DC               */
    "*",                    /* ignores os2.ini           */
    4L,                     /* uses first 4 fields in dop */
    (PDEVOPENDATA) &dop,    /* structure for system info  */
    NULL);                  /* compatible with screen     */
```

30.3.4 Reassociating a Normal Presentation Space

The following code fragment shows how to break the link between a normal presentation space and a device context and reassociate the presentation space with a metafile device context:

```
if (!GpiAssociate(hps, NULL))    /* breaks link between PS and DC   */
    ReportError(...);
if (!GpiAssociate(hps, hdcMeta)) /* reassociates PS and metafile DC */
    ReportError(...);
```

30.3.5 Associating a Device Context with a Presentation Space

The following code fragment shows how to open a cached device context and associate it with a normal presentation space:

```
HDC hdcWin;           /* cached-device-context handle     */
HPS hpsWin;           /* normal-presentation-space handle */
HWND hwndClient;      /* client-window handle             */
HAB hab;              /* anchor-block handle              */
SIZEL sizlPage;       /* presentation page                */

hdcWin = WinOpenWindowDC(hwndClient);
hpsWin = GpiCreatePS(hab, hdcWin, &sizlPage,
    PU_LOENGLISH | GPIA_ASSOC);
```

30.4 Summary

The following list summarizes the MS OS/2 presentation-space and device-context functions:

DevCloseDC Closes a device context opened by the **DevOpenDC** function.

DevOpenDC Opens a device context for a printer or plotter.

DevQueryCaps Retrieves information about an output device.

GpiAssociate Associates a presentation space with a device context. This function is only valid if the presentation-space type is normal; it is invalid for micro and cached-micro presentation spaces. Remember that a presentation space can be associated with only one device at a time. To associate a presentation space with a second device, you should do the following:

- Disassociate the presentation space and the original device by calling **GpiAssociate** and passing it NULL as the second argument.
- Associate the presentation space with the second device by calling **GpiAssociate**, passing it a handle that identifies the device context of the new device.

GpiCreatePS Creates a normal or a micro presentation space. A normal presentation space is required if your application's output is sent to more than one device or if your application's output is complex. A micro presentation space is more useful if your application's output is sent to a single device and if the output is not complex.

GpiDestroyPS Destroys a presentation space created by the **GpiCreatePS** function. Do not use this function to destroy a presentation space that was retrieved by calling the **WinGetPS** or **WinBeginPaint** function.

GpiQueryPS Retrieves the dimensions of the presentation page.

GpiResetPS Resets the attributes in a presentation space to one of the following states:

- The original state of the presentation space.
- The original state with one exception: Any resources such as regions, loaded color tables, or loaded fonts are saved and will not be destroyed.
- The state of the presentation space prior to the creation of any retained segments.

GpiRestorePS Restores the attributes of a presentation space that were saved by calling the **GpiSavePS** function.

GpiSavePS Saves the attributes of a presentation space onto a special stack.

Chapter
31

Coordinate Spaces and Transformations

31.1 Introduction 405

31.2 About Coordinate Spaces and Transformations 405
- 31.2.1 Single-Coordinate-Space Systems 406
- 31.2.2 Multiple-Coordinate-Space Systems 408
- 31.2.3 Transformations 408
  - 31.2.3.1 The MATRIXLF Structure and Fixed Values... 409
  - 31.2.3.2 Scaling Transformations 410
  - 31.2.3.3 Rotation Transformations.......................... 411
  - 31.2.3.4 Translation Transformations....................... 411
  - 31.2.3.5 Shear Transformations 411
  - 31.2.3.6 Reflection Transformations 412
  - 31.2.3.7 Combining Transformations 412
  - 31.2.3.8 Round-off Error 412
- 31.2.4 World Coordinate Space 413
- 31.2.5 Model Coordinate Space 414
  - 31.2.5.1 World-to-Model-Space Transformations 415
- 31.2.6 Page Coordinate Space 417
  - 31.2.6.1 Model-to-Page-Space Transformations 418
- 31.2.7 Device Coordinate Space 418
  - 31.2.7.1 Page-to-Device-Space Transformations 418

31.3 Using Coordinate Spaces and Transformations 421
- 31.3.1 Setting Convenient Drawing Units 421
- 31.3.2 Zooming a Picture 421
- 31.3.3 Rotating an Object in a Picture 422

31.4 Summary 423

31.1 Introduction

This chapter describes coordinate spaces and transformations. You should also be familiar with the following topics:

- Presentation spaces
- Device contexts
- Retained drawing
- Clipping

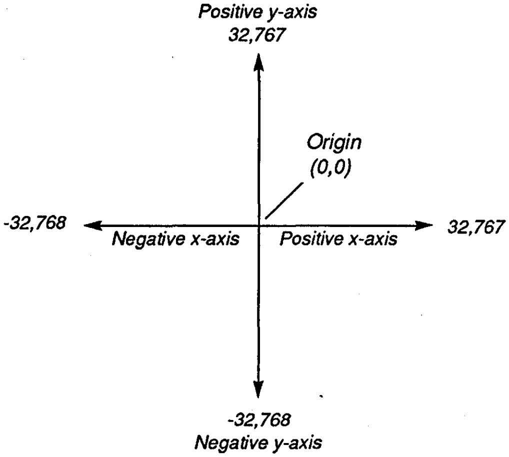
31.2 About Coordinate Spaces and Transformations

A coordinate space is a two-dimensional set of points. A point is the smallest definable location in a coordinate space. There are four coordinate spaces in Microsoft Operating System/2 Presentation Manager:

- World coordinate space
- Model coordinate space
- Page coordinate space
- Device coordinate space

A coordinate system is a means of specifying the location of a point in a coordinate space. For each of the four coordinate spaces in MS OS/2, there is a corresponding coordinate system. The four coordinate systems for MS OS/2 are two-dimensional systems with *x*- and *y*-axes, an origin, and dimensions. The *x*-axis extends horizontally across the system; the *y*-axis, vertically across the system. The origin is located at the intersection of the *x*- and *y*-axes and lies at the center of the system. The world, model, and page coordinate systems' *x*- and *y*-axes extend from 0 to 134,217,727 in the positive direction and from 0 to - 134,217,728 in the negative direction. The device coordinate system's *x*- and *y*-axes extend from 0 to 32,767 in the positive direction and from 0 to - 32,768 in the negative direction. Figure 31.1 shows the device coordinate system, with its origin, axes, and dimensions:

Figure 31.1
Device Coordinate System

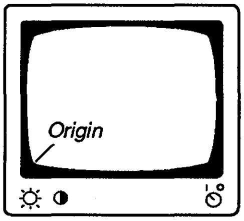


When you specify a coordinate in MS OS/2, the values you use are long integers (32-bits).

31.2.1 Single-Coordinate-Space Systems

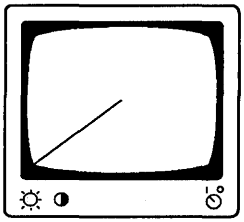
All graphics systems use at least one coordinate space and system to generate output on a video display or printer. The simplest systems use a single coordinate space, whose points are the pels on the display. In single-coordinate-space, or single-space, systems, one corner of the display (usually the lower-left) corresponds to the origin of the coordinate system; the positive *x*-axis extends from the origin to the lower-right corner of the display; and the positive *y*-axis extends from the origin to the upper-left corner of the display. Figure 31.2 shows how the axes correspond to the display:

Figure 31.2
Video Display and Single-Coordinate-Space System



When drawing, applications written for single-coordinate-space systems must specify distances and locations in pels. For example, when the application draws a straight line, it specifies starting and ending points in pels. So if the display measures 100 pels by 100 pels and the application supplies a line-drawing command with a starting point of (0,0) and an ending point of (50,50), the line shown in Figure 31.3 would appear on the display:

Figure 31.3
Diagonal Line in a Single-Coordinate-Space System

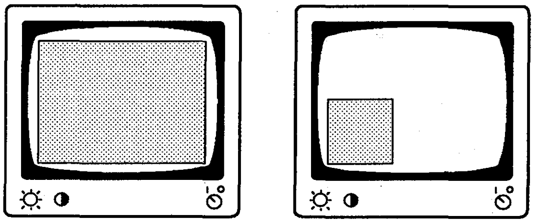


There are two fundamental drawbacks to a graphics system with a single coordinate space:

- Any application written for the graphics system is device dependent.
- It is difficult to draw output in convenient units like centimeters, feet, inches, or twips.

Since the coordinate space in a simple system is made up of pels, and since the shape of pels varies from device to device, the output from an application written for one device may look different on another device. Figure 31.4 demonstrates this principle: The display on the left measures 100 pels by 100 pels, the display on the right 640 pels by 350 pels. In each case, a rectangle has been drawn with a lower-left corner at (10,10) and an upper-right corner at (90,90). Note the different shapes of each rectangle.

Figure 31.4
One Rectangle Displayed on Two Different Devices



Single-coordinate-space systems are difficult to use for drawing or computer-aided-design (CAD) applications since these applications work in units of inches, feet, meters, and so on. For example, to draw a simple horizontal line that is 1 inch long, you must perform the following tasks:

1 Determine the width (in inches) of the video display or printer paper.

2 Determine the width (in pels) of the video display.

3 Divide the width in pels by the width in inches to determine the ratio of pels to inches.

4 Using this ratio, perform the line-drawing command.

31.2.2 Multiple-Coordinate-Space Systems

Single-coordinate-space systems do not provide the flexibility that most applications and application developers need. An application written for a single-space system will create meaningful output only on a device on which the pels are a certain width and height. Since spreadsheet, word-processing, desktop-publishing, and CAD applications normally draw output on video displays *and* printers or plotters, it is obvious that a single-space system is impractical. In Presentation Manager, however, you can avoid the limitations of a single-coordinate-space system by using additional coordinate spaces and other operations, called transformations. Transformations require two coordinate spaces—a source coordinate space and a target coordinate space.

31.2.3 Transformations

When copying objects from one coordinate space to another, MS OS/2 applies one or more of five operations, called transformations, to the points that define the object. These five transformations—scaling, rotation, translation, shear, and reflection—enable you to write applications that can do the following tasks:

- Run on a variety of video displays, printers, and plotters.
- Use convenient units of measurement (centimeters, inches, feet, kilometers, miles, etc.).
- Scroll and zoom pictures.
- Rotate and scale objects in pictures.
- Shift the positions of objects in pictures.
- Display mirror images of objects in pictures.
- Display sheared objects in pictures.

When your application copies an object from the source coordinate space to the target coordinate space, the five transformations work as follows:

- The scaling transformation makes the object look bigger or smaller.
- The rotation transformation rotates the object.
- The translation transformation shifts the object with respect to the origin of the coordinate system.
- The shear transformation rotates either all the vertical or all the horizontal lines in an object.
- The reflection transformation creates a mirror image of an object with respect to the *x*- or *y*-axis.

Presentation Manager uses three-by-three matrices to represent the three transformations. For any point (x,y) in a source coordinate space, you can determine the corresponding point (x',y') in the target coordinate space by using the following matrix multiplication:

$$[x\ y\ 1] \times \begin{bmatrix} M11 & M12 & M22 \\ M21 & M22 & M23 \\ M31 & M32 & M33 \end{bmatrix} = [x'\ y'\ 1]$$

In the following example, the transformations are set to unity (represented by the identity matrix). In such cases, the original point in the source coordinate space is always identical to its corresponding point in the target coordinate space.

$$[3\ 100\ 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = [3\ 100\ 1]$$

In the next example, the identity matrix is replaced with a new matrix that translates the point when the point is copied from the source coordinate space to the target coordinate space:

$$[3\ 100\ 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 97 & -97 & 1 \end{bmatrix} = [100\ 3\ 1]$$

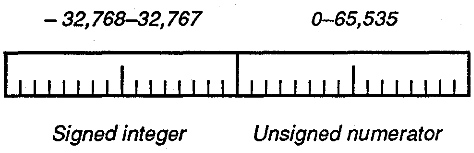
In this example, the point (3,100) was translated to (100,3) by exchanging the values of *M31* and *M32*.

31.2.3.1 The MATRIXLF Structure and Fixed Values

A special data structure called the **MATRIXLF** structure contains nine fields that correspond to the nine elements in a three-by-three transformation matrix. Five of these fields are 32-bit long integer values; the remaining four are special 32-bit fixed values.

A fixed value is a binary representation of a floating-point number. A fixed value has two parts: the high-order 16-bits and the low-order 16-bits. The high-order 16-bits contain a signed integer in the range -32,768 through 32,767; the low-order 16-bits contain the numerator of a fraction, in the range 0 through 65,535 (the denominator for this fraction is 65,536). Figure 31.5 shows the two parts of a fixed value:

Figure 31.5
Fixed Value



You use fixed values to store the sines and cosines of angles for the rotation transformation. You also use fixed values to store scaling factors for the scaling transformation. To store the cosine of 60 degrees in a fixed value, you would multiply 0.5 by 65,536. The result, 32,768, is the value you would assign to a fixed value in the **MATRIXLF** structure. To store a scaling factor of, for example, 3 in a fixed value, you would multiply 3 by 65,536. Again, the result, 196,608, is the value you would assign to a fixed value in the **MATRIXLF** structure.

The **MAKEFIXED** macro provides a quick and convenient method for determining fixed values. This macro requires two arguments: The first is the integer part of the fixed value, and the second is the fraction part of the fixed value. In the following example, **MAKEFIXED** is used to determine the fixed-value equivalent of 1-and-1/8:

```
matlf.fxM11 = MAKEFIXED(1, 8192)
```

The first argument, 1, is the integer part of the fixed value. The second argument, 8192, is the result of multiplying 65,536 by 1/8.

The following type definition shows each of the fields in the **MATRIXLF** structure. Fields with an **fx** prefix contain fixed values; fields with an **l** prefix contain long values.

```
typedef struct _MATRIXLF {      /* matlf */
    FIXED fxM11;        /* M11 */
    FIXED fxM12;        /* M12 */
    LONG  lM13;         /* M13 */
    FIXED fxM21;        /* M21 */
    FIXED fxM22;        /* M22 */
    LONG  lM23;         /* M23 */
    LONG  lM31;         /* M31 */
    LONG  lM32;         /* M32 */
    LONG  lM33;         /* M33 */
} MATRIXLF;
```

31.2.3.2 Scaling Transformations

When you scale an object by using the scaling transformation, the matrix element *M11* contains the horizontal scaling component (*horz*), and the matrix element *M22* contains the vertical scaling component (*vert*):

$$\begin{bmatrix} horz & 0 & 0 \\ 0 & vert & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

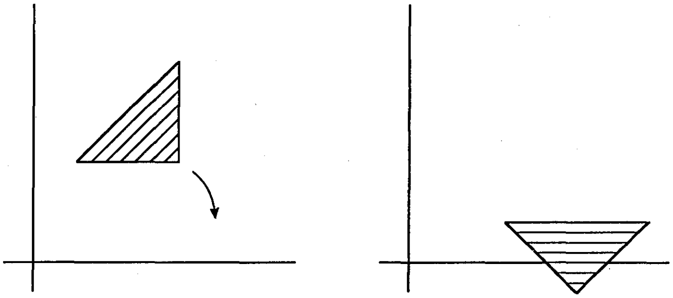
31.2.3.3 Rotation Transformations

When you rotate an object clockwise by using the rotation transformation, the matrix element *M11* contains the cosine of the rotation angle; *M12*, the negative sine of the rotation angle; *M21*, the sine of the rotation angle; and *M22*, the cosine of the rotation angle:

$$\begin{bmatrix} \cos & -\sin & 0 \\ \sin & \cos & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Remember that MS OS/2 applies a transformation to *all* points in the source coordinate space. This means that unless an object is drawn about the origin of the source coordinate space, translation will occur when the object is rotated or scaled. Figure 31.6 shows a triangle, first in its source coordinate space and then in its target coordinate space, rotated 45 degrees clockwise. Note the translation with respect to the origin.

Figure 31.6
Rotating an Object



31.2.3.4 Translation Transformations

When you translate an object by using the translation transformation, the matrix element *M31* contains the horizontal translation component, and the matrix element *M32* contains the vertical translation component, as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ \mathit{horz} & \mathit{vert} & 1 \end{bmatrix}$$

31.2.3.5 Shear Transformations

There are two shear transformations: vertical and horizontal. When an application uses the vertical shear transformation, it affects only the vertical lines in the object. In the matrix of the vertical shear transformation, the element *M21* contains a horizontal shear component, and the element *M22* contains a vertical shear component, as follows:

$$\begin{bmatrix} 1 & 0 & 0 \\ \mathit{horz} & \mathit{vert} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

When an application uses the horizontal shear transformation, it affects only the horizontal lines in the object. In the matrix of the horizontal shear transformation, the element *M11* contains a horizontal shear component, and the matrix element *M12* contains a vertical shear component, as follows:

$$\begin{bmatrix} \mathit{horz} & 0 & 0 \\ \mathit{vert} & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If an application shears an object that contains two orthogonal vectors (two perpendicular lines), the vectors will no longer be orthogonal.

31.2.3.6 Reflection Transformations

The reflection transformation creates a mirror image of an object with respect to the *x*- or the *y*-axis. The matrix element *M11* contains the horizontal reflection component, which causes reflection about the *y*-axis. The matrix element *M22* contains the vertical reflection component, which causes reflection about the *x*-axis. The reflection components are always negative.

$$\begin{bmatrix} \mathit{horz} & 0 & 0 \\ 0 & \mathit{vert} & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

31.2.3.7 Combining Transformations

Each time you call a **Gpi** function that sets a transformation, you can specify whether you want to combine it with existing transformations and, if so, whether you want MS OS/2 to apply the existing transformation before or after the new transformation.

If you want MS OS/2 to replace any existing transformations with the new transformation, use the TRANSFORM_REPLACE flag when you set the transformation. If you want MS OS/2 to apply the new transformation *before* it applies the existing transformation, use the TRANSFORM_PREEMPT flag when you set the new transformation. If you want MS OS/2 to apply the new transformation *after* it applies the existing transformation, use the TRANSFORM_ADD flag when you set the new transformation.

31.2.3.8 Round-off Error

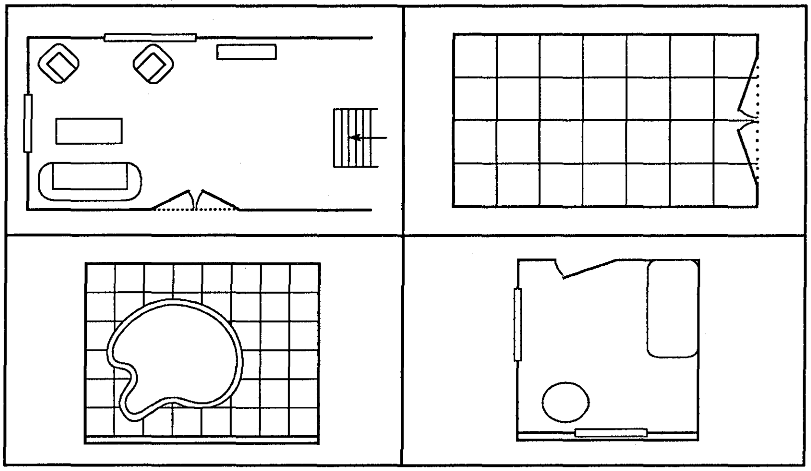
Whenever you use transformations in your application, you should check for any round-off error that occurs after multiple scaling, rotation, shear, or reflection transformations. For example, if your application uses a rotation transformation to rotate the hands of a clock, the accuracy of the clock will diminish due to rounding off after the transformation. In order to prevent the loss of accuracy, your application should check the results of each transformation.

31.2.4 World Coordinate Space

A world coordinate space, or world space, is where all Presentation Manager drawing operations begin. Whenever you draw with one of the line, arc, character, area, or image primitives, the coordinates that you specify are world coordinates. World-coordinate units can be inches, fractions of an inch, meters, centimeters, miles, kilometers, yards, and so on.

If your application uses the retained-drawing mode to store subpictures in chained segments, MS OS/2 assigns a new world space to each segment. In other words, if the segment attribute is set to ATTR_CHAINED and you call the **GpiOpenSegment** function, the subpicture associated with that segment is drawn in a new world space. The following illustration shows the subpictures in four chained segments. Each subpicture is drawn in its own world space.

Figure 31.7
Subpictures in Four Chained Segments in World Space

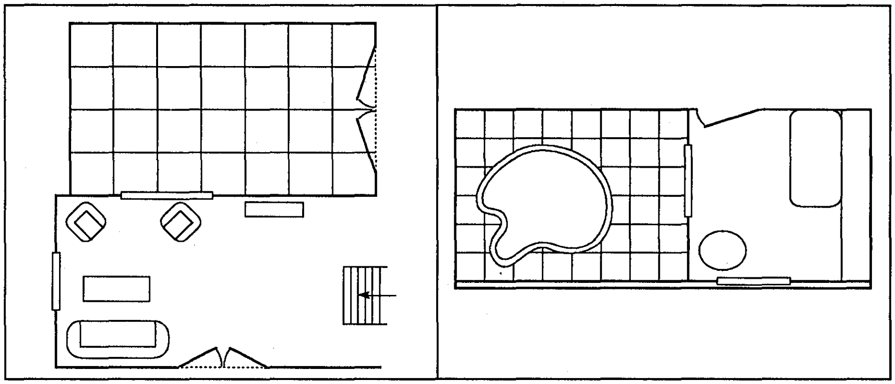


There is a special clipping area called a clip path that you can use to define a part of a world space that you want to copy into the next coordinate space (the model space). The advantage of a clip path is that it is the only clipping region that can be nonrectangular. Its edges can include arcs, curves, and straight lines. The coordinates that define the dimensions and shape of a clip path are always world coordinates. You can create a clip path by calling the **GpiBeginPath**, **GpiEndPath**, and **GpiSetClipPath** functions and one or more **Gpi** primitives that define the dimensions and location of the path. For more information, see Chapter 40, "Clipping."

31.2.5 Model Coordinate Space

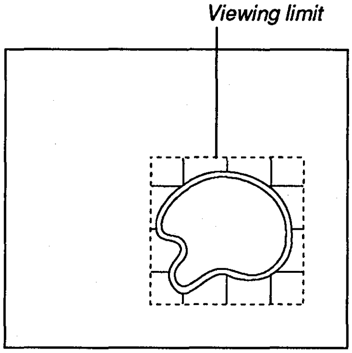
A model coordinate space, or model space, is where you build a model, or a complete picture, from subpictures in one or more world spaces. You can have more than one model space, each corresponding to a particular model. Figure 31.8 shows two model spaces that were defined by calls to the **GpiSetSegmentTransformMatrix** function:

Figure 31.8
Two Model Spaces



There is a special clipping area called a viewing limit that you can use to define a part of a model space that you want to copy into the next coordinate space (the page space). A viewing limit is always rectangular, and the coordinates that define its location and dimensions are always model coordinates. You can set the viewing limit by calling the **GpiSetViewingLimit** function. Figure 31.9 shows a viewing limit set in a model space; the dashed line represents the viewing-limit border:

Figure 31.9
Viewing Limit in Model Space



31.2.5.1 World-to-Model-Space Transformations

There are three transformations that operate between the world space and the model space:

- The model transformation
- The segment transformation
- The instance transformation

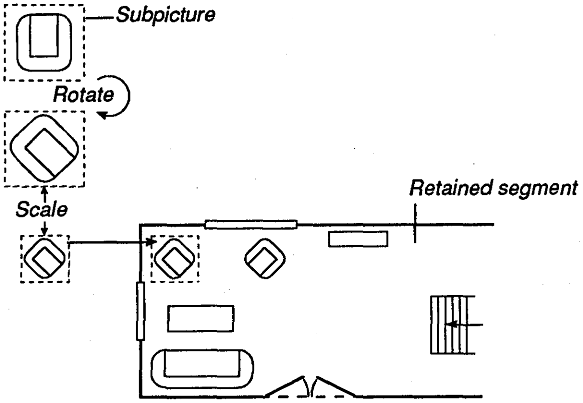
Model Transformations

Model transformations are used for retained- or nonretained-drawing operations. This means model transformations affect the output from the **GpiDrawChain**, **GpiDrawFrom**, and **GpiDrawSegment** functions and any of the **Gpi** functions that occur outside of a retained-drawing segment.

You can determine the values for the current model transformation by calling the **GpiQueryModelTransformMatrix** function, which returns the model transformation's scaling, rotation, and translation values in a three-by-three matrix. Similarly, you can set these values by calling the **GpiSetModelTransformMatrix** function, passing it the scaling, rotation, and translation values in a three-by-three matrix.

In a retained-graphics application, you could use a model transformation to rotate, scale, or translate parts of a subpicture. You can set and reset the model transformation any number of times within a retained-drawing segment. Figure 31.10 shows part of a floor plan as a single subpicture stored in a retained segment. Each chair was drawn by setting a new translation component in the model transformation before calling the **GpiBox** function.

Figure 31.10
Retained Subpicture Drawn Using Model Transformations



Segment Transformations

Segment transformations alter retained-drawing output. Unlike a model transformation, which can be set and reset within a segment bracket, a segment transformation must be set outside of a segment bracket.

You can determine the values for the current segment transformation by calling the **GpiQuerySegmentTransformMatrix** function, which returns the scaling, rotation, and translation values in a three-by-three matrix and passes a segment identifier. Similarly, you can set the values for the segment transformation by calling the **GpiSetSegmentTransformMatrix** function, passing it the necessary values (in a three-by-three matrix) and a segment identifier.

The user of a CAD or drafting application may need to zoom in on part of a subpicture that was clipped in world space by a clip path. The application could perform this operation by setting the segment transformation for that particular subpicture after setting the clip path. Figure 31.11 shows a subpicture in world space that contains a clip path; the dashed line represents the outline of the clip path:

Figure 31.11
Clip Path in World Space

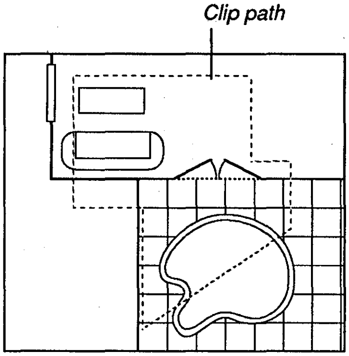
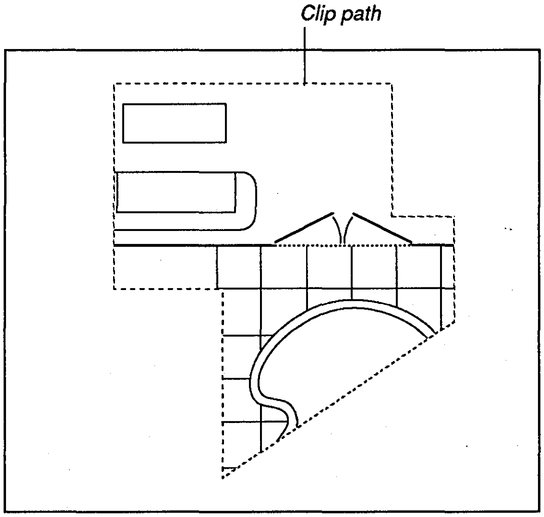


Figure 31.12 shows a zoomed view (in model space) of the clipped part of the subpicture:

Figure 31.12
Scaling the Clipped Part of a Subpicture in Model Space



Instance Transformations

Instance transformations alter the retained-drawing output from special segments referred to as "called" segments. A called segment usually contains a subpicture that is duplicated several times in other subpictures. The instance transformation positions, sizes, and rotates the subpicture each time it is duplicated. You can set the values for the instance transformation by calling the **GpiCallSegmentMatrix** function, passing it the transformation values (in a three-by-three matrix) and a segment identifier.

31.2.6 Page Coordinate Space

A page coordinate space, or page space, is where a complete picture is prepared for viewing in a window on a video display, printing on a page of printer paper, or plotting on a page of plotter paper. Page-coordinate units can be increments of an inch, a meter, several pels, a twip, or some arbitrary value. (A twip is a standard unit in the typesetting industry that measures 1/1440 of an inch.) You specify the units used for page coordinates when you call the **GpiCreatePS** function and create a presentation space.

If your application uses retained-drawing mode, the picture in page space will contain parts of models (or pictures) from each of the model spaces that have not been clipped; the picture will also contain any additional graphics-primitive output (that has not been clipped) that the application generated in nonretained-drawing mode. If your application uses nonretained-drawing mode, the picture in page space will contain all of the graphics-primitive output that has not been clipped.

In the page space, there is a special clipping area called a graphics field that you can use to define the part of the page space that you want to copy into the next coordinate space (the device space). The graphics field is always rectangular. The coordinates that define the location and dimensions of the graphics field are always page coordinates.

31.2.6.1 Model-to-Page-Space Transformations

There are two transformations that operate between the model space and the page space:

- The default viewing transformation
- The viewing transformation

Default Viewing Transformations

Default viewing transformations scroll or zoom pictures in a window on a video display or on a page of printer or plotter paper. You can determine the current values for the default viewing transformation by calling the **GpiQueryDefaultViewMatrix** function, which returns the default-viewing-transformation values in a three-by-three matrix. You can set these values by calling the **GpiSetDefaultViewMatrix** function and passing it the transformation values (in a three-by-three matrix).

Viewing Transformations

Viewing transformations create pictures from multiple model spaces. For example, you could copy a picture from a model space to a page space, leaving all parts of the picture intact. Then you could copy a part of another picture from a second model space by using a viewing limit to define the part of the picture that you want to copy and then using the viewing transformation to scale and translate that part. You can determine the current values for the viewing transformation by calling the **GpiQueryViewingTransformMatrix** function, which returns the values for the viewing transformation in a three-by-three matrix. You can set the values by calling the **GpiSetViewingTransformMatrix** function and passing it the transformation values (in a three-by-three matrix).

31.2.7 Device Coordinate Space

A device coordinate space, or device space, is the final space in which a picture is drawn before it appears in a window or on the page of a printer or plotter. Device-coordinate units are pels if the page units are pels, twips, increments of an inch, or increments of a meter. Device-coordinate units are arbitrary if the page units are arbitrary.

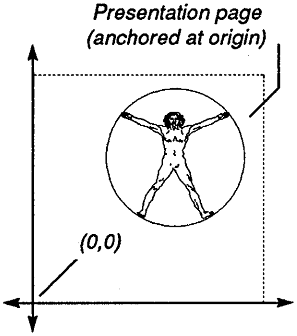
31.2.7.1 Page-to-Device-Space Transformations

There is one transformation between the page space and the device space—it is called the device transformation. Unlike the other transformations that rotate, scale, and translate objects, the device transformation only scales and translates objects; also, instead of a three-by-three matrix, the device transformation uses two rectangles. (The location of the first rectangle is fixed; the location of the second rectangle is movable.) These two rectangles are the presentation page and the page viewport.

Presentation Pages

A presentation page is a rectangle in a page space. Its lower-left corner is always positioned at the origin of the page space. You can determine the dimensions of the presentation page by calling the **GpiQueryPS** function, which returns a pointer to a **SIZEL** structure that contains the page dimensions. If you specify arbitrary page units when you create a presentation space, you must also specify the dimensions of the presentation page. If you specify any other page unit, MS OS/2 automatically sets the dimensions of the presentation page for you. Figure 31.13 shows a presentation page:

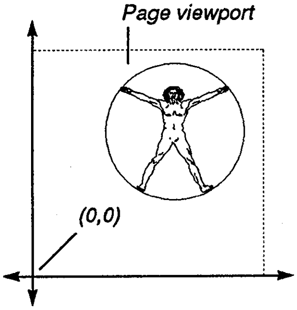
Figure 31.13
Presentation Page in Page Space



Page Viewports

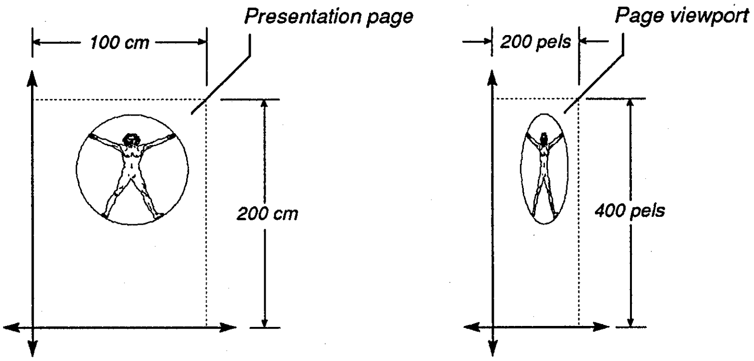
A page viewport is a rectangle in a device space. MS OS/2 always copies the presentation-page rectangle into the page-viewport rectangle. You can determine the current dimensions of the page viewport by calling the **GpiQueryPageViewport** function, which returns a pointer to a **RECTL** structure that contains the coordinates of the viewport. You can set the location and dimensions of the page viewport by calling the **GpiSetPageViewport** function and passing it a pointer to a **RECTL** structure that contains the new values. Figure 31.14 shows the object from Figure 31.13 as it would appear in the page viewport:

Figure 31.14
Page Viewport in Device Space



The ratio of the page width to the page-viewport width defines a horizontal scaling factor, and the ratio of the page height to the viewport height defines a vertical scaling factor. In Figure 31.15, the presentation page measures 100 centimeters by 200 centimeters, and the page viewport measures 200 pels by 400 pels (where each pel measures 0.25 centimeters by 0.5 centimeters):

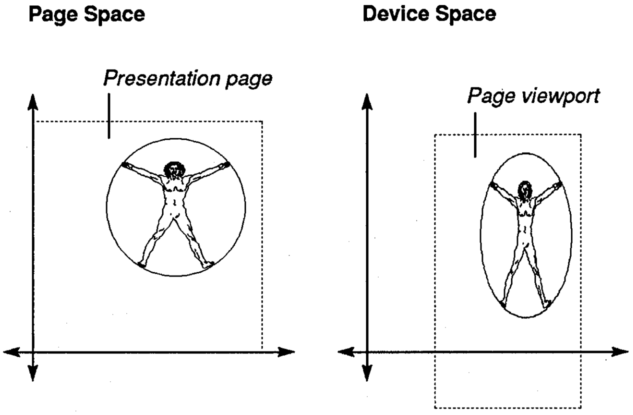
Figure 31.15
Determining Scaling Factors



The image in the page space was not scaled in the device space since each centimeter in the page space mapped to a centimeter in the device space.

Figure 31.16 shows the effect of shifting the page viewport in the device space. Note the translation that this shift causes.

Figure 31.16
Translating the Page Viewport in Device Space



31.3 Using Coordinate Spaces and Transformations

You can perform the following tasks by using the coordinate-space and transformation functions:

- Set an application's drawing units to convenient units.
- Scroll and zoom a picture.
- Rotate, scale, and shift an object in a picture.

31.3.1 Setting Convenient Drawing Units

You can use the **GpiCreatePS** function to set the device transformation so that it uses more convenient page units—for example, centimeters. Follow these steps:

1. If the output device is a screen, open a device context by calling the **WinOpenWindowDC** function. (If the output device is a printer or plotter, open a printer or plotter device context by calling the **DevOpenDC** function.)
2. Create a presentation space by calling the **GpiCreatePS** function, specifying low-metric page units and associating the device context with the presentation space.

The following code fragment demonstrates these steps:

```
HDC hdc;          /* device-context handle       */
HWND hwndClient;  /* client-window handle        */
SIZEL sizlPage;   /* presentation-page rectangle */
HAB hab;          /* anchor-block handle         */
HPS hps;          /* presentation-space handle   */
    .
    .
    .
hdc = WinOpenWindowDC(hwndClient);
sizlPage.cx = 0;
sizlPage.cy = 0;
hps = GpiCreatePS(hab,
    hdc,            /* device-context handle       */
    &sizlPage,      /* address of SIZEL structure  */
    PU_LOMETRIC     /* centimeters as page units   */
    | GPIA_ASSOC);  /* associates window DC with PS */
```

31.3.2 Zooming a Picture

You can use the **GpiSetDefaultViewMatrix** function to zoom a picture. The following code fragment shows how to zoom out to 1/4 or 1/8 scale by using the default viewing transformation:

```
/* Zoom 1/8 */

matlfTransform.fxM11 = MAKEFIXED(0, 8192);
matlfTransform.fxM12 = MAKEFIXED(0, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 0);
matlfTransform.fxM22 = MAKEFIXED(0, 8192);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = 0L;
matlfTransform.lM32 = 0L;
matlfTransform.lM33 = 1L;
GpiSetDefaultViewMatrix(hps, 9L, &matlfTransform, TRANSFORM_REPLACE);
```

```
/* Zoom 1/4 */

matlfTransform.fxM11 = MAKEFIXED(0, 1634);
matlfTransform.fxM12 = MAKEFIXED(0, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 0);
matlfTransform.fxM22 = MAKEFIXED(0, 1634);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = 0L;
matlfTransform.lM32 = 0L;
matlfTransform.lM33 = 1L;
GpiSetDefaultViewMatrix(hps, 9L, &matlfTransform, TRANSFORM_REPLACE);
```

31.3.3 Rotating an Object in a Picture

To rotate an object in a world space by using the model transformation, you must perform the following steps:

1 Translate the object over the coordinate-space origin.

2 Rotate the object.

3 Translate the object back to its original position.

4 Draw the object.

The following code fragment rotates a box 45 degrees:

```
/* translates, rotates 45 degrees, translates */

matlfTransform.fxM11 = MAKEFIXED(1, 0);
matlfTransform.fxM12 = MAKEFIXED(0, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 0);
matlfTransform.fxM22 = MAKEFIXED(1, 0);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = -150L;      /* translates box 150 units left */
matlfTransform.lM32 = -150L;      /* translates box 150 units down */
matlfTransform.lM33 = 1L;
GpiSetModelTransformMatrix(hps, 9L, &matlfTransform,
    TRANSFORM_REPLACE);

matlfTransform.fxM11 = MAKEFIXED(0, 46340);  /* cos 45 * 65536  */
matlfTransform.fxM12 = -MAKEFIXED(0, 46340); /* -sin 45 * 65536 */
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 46350);  /* sin 45 * 65536  */
matlfTransform.fxM22 = MAKEFIXED(0, 46340);  /* cos 45 * 65536  */
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = 0L;
matlfTransform.lM32 = 0L;
matlfTransform.lM33 = 1L;
GpiSetModelTransformMatrix(hps, 9L, &matlfTransform, TRANSFORM_ADD);

matlfTransform.fxM11 = MAKEFIXED(1, 0);
matlfTransform.fxM12 = MAKEFIXED(0, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 0);
matlfTransform.fxM22 = MAKEFIXED(1, 0);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = 150L;    /* shifts back to original pos. */
matlfTransform.lM32 = 150L;    /* shifts back to original pos. */
matlfTransform.lM33 = 1L;
GpiSetModelTransformMatrix(hps, 9L, &matlfTransform, TRANSFORM_ADD);
```

31.4 Summary

The following list summarizes the coordinate-space and transformation functions:

GpiConvert Determines the relationship between coordinates in two coordinate systems when a transformation is in effect. For example, if the page units are low English (0.01 inches) and an application is drawing in world units of feet, the application can call **GpiConvert** to determine the ratio of world units to page units and use this value to scale a picture down before printing.

GpiQueryDefaultViewMatrix Retrieves the scaling, rotation, and translation values for the default viewing transformation. You can use the default viewing transformation to scroll and zoom a picture in a page space.

GpiQueryModelTransformMatrix Retrieves the scaling, rotation, and translation values for the model transformation. You can use the model transformation to alter retained subpictures and nonretained pictures in a world space.

GpiQueryPageViewport Retrieves the dimensions of the page viewport. You can use the page viewport to shrink and expand pictures so that they fill the client area of a window as it shrinks and expands.

GpiQuerySegmentTransformMatrix Retrieves the scaling, rotation, and translation values for the segment transformation. You can use the segment transformation to alter subpictures stored in retained segments in a world space.

GpiQueryViewingTransformMatrix Retrieves the scaling, rotation, and translation values for the viewing transformation. You can use the viewing transformation with the viewing-limit clipping region to transform part of a picture in a model space and then copy it to a page space. You can also use the viewing transformation to transform a complete picture in a model space before copying it to a page space.

GpiSetDefaultViewMatrix Sets the scaling, rotation, and translation values for the default viewing transformation. You can use the default viewing transformation to scroll and zoom a picture in a page space.

GpiSetModelTransformMatrix Sets the scaling, rotation, and translation values for the model transformation. You can use the model transformation to alter retained subpictures and nonretained pictures in a world space.

GpiSetPageViewport Sets the dimensions of the page viewport. You can use the page viewport to shrink and expand pictures so that they fill the client area of a window as it shrinks and expands.

GpiSetSegmentTransformMatrix Sets the scaling, rotation, and translation values for the segment transformation. You can use the segment transformation to alter subpictures stored in retained segments in a world space.

GpiSetViewingTransformMatrix Sets the scaling, rotation, and translation values for the viewing transformation. You can use the viewing transformation and the viewing-limit clipping region to transform part of a picture in a model space and then copy it to a page space. You can also use the viewing transformation to transform a complete picture in a model space before copying it to a page space.

Chapter

32

Line and Arc Primitives

32.1 Introduction 427

32.2 About Line and Arc Primitives 427
32.2.1 Drawing Straight Lines 428
32.2.2 Drawing Arcs 428
32.2.3 Arc Parameters 429
32.2.4 Line Attributes 431

32.3 Using Line and Arc Primitives 432
32.3.1 Drawing a Straight Line 432
32.3.2 Creating a Rubber-Banding Effect with a Straight Line 433
32.3.3 Drawing a Circle 434
32.3.4 Drawing an Ellipse 434
32.3.5 Drawing a Fillet 435
32.3.6 Drawing a Spline 436

32.4 Summary 436

32.1 Introduction

This chapter describes the line and arc primitives. You should also be familiar with the following topics:

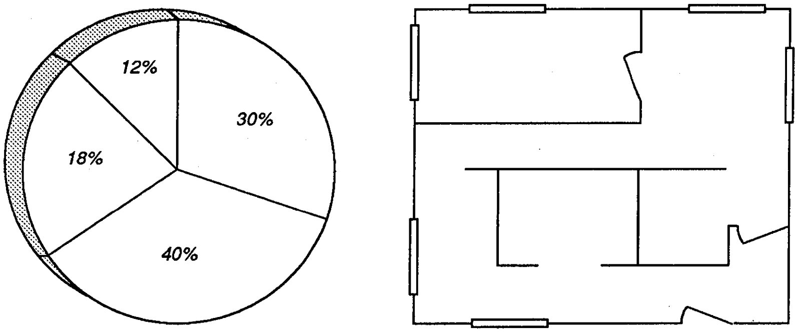
- Presentation spaces and device contexts
- Coordinate spaces and transformations
- Color and mix modes

32.2 About Line and Arc Primitives

Line and arc primitives are straight lines and arcs, respectively, that are drawn by line and arc functions. These primitives are useful for creating pictures that consist of objects such as polygons, circles, fillets, ellipses, and other geometric figures.

Line and arc primitives are useful in many applications. Spreadsheet applications use them to construct pie charts, bar charts, and graphs. Simple drawing applications use them as drawing tools. Computer-aided-design (CAD) applications combine them to draw such complex pictures as schematic diagrams for electrical wiring, blueprints for a building site, and cross-sectional views of machinery. This broad range of pictures, from simple pie charts to complex blueprints, demonstrates how these primitives can benefit your applications. Figure 32.1 shows some sample illustrations that were drawn using line and arc functions:

Figure 32.1
Sample Illustrations Using Line and Arc Functions



In addition to the line and arc functions, there are numerous other functions that you can use to alter the appearance of lines and arcs drawn by the primitive functions. These "other" functions are called attribute functions since they alter the attributes, or characteristics, of lines and arcs.

32.2.1 Drawing Straight Lines

You draw straight lines by calling the **GpiMove** function and either the **GpiLine** or **GpiPolyLine** function. **GpiMove** sets the current position to the starting point of a line; **GpiLine** draws a single line from the current position to a specified point; and **GpiPolyLine** draws a series of connected lines, from the current position through successive points in an array.

When MS OS/2 draws a line, it includes the pels at the starting and ending points of the line. The algorithm used to draw the rest of the line depends on the device driver. For example, a driver for a raster device may use a modified Brezenham algorithm to draw a line, but a driver for a vector device, such as a plotter, simply connects the line's starting and ending points. In all cases, the result is a line primitive that looks the same from device to device.

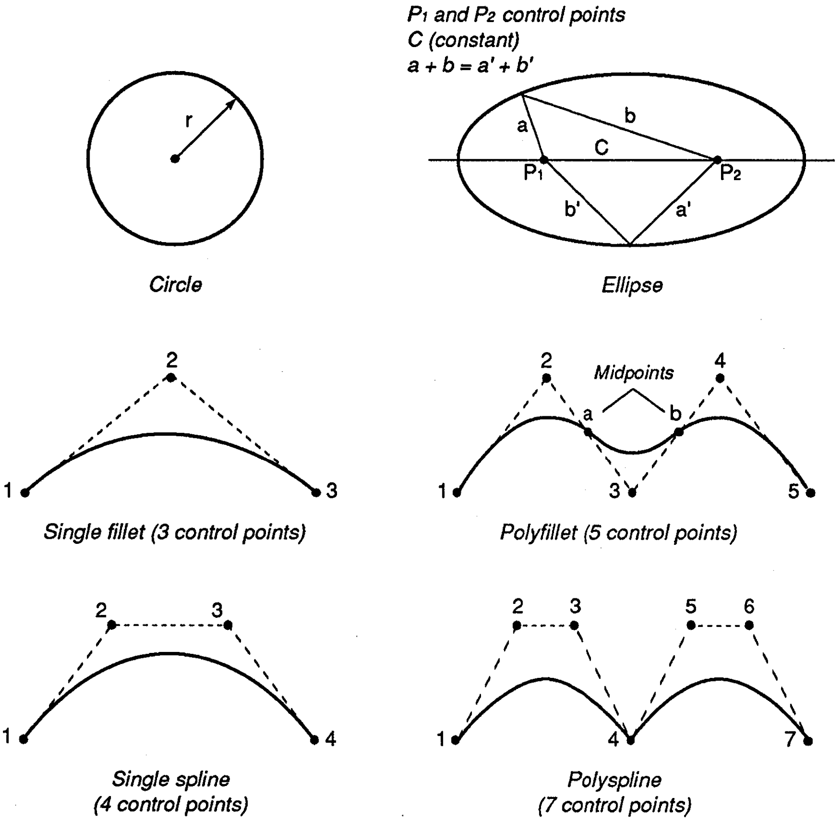
32.2.2 Drawing Arcs

You draw arcs by using **GpiMove** and one of the following functions:

Function	Description
GpiFullArc	Draws a circle or an ellipse.
GpiPartialArc	Draws a section of a circle or ellipse.
GpiPointArc	Draws an arc through three points.
GpiPolyFillet	Draws one or more fillets.
GpiPolyFilletSharp	Draws one or more fillets with varying degrees of sharpness.
GpiPolySpline	Draws one or more splines.

The terms circle, ellipse, fillet, and spline are important in discussing arc primitives. A circle is a closed curve with a center from which every point on the curve is equidistant. An ellipse is a closed curve defined by two fixed points such that the sum of the distances from any point on the curve to the two fixed points is constant. A fillet is a curve that, given three control points, is tangent to the first and last points. A spline is a curve that, given four control points, is tangent to the first and last lines. Figure 32.2 shows a circle, an ellipse, a fillet, and a spline:

Figure 32.2
Arcs



Circle

Ellipse

Single fillet (3 control points)

Polyfillet (5 control points)

Single spline
(4 control points)

Polyspline
(7 control points)

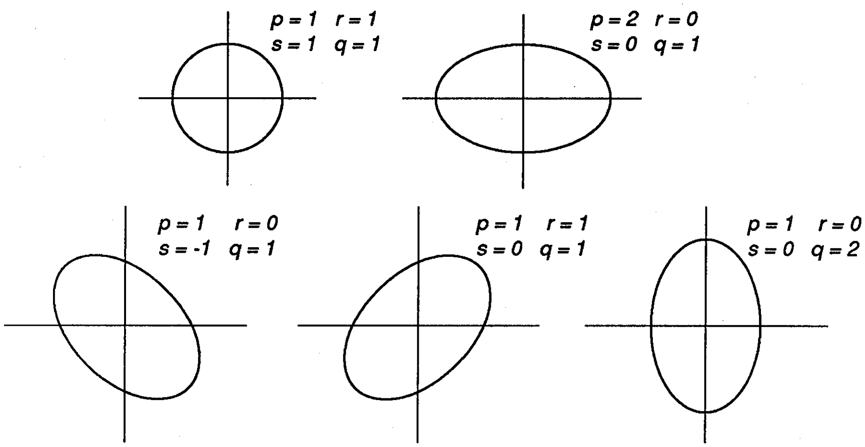
32.2.3 Arc Parameters

When you draw an arc or an ellipse in your application, the image that appears on the drawing surface reflects the current values of the arc parameters, *p*, *q*, *r*, and *s*. The default value of *p* is 1, as is the default value of *q*; the default value of *r* is 0, as is the default value of *s*. These parameters form a two-by-two matrix that scales and shears ellipses and arcs drawn by the **GpiFullArc**, **GpiPointArc**, and **GpiPartialArc** functions:

$$\begin{bmatrix} p=1 & r=0 \\ s=0 & q=1 \end{bmatrix}$$

The parameters *p* and *q* are scaling values: *p* scales in the *x*-direction; *q* scales in the *y*-direction. The parameters *r* and *s* are shear components. When you alter *r* and *s*, a sheared ellipse or arc results. Figure 32.3 shows four ellipses drawn by the **GpiMove** and **GpiFullArc** functions. In each case, one of the default arc parameters has been changed to scale or shear the ellipse.

Figure 32.3
Transforming the Unit Circle Using the Arc Parameters



All of the arc operations begin with a unit circle that lies at the origin in the world coordinate system. The arc parameters define a transformation that is applied to each point on the perimeter of the unit circle. For any point (x,y) on the perimeter of the unit circle, there exists a new point (x',y'), as determined by the following two algorithms:

$$x' = p \times x + r \times y$$
$$y' = s \times x + q \times y$$

If the addition of $(p \times r)$ and $(s \times q)$ is zero, the transformation is orthogonal, and the line from the origin (0,0) to the point (p,s) is either the radius of a circle or half of the major or minor axis of an ellipse. The line from the origin to the point (r,q) is either the radius of a circle or half of the minor or major axis of an ellipse.

Additional scaling transformations in your application can change the shape of the figure accordingly. For instance, if the page units in your application are PU_PELS and the pels on your device are rectangular (rather than square), MS OS/2 will draw an ellipse (rather than a circle) with **GpiFullArc** when the arc parameters are set to their default values.

If $(p \times q)$ is greater than $(r \times s)$, MS OS/2 draws the ellipse counterclockwise for the **GpiFullArc** and **GpiPartialArc** functions. If $(p \times q)$ is less than $(r \times s)$, MS OS/2 draws the ellipse clockwise, and if $(p \times q)$ is equal to $(r \times s)$, MS OS/2 draws a straight line rather than an ellipse.

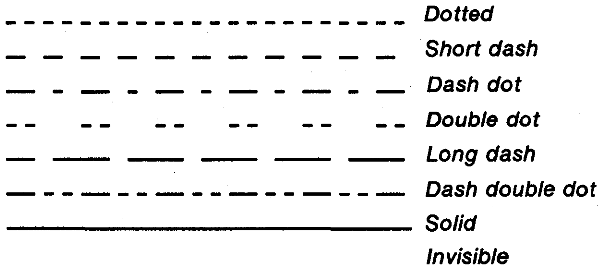
You can determine the current arc parameters by calling the **GpiQueryArcParams** function, which copies the current arc parameters to their corresponding fields in the **ARCPARAMS** structure. You can set the arc parameters by calling **GpiSetArcParams** and passing it a copy of the **ARCPARAMS** structure that contains the new arc parameters you want to use. The **ARCPARAMS** structure has the following form:

```
typedef struct _ARCPARAMS {     /* arcp */
    LONG lP;
    LONG lQ;
    LONG lR;
    LONG lS;
} ARCPARAMS;
```

32.2.4 Line Attributes

Every line and arc has style, width, and color characteristics. These characteristics are called line attributes. The first line attribute, line style, defines the way the line is drawn: solid, as a series of dashes, as a series of dots, or as a combination of dashes and dots. Figure 32.4 shows the various line styles:

Figure 32.4
Line Styles



The second line attribute, line width, defines the width of the line in pels. The third line attribute, line color, defines the color used to draw the line. Usually, this color is mixed with colors already on the drawing surface. This means that the final color of a line or curve depends not only on the line color but on the mix mode and the color of the drawing surface as well. If you draw a dotted or dashed line, the color that appears between the dots or dashes is the current drawing-surface color. The mix mode tells MS OS/2 how to combine the line and background colors (using a bitwise operation on the bits in the corresponding RGB values). For example, one mix mode combines the bits by using the bitwise OR operator. The resulting line has both the line and background colors. Using this mix mode, you would make a purple line when the line color is red and the background color is blue. For more information about color and mix-mode attributes, see Chapter 34, "Color and Mix Modes."

When you create a normal presentation space, the line attributes are set to the following default values:

Line attribute	Default value
Line style	LINETYPE_SOLID
Line color	CLR_NEUTRAL
Line width	1.0
Mix mode	FM_OVERPAINT

You can retrieve the current line-attribute values by calling the **GpiQueryAttrs** function. This function copies the current line attributes to a **LINEBUNDLE** structure. You can change the values at any time by calling the **GpiSetAttrs** function. To make changes, you retrieve the current line attributes, set the appropriate fields in the **LINEBUNDLE** structure to new values, and pass the structure to **GpiSetAttrs**. The **LINEBUNDLE** structure has the following form:

```
typedef struct _LINEBUNDLE {    /* lbnd                       */
    LONG    lColor;             /* line color                 */
    LONG    lReserved;
    USHORT  usMixMode;          /* mix mode                   */
    USHORT  usReserved;
    FIXED   fxWidth;            /* line width                 */
    LONG    lGeomWidth;         /* used with path functions */
    USHORT  usType;             /* line style                 */
    USHORT  usEnd;              /* used with path functions */
    USHORT  usJoin;             /* used with path functions */
} LINEBUNDLE;
```

Several of the fields in this structure apply to a special wide line that you can construct using the path functions. For more information about paths and wide lines, see Chapter 35, "Paths."

32.3 Using Line and Arc Primitives

You can use the line and the arc functions to perform the following tasks:

- Draw a straight line.
- Create a "rubber-banding" effect with straight lines or arcs.
- Draw a circle, ellipse, fillet, or spline.

32.3.1 Drawing a Straight Line

To draw a straight line, you must first set the current position by calling the **GpiMove** or **GpiSetCurrentPosition** function, then draw the line by calling the **GpiLine** function. The following code fragment shows how to draw straight lines:

```
LONG DrawLine(hps, pptlStart, pptlEnd)
HPS hps;            /* presentation-space handle       */
PPOINTL pptlStart;  /* pointer to coordinates of line start */
PPOINTL pptlEnd;    /* pointer to coordinates of line end   */
{
    POINTL ptl_start, ptl_end;    /* point structures           */

    ptl_start.x = pptlStart->x;   /* loads starting x-coordinate */
    ptl_start.y = pptlStart->y;   /* loads starting y-coordinate */
    ptl_end.x = pptlEnd->x;       /* loads ending x-coordinate   */
    ptl_end.y = pptlEnd->y;       /* loads ending y-coordinate   */
    GpiMove(hps, &ptl_start);     /* sets current position       */
    if (GpiLine(hps, &ptl_end))   /* draws line                  */
        return (1L);
    else
        return (OL);
}
```

The second argument to the **GpiMove** function is the address of a structure that contains coordinates of the line's starting point; the second argument to the **GpiLine** function is the address of a structure that contains the coordinates of the last point on the line.

32.3.2 Creating a Rubber-Banding Effect with a Straight Line

When lines are drawn with a rubber-banding effect, two things happen: The original line (if one exists) is erased, and a new line is drawn in its place. This process takes place each time the mouse is dragged and continues until the mouse button is released. The quickest way to erase the original line is to set the foreground mix mode to FM_XOR and redraw the line. The following code fragment demonstrates how you can create this effect:

```
HPS hps;            /* presentation-space handle  */
POINTL ptlStart;    /* starting point of line     */
POINTL ptlNew;      /* ending point of line       */
POINTL ptlPrev;     /* previous end point of line */
BOOL fDraw;         /* line-drawing flag          */
    .
    .
    .
GpiSetColor(hps, CLR_GREEN); /* sets line-drawing color to green */
    .
    .
    .
MRESULT FAR PASCAL GenericWndProc(hwnd, usMessage, mp1, mp2)
HWND hwnd;
USHORT usMessage;
MPARAM mp1;
MPARAM mp2;
{
    .
    .
    .
    case WM_BUTTON1DOWN: /* user begins drawing */
        ptlStart.x = (LONG) (LOUSHORT(mp1));
        ptlStart.y = (LONG) (HIUSHORT(mp1));
        GpiConvert(hps, CVTC_DEVICE, CVTC_WORLD, 1L, &ptlStart);
        ptlPrev.x = ptlStart.x;
        ptlPrev.y = ptlStart.y;
        GpiMove (hps, &ptlStart);
        fDraw = TRUE;
    return TRUE;

    case WM_MOUSEMOVE: /* user draws line */
    if (fDraw) {
        ptlNew.x = (LONG) (LOUSHORT(mp1));
        ptlNew.y = (LONG) (HIUSHORT(mp1));
        GpiConvert(hps, CVTC_DEVICE, CVTC_WORLD, 1L, &ptlNew);
        GpiSetMix(hps, FM_XOR);
        if ((ptlStart.x != ptlPrev.x) || (ptlStart.y != ptlPrev.y)) {
            GpiMove(hps, &ptlStart);
            GpiLine(hps, &ptlPrev);
        }
        if ((ptlStart.x != ptlNew.x) || (ptlStart.y != ptlNew.y)) {
            GpiMove(hps, &ptlStart);
            GpiLine(hps, &ptlNew);
            ptlPrev.x = ptlNew.x; ptlPrev.y = ptlNew.y;
        }
        GpiSetMix(hps, FM_OVERPAINT);
    return TRUE;
    }

    case WM_BUTTON1UP: /* user stops drawing */
        fDraw = FALSE;
        return TRUE;
    .
    .
    .
}
```

32.3.3 Drawing a Circle

To draw a circle, all of the transformations between the world, model, page, and device spaces must maintain square units. This means that instead of pels for page units, the application should select metric or English page units. (On most devices, a pel is rectangular, not square.) This also means that the *x*-values and *y*-values for any scaling transformations should be equal. If the transformations maintain square units, the default arc parameters will transform an ellipse drawn by the **GpiFullArc** function into a circle.

If the page units are Low English and the default transformations are set, the following code fragment draws a circle with a diameter of 1 inch:

```
ARCPARAMS arcp;          /* structure for arc parameters    */
HPS hps;                 /* presentation-space handle       */
POINTL ptlPos;           /* structure for current position */
FIXED fxMult;            /* multiplier for circle           */
    .
    .
    .
    arcp.lP = 1L; arcp.lQ = 1L;
    arcp.lR = OL; arcp.lS = OL;
    GpiSetArcParams(hps,
        &arcp);                     /* sets parameters to default */
    ptlPos.x = 100;                 /* loads x-coordinate         */
    ptlPos.y = 100;                 /* loads y-coordinate         */
    GpiMove(hps, &ptlPos);          /* sets current position      */
    fxMult = (50 * 65536);          /* sets multiplier            */
    GpiFullArc(hps, DRO_OUTLINE,    /* draws circle               */
        fxMult);
```

The second argument to the **GpiFullArc** function, DRO_OUTLINE, specifies that MS OS/2 should draw only the outline of the circle (rather than filling the interior with the current fill pattern). The third argument, fxMult, specifies that MS OS/2 should multiply the size of the circle by 50 units. Since the page units are PU_LOENGLISH and the default transformations are set, 50 units are equivalent to 1/2 inch. The circle will have a 1/2-inch radius and a 1-inch diameter.

32.3.4 Drawing an Ellipse

If the world, model, page, and device transformations are set so that they maintain square units, you can use the arc parameters to transform the shape of the ellipse drawn by the **GpiFullArc** function. The following code fragment alters the arc parameter *p* by doubling its value, making the ellipse twice as wide horizontally as it is vertically.

If the page units are PU_LOENGLISH and the default transformations are set, the following code fragment draws an ellipse with a 2-inch major axis (parallel to the *x*-axis) and a 1-inch minor axis (parallel to the *y*-axis). The ellipse is centered over a point in the lower-left corner of a maximized window.

```
ARCPARAMS arcp;      /* structure for arc parameters    */
HPS hps;             /* presentation-space handle       */
POINTL ptlPos;       /* structure for current position */
LONG fxMult;         /* multiplier for ellipse          */
    .
    .
    .
    arcp.lP = 2L; arcp.lQ = 1L;
    arcp.lR = OL; arcp.lS = OL;
    GpiSetArcParams(hps,
        &arcp);                    /* sets parameters to default */
    ptlPos.x = 200;                /* loads x-coordinate          */
    ptlPos.y = 100;                /* loads y-coordinate          */
    GpiMove(hps, &ptlPos);         /* sets current position       */
    fxMult = (50 * 65536);         /* sets multiplier             */
    GpiFullArc(hps, DRO_OUTLINE,   /* draws circle                */
        fxMult);
    .
    .
    .
```

The arc-parameter field **IP** is set to 2, and the arc-parameter field **IQ** is set to 1. From these parameters, MS OS/2 creates an ellipse with a major axis that is twice as long as the minor axis.

32.3.5 Drawing a Fillet

When you draw a fillet, each curve is tangent to two lines. The curve of the first fillet is always tangent to a line drawn between the current position and the first control point and a line drawn between the current position and the second control point.

The following code fragment shows how to draw a single curve by using the current position and two control points:

```
POINTL aptl[2];  /* structure for control points */
HPS hps;         /* presentation-space handle    */
    .
    .
    .
    aptl[0].x = 50;       /* loads x-coord. of first control point  */
    aptl[0].y = 50;       /* loads y-coord. of first control point  */
    GpiMove(hps, aptl);   /* sets current position                  */
    aptl[0].x = 75;       /* loads x-coord. of second control point */
    aptl[0].y = 75;       /* loads y-coord. of second control point */
    aptl[1].x = 100;      /* loads x-coord. of third control point  */
    aptl[1].y = 50;       /* loads y-coord. of third control point  */
    GpiPolyFillet(hps,    /* draws fillet                           */
        2L, aptl);
    .
    .
    .
```

When you draw a sharp fillet, the sharpness value controls the shape of the curve: If the value is greater than 1, a hyperbola is drawn; if the value is 1, a parabola is drawn; and if the value is less than 1, an ellipse is drawn. The following code fragment uses a sharpness value greater than 1, which creates a hyperbolic curve:

```
POINTL aptl[2];        /* structure for control points */
FIXED fxSharpness[1];  /* array with sharpness value   */
HPS hps;               /* presentation-space handle    */
    .
    .
    .
    aptl[0].x = 50;     /* loads x-coord. of first control point  */
    aptl[0].y = 50;     /* loads y-coord. of first control point  */
    GpiMove(hps, aptl); /* sets current position                  */
    aptl[0].x = 75;     /* loads x-coord. of second control point */
    aptl[0].y = 75;     /* loads y-coord. of second control point */
    aptl[1].x = 100;    /* loads x-coord. of third control point  */
    aptl[1].y = 50;     /* loads y-coord. of third control point  */
    fxSharpness[0] = 1; /* sets sharpness value                   */
    GpiPolyFilletSharp(hps, /* draws fillet                       */
        2L, aptl, fxSharpness);
    .
    .
    .
```

32.3.6 Drawing a Spline

When you use the **GpiPolySpline** function to draw a spline, each curve is tangent to the first and last lines of three intersecting lines. The following code fragment shows how to draw a spline:

```
POINTL aptl[3];  /* structure for control points */
HPS hps;         /* presentation-space handle    */
    .
    .
    .
    aptl[0].x = 50;        /* loads x-coord. of first control point  */
    aptl[0].y = 50;        /* loads y-coord. of first control point  */
    GpiMove(hps, aptl);    /* sets current position                  */
    aptl[0].x = 75;        /* loads x-coord. of second control point */
    aptl[0].y = 75;        /* loads y-coord. of second control point */
    aptl[1].x = 100;       /* loads x-coord. of third control point  */
    aptl[1].y = 75;        /* loads y-coord. of third control point  */
    aptl[2].x = 125;       /* loads x-coord. of fourth control point */
    aptl[2].y = 50;        /* loads y-coord. of fourth control point */
    GpiPolySpline(hps,     /* draws spline                           */
        3L, aptl);
    .
    .
    .
```

32.4 Summary

The following list summarizes the MS OS/2 line and arc functions:

GpiFullArc Draws an ellipse by translating the ellipse generated by the arc parameters to the current position and scaling that ellipse with a specified multiplier. If the arc parameters are set to their default values and the page units are LOENGLISH, HIENGLISH, LOMETRIC, or HIMETRIC, the function draws a circle. Depending on the control argument, this function will fill the ellipse with the current fill pattern, draw the ellipse's outline only, or fill and draw the outline.

GpiLine Draws a straight line from the current position to a specified end point.

GpiMove Sets the current position. MS OS/2 does not save the old position (as it does for **GpiSetCurrentPosition**) if the attribute mode is AM_PRESERVE.

GpiPartialArc Draws a straight line from the current position to the starting point of an arc and then draws the arc itself, using the current arc parameters, a sweep angle, and a multiplier. The arc parameters determine the shape of the arc and the direction in which it is drawn; the sweep angle and multiplier determine the length of the arc.

GpiPointArc Draws an arc through three points. The first point is the current position; the last two points are control points passed as arguments to the function. The function uses the current arc parameters to determine the size and shape of the arc.

GpiPolyFillet Draws a fillet. A fillet is a special curve that does not fit the circumference of an ellipse. (All of the other curves drawn by the various arc functions fit onto the circumference of an ellipse generated by the arc parameters.) A minimum of three control points defines a fillet.

GpiPolyFilletSharp Draws a special fillet, using an array of sharpness values that correspond to each curve in the fillet. Sharpness values less than zero generate curves resembling hyperbolas; values of zero generate straight lines; and values greater than zero generate curves resembling parts of ellipses.

GpiPolyLine Draws a series of straight lines starting at the current position. The attributes in the current line structure determine the style and color of lines.

GpiPolySpline Draws a spline. A spline consists of one or more special curves called Bezier curves. A minimum of four control points defines a spline.

GpiQueryArcParams Retrieves a pointer to a structure that contains the current arc parameters. MS OS/2 uses the arc parameters when it draws an arc using the **GpiFullArc**, **GpiPointArc**, and **GpiPartialArc** functions.

GpiQueryCurrentPosition Retrieves a pointer to a structure that contains the *x*- and *y*-coordinates of the current position.

GpiQueryLineType Retrieves the current line type, which can be one of nine possible values.

GpiQueryLineWidth Retrieves the current line width. Currently in MS OS/2, this value should always be 65,536 decimal (0x1000 hexadecimal).

GpiSetArcParams Sets the parameters of the arc. These parameters determine the shape of an ellipse and the direction in which MS OS/2 draws it. There are four arc parameters—*p*, *s*, *r*, and *q*—represented by four fields in the **ARCPARAMS** structure.

GpiSetCurrentPosition Sets the current position to a point specified by an *x*- and a *y*-coordinate. MS OS/2 uses the current position when it draws lines, arcs, fillets, and splines.

GpiSetLineType Sets the current line type, which can be one of nine possible values.

GpiSetLineWidth Sets the current line width. Currently in MS OS/2, this value should always be 65,536 decimal (0x1000 hexadecimal).

Chapter

33

Fonts and Character Primitives

33.1 Introduction 441

33.2 About Fonts and Character Primitives 441
33.2.1 Font Metrics 442
33.2.2 Image and Outline Fonts 450
33.2.3 Proportional and Fixed Fonts 450
33.2.4 Glyphs, Code Pages, and Code Points 451
33.2.5 Text Output 451
33.2.5.1 Text and Character Attributes 452
33.2.5.2 Character Modes 453
33.2.6 Font Files and Dynamic-Link Libraries 456
33.2.7 Selecting New Fonts 457
33.2.7.1 Public Fonts 457
33.2.7.2 Private Fonts 458

33.3 Using Fonts and Character Primitives 459
33.3.1 Selecting a Public Font 460
33.3.2 Drawing Text 461
33.3.3 Transforming Text from an Outline Font 461
33.3.4 Transforming Text from an Image Font 462

33.4 Summary 462

33.1 Introduction

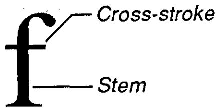
This chapter defines typographic terms and concepts that are part of the MS OS/2 application programming interface (API). You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Coordinate spaces and transformations
- Color and mix modes

33.2 About Fonts and Character Primitives

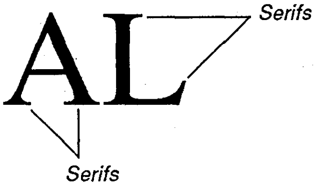
A font family is a collection of fonts that share common stroke-width and serif characteristics. The term stroke width refers to the width of lines used to draw characters and symbols from a font. Figure 33.1 shows a lowercase letter *f* that consists of two strokes: a stem (the main vertical stroke) and a cross-stroke:

Figure 33.1
Strokes



A serif is a short cross-line drawn at the ends of main strokes that form a character or symbol. Figure 33.2 shows serifs drawn at the ends of the strokes in the uppercase letters *A* and *L*:

Figure 33.2
Serifs



A font, part of a font family, is a collection of characters and symbols that share a common height, line weight, and appearance. The height of a font is specified in printer's points, or points, a point being a typographic unit of measurement equal to 1/72 of an inch. The five categories of line weight and appearance are as follows:

- Normal
- Bold
- Condensed
- Expanded
- Italic

Characters from a bold font are drawn with a heavier line weight. A 16-point Times Roman Bold is an example of a bold font, as shown in Figure 33.3:

Figure 33.3
Bold Font

This is a sample of 16-point Times Roman Bold.

Characters from an italic font are drawn with a normal line weight and slanted up and to the right. A 10-point Courier Italic is an example of an italic font, as shown in Figure 33.4:

Figure 33.4
Italic Font

This is a sample of 10-point Courier Italic.

Characters from a normal font are drawn with a normal line weight. An 8-point Helvetica is an example of a normal font, as shown in Figure 33.5:

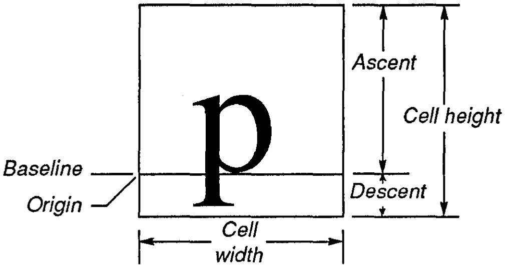
Figure 33.5
Normal Font

This is a sample of 8-point Helvetica.

33.2.1 Font Metrics

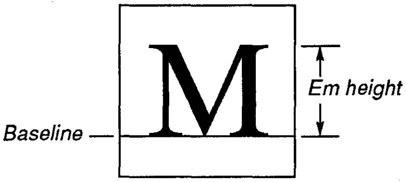
Every character in a font is drawn within a rectangular region called a character cell. Through the lower half of the character cell is drawn an imaginary horizontal line called the baseline. All uppercase letters and most lowercase letters in a given font rest on the baseline. Some lowercase letters, such as *g* or *y*, descend below the baseline. MS OS/2 uses the baseline to position characters. When an application draws a string of text, MS OS/2 positions the leftmost point of the baseline over a predetermined point for each character in the string. The distance from the bottom of the character cell to its top is the character-cell height, and the distance from the baseline to the top of the character cell is the character-cell ascent. Similarly, the character-cell descent is the distance from the baseline to the bottom of the character cell, and the width of the character cell is the distance from one side to the other. Figure 33.6 shows a character cell, its origin, baseline, height, ascent, descent, and width:

Figure 33.6
Character Cell



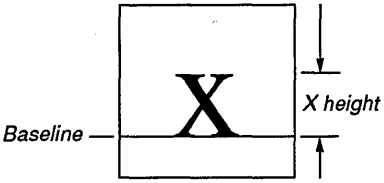
The average distance from the baseline to the top of any uppercase character is called a font's em height. This measurement was given its name because the height of an uppercase letter *M* is usually equal to the average height of all uppercase characters in the font. Figure 33.7 shows the em height:

Figure 33.7
Em Height



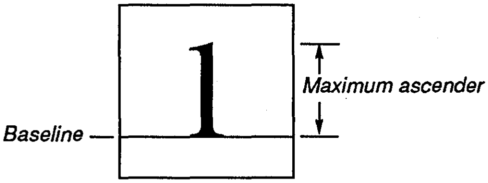
The average distance from the baseline to the top of any lowercase character is called a font's x height. This measurement was given its name because the height of a lowercase letter *x* is usually equal to the average height of all lowercase characters in the font. Figure 33.8 shows the x height:

Figure 33.8
X Height



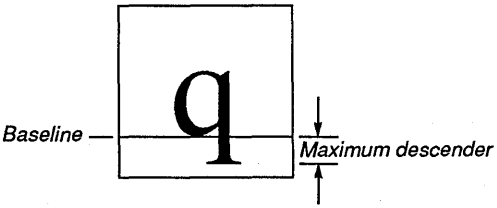
The maximum ascender, which is the height of the tallest character in a font, is shown in Figure 33.9:

Figure 33.9
Maximum Ascender



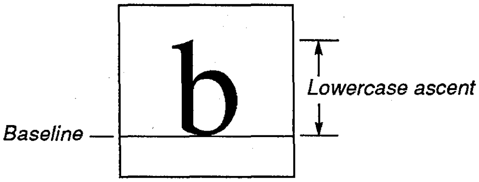
The maximum descender, which is the depth (below the baseline) of the lowest character in a font, is shown in Figure 33.10:

Figure 33.10
Maximum Descender



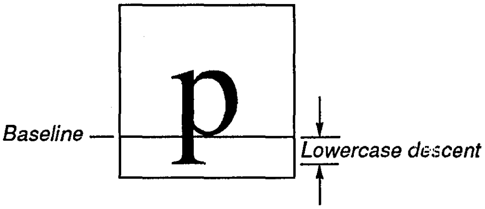
The lowercase ascent, which is the height of the tallest lowercase character in a font, is shown in Figure 33.11:

Figure 33.11
Lowercase Ascent



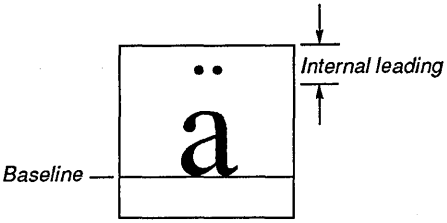
The lowercase descent, which is the depth (below the baseline) of the lowest lowercase character in a font, is shown in Figure 33.12:

Figure 33.12
Lowercase Descent



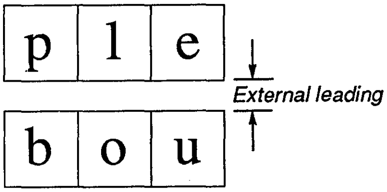
Many fonts reserve part of the space in the top of each character cell for accent marks. This reserved space, called internal leading, is shown in Figure 33.13:

Figure 33.13
Internal Leading



Some fonts designate a certain amount of space to leave between rows of text. This space, called external leading, is not included in the character-cell height or ascent measurements. It is shown in Figure 33.14:

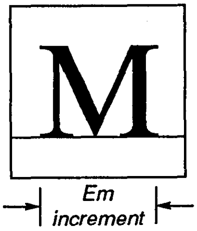
Figure 33.14
External Leading



The average character width is determined by multiplying the width of each lowercase letter by a predetermined factor, adding the results for each letter in the alphabet, and then dividing by 1000. The average character width is determined by the setting in the **lAveCharWidth** field in the **FONTMETRICS** structure. For information about **FONTMETRICS**, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

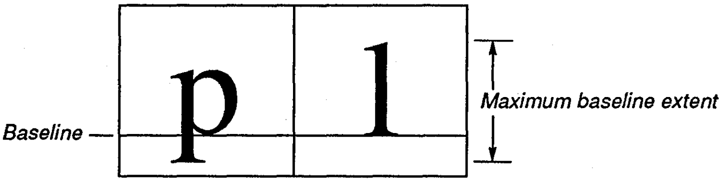
The em increment, which is the width of the uppercase letter *M* in a font, is shown in Figure 33.15:

Figure 33.15
Em Increment



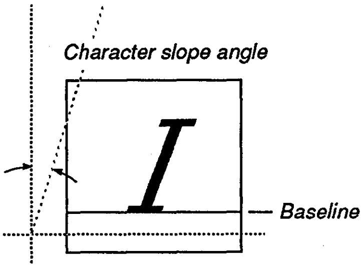
The maximum baseline extent for a font is the sum of the maximum ascender and the maximum descender. Figure 33.16 shows an example of a maximum baseline extent:

Figure 33.16
Maximum Baseline Extent



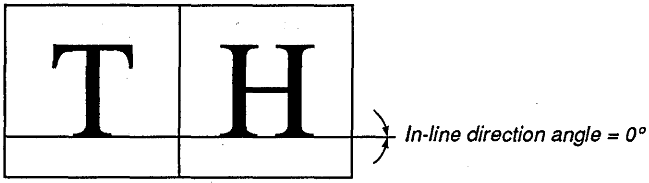
The character slope is an angle measured clockwise with respect to a vertical line. The slope of a normal font is zero; the slope of an italic font is nonzero. Figure 33.17 shows a character slope:

Figure 33.17
Character Slope



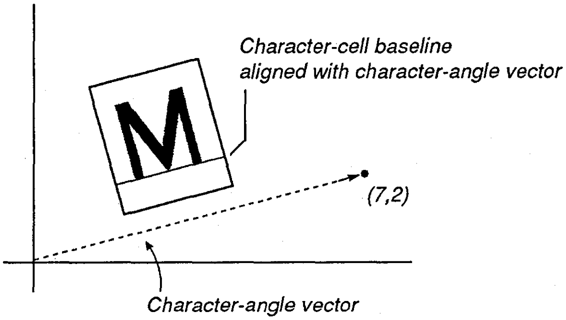
The in-line direction is an angle measured clockwise with respect to a horizontal line. The in-line direction for a Swiss, Helvetica, or Roman font is normally zero; the in-line direction for a Hebrew font is normally 180. Figure 33.18 shows the in-line direction for a Roman font:

Figure 33.18
In-line Direction



The character-rotation angle is an angle measured counterclockwise with respect to a horizontal line. The baselines of characters are aligned with a vector drawn at the angle of rotation. Figure 33.19 shows a character-rotation angle and a character-angle vector:

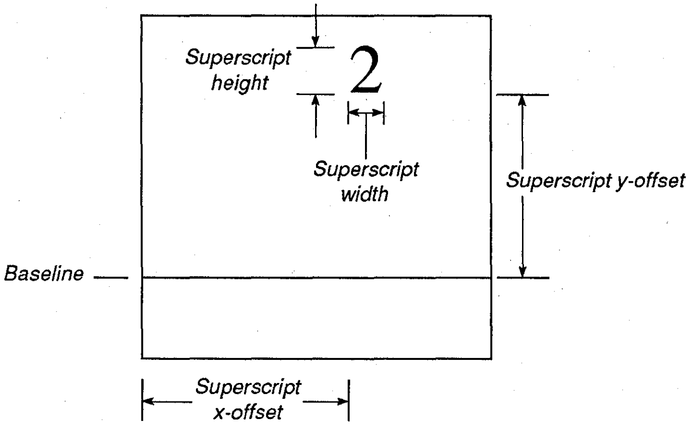
Figure 33.19
Character Rotation



The weight class specifies the thickness of each stroke that forms part of each character in a font.

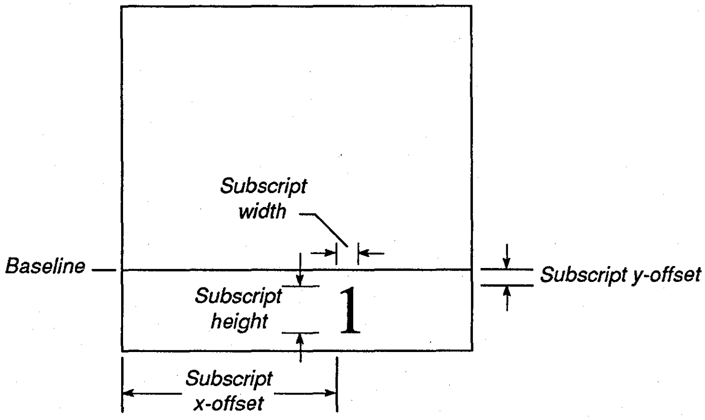
A superscript is a character drawn immediately above and to the right of a normal character in a string of text. Superscripts are identified by a width and a height, and by vertical and horizontal offsets. Figure 33.20 shows a superscript:

Figure 33.20
Superscript



A subscript is a character drawn immediately below and to the right of a normal character in a string of text. Subscripts are identified by a width and a height, and by vertical and horizontal offsets. Figure 33.21 shows a subscript:

Figure 33.21
Subscript



Kerning is an adjustment to space between certain characters in a font. Some fonts contain a kerning table, which is a table of kerning values specifying the amount of space that should appear between certain characters. You can examine the kerning information by calling the **GpiQueryKerningPairs** function. You cannot set kerning for a font, but you can simulate kerning for an image font, using MS OS/2 Font Editor, by adjusting the *a*-space and *c*-space for each character. The *a*-space is the space between the left edge of a character cell and the left edge of the cell's character. The *c*-space is the space between the right edge of a character cell and the right edge of its character.

Most of the terms described in the previous pages have corresponding fields in a special structure called a **FONTMETRICS** structure. This structure has the following form:

```
typedef struct _FONTMETRICS {    /* fm */
    CHAR    szFamilyname[FACESIZE];
    CHAR    szFacename[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lEmHeight;
    LONG    lXHeight;
    LONG    lMaxAscender;
    LONG    lMaxDescender;
    LONG    lLowerCaseAscent;
    LONG    lLowerCaseDescent;
    LONG    lInternalLeading;
    LONG    lExternalLeading;
    LONG    lAveCharWidth;
    LONG    lMaxCharInc;
    LONG    lEmInc;
    LONG    lMaxBaselineExt;
    SHORT   sCharSlope;
    SHORT   sInlineDir;
    SHORT   sCharRot;
    USHORT  usWeightClass;
    USHORT  usWidthClass;
    SHORT   sXDeviceRes;
    SHORT   sYDeviceRes;
    SHORT   sFirstChar;
    SHORT   sLastChar;
    SHORT   sDefaultChar;
    SHORT   sBreakChar;
    SHORT   sNominalPointSize;
    SHORT   sMinimumPointSize;
    SHORT   sMaximumPointSize;
    USHORT  fsType;
    USHORT  fsDefn;
    USHORT  fsSelection;
    USHORT  fsCapabilities;
    LONG    lSubscriptXSize;
    LONG    lSubscriptYSize;
    LONG    lSubscriptXOffset;
    LONG    lSubscriptYOffset;
    LONG    lSuperscriptXSize;
    LONG    lSuperscriptYSize;
    LONG    lSuperscriptXOffset;
    LONG    lSuperscriptYOffset;
    LONG    lUnderscoreSize;
    LONG    lUnderscorePosition;
    LONG    lStrikeoutSize;
    LONG    lStrikeoutPosition;
    SHORT   sKerningPairs;
    SHORT   sReserved;
    LONG    lMatch;
} FONTMETRICS;
```

For a complete description of each field in the **FONTMETRICS** structure, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

33.2.2 Image and Outline Fonts

Characters in a font are stored either as bitmaps or as collections of calls to line, arc, and path functions. Fonts stored as bitmaps are image fonts. Fonts stored as collections of line, arc, and path calls are outline fonts. Figure 33.22 shows an enlarged view of a lowercase letter *a* from an image font:

Figure 33.22
Image Font

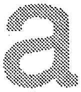
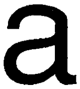


Figure 33.23 shows an enlarged view of a lowercase letter *a* from an outline font:

Figure 33.23
Outline Font



33.2.3 Proportional and Fixed Fonts

When text is drawn, MS OS/2 aligns each character by positioning its character cell next to the previous character's cell, as shown in Figure 33.24:

Figure 33.24
Character-Cell Alignment

T	h	i	s		i	s	

Some fonts adjust the width of character cells so that narrow letters like a lowercase *l* or *i* appear closer to adjacent characters. Fonts that adjust the width of character cells are called proportional fonts; fonts that do not are called fixed fonts. Proportional fonts are generally easier to read than fixed fonts. Figure 33.25 shows a line of text from a proportional font, followed by a line of text from a fixed font:

Figure 33.25
Proportional and Fixed Fonts

This is proportional.

```
This is fixed.
```

33.2.4 Glyphs, Code Pages, and Code Points

The image or picture that you associate with each character or symbol in a font is a glyph. A set of glyphs is called a code page. Each code page contains 256 code points (16-bit integers), in the range 0 through 255, that correspond to the glyphs in that code page. MS OS/2 assigns unique identifiers to each of its code pages. A common code page is code page 437. Figure 33.26 shows the glyphs and corresponding code points for this code page:

Figure 33.26
Code Page 437

Hex Digits 1st → 2nd ↓	0-	1-	2-	3-	4-	5-	6-	7-		8-	9-	A-	B-	C-	D-	E-	F-
-0		►		0	@	P	`	p		Ç	É	á	░	└	╨	α	≡
-1	☺	◄	!	1	A	Q	a	q		ü	æ	í	▒	┴	╤	β	±
-2	☻	↕	"	2	B	R	b	r		é	Æ	ó	▓	┬	╥	Γ	≥
-3	♥	‼	#	3	C	S	c	s		â	ô	ú	│	├	╙	π	≤
-4	♦	¶	$	4	D	T	d	t		ä	ö	ñ	┤	─	╘	Σ	⌠
-5	♣	§	%	5	E	U	e	u		à	ò	Ñ	╡	┼	╒	σ	⌡
-6	♠	▬	&	6	F	V	f	v		å	û	ª	╢	╞	╓	µ	÷
-7	•	↨	'	7	G	W	g	w		ç	ù	º	╖	╟	╫	τ	≈
-8	◘	↑	(	8	H	X	h	x		ê	ÿ	¿	╕	╚	╪	Φ	°
-9	○	↓	)	9	I	Y	i	y		ë	Ö	⌐	╣	╔	┘	Θ	•
-A	◙	→	•	:	J	Z	j	z		è	Ü	¬	║	╩	┌	Ω	•
-B	♂	←	+	;	K	[	k	{		ï	¢	½	╗	╦	█	δ	√
-C	♀	∟	,	<	L	\	l	|		î	£	¼	╝	╠	▄	∞	ⁿ
-D	♪	↔	-	=	M	]	m	}		ì	¥	¡	╜	═	▌	φ	²
-E	♫	▲	.	>	N	^	n	~		Ä	Pt	«	╛	╬	▐	ε	■
-F	☼	▼	/	?	O	_	o	⌂		Å	ƒ	»	┐	╧	▀	∩	

Each code page contains four special code points: a first character, a last character, a default character, and a break character. The default character is the one that appears in text when an application specifies a code point that does not lie between the first and last character code points. The break character is the space character and often has the same code point as the default character.

If the current font is the default system font, you can determine the current code page by calling the **GpiQueryCp** function, or you can assign a new code page by calling the **GpiSetCp** function. For more information about code pages, see the *Microsoft Operating System/2 Programmer's Reference, Volume 3.*

33.2.5 Text Output

Text output is alphanumeric output drawn with characters and symbols from a font. In MS OS/2, fonts are stored either in memory or on devices. Applications access them through a device context associated with the current presentation space. When you create a presentation space by calling the **GpiCreatePS** or **WinGetPS** function, MS OS/2 assigns the presentation space a default font from one of the fonts available either in memory or on the associated device. You can retrieve information about this default font by calling the **GpiQueryFontMetrics** function.

33.2.5.1 Text and Character Attributes

There are five character attributes that affect the appearance of your application's text output. These character attributes are listed as follows:

- Character color
- Character-color mix mode
- Character angle
- Character shear
- Character box

There are two character colors: foreground and background. The foreground color is the color of the strokes in each character. The background color is the color that appears behind the character.

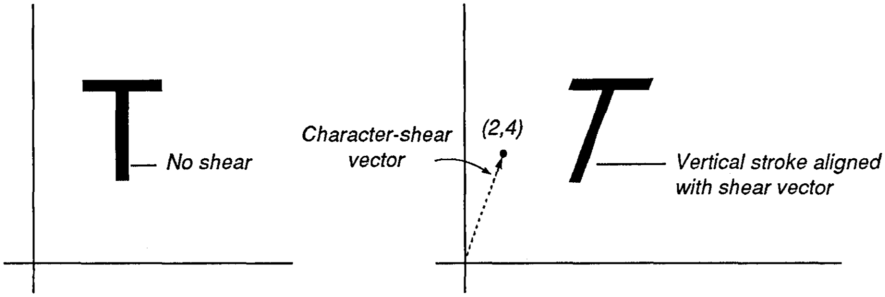
There are two character mix modes that affect how MS OS/2 combines the character colors with the existing color(s) on your application's drawing surface. The foreground character mix mode is overpaint; the background character mix mode is leave-alone. If the default colors and mix modes do not suit your application, you can change them by calling the **GpiSetAttrs** function. You can determine the current color and mix-mode settings by calling the **GpiQueryAttrs** function. For more information about color and mix modes, see Chapter 34, "Color and Mix Modes."

A special vector, drawn from the origin of an imaginary Cartesian coordinate system through a specified point, defines the character-angle attribute. MS OS/2 aligns the baseline with this vector. You can retrieve the point that defines the character-angle vector by calling the **GpiQueryCharAngle** function, or you can set the character angle by calling the **GpiSetCharAngle** function and passing it the coordinates of a point that defines the new vector.

A character box is an imaginary rectangle that applications use to scale characters in a font. You can determine the current character-box dimensions by calling the **GpiQueryCharBox** function. You can set new character-box dimensions by calling the **GpiSetCharBox** function. The dimensions that you pass to **GpiSetCharBox** and the dimensions that **GpiQueryCharBox** returns are fixed values. A fixed value, which is a representation of a floating-point number, is a 32-bit value whose high-order 16-bits contain the integral part of the floating-point number and whose low-order 16-bits contain the fractional part. The fractional part is the numerator of a fraction whose denominator is fixed at 65,536. If, for example, one of the box dimensions were 7.635 world units, you could obtain the corresponding fixed value by multiplying 7.635 by 65,536 and storing the integer part of the result (500,367) in a fixed variable. The high-order 16-bits would contain 0x0007 and the low-order 16-bits would contain 0xA28F.

A special vector, drawn from the origin of an imaginary Cartesian coordinate system through a specified point, defines the character-shear attribute. MS OS/2 aligns the vertical lines of the character box and the vertical strokes in a character with this vector. Figure 33.27 shows the effect of shearing a character:

Figure 33.27
Character Shear



You can determine the current character-shear angle by calling the **GpiQueryCharShear** function. This function returns a point you can use to determine the shear vector. You can set the character-shear angle by calling the **GpiSetCharShear** function and passing it a point structure with the appropriate values.

33.2.5.2 Character Modes

There are three character modes that determine whether MS OS/2 draws text using the current character attributes. When character mode 1 is set and the current font is an image font, MS OS/2 ignores the current shear, angle, and box attributes. When character mode 2 is set and the current font is an image font, MS OS/2 uses the current shear, angle, and box character attributes when it draws image-font text. In the current release of MS OS/2 Presentation Manager, mode 3 is reserved for outline fonts; if an application attempts to draw text output in mode 3 using an image font, MS OS/2 issues an error.

Figure 33.28 shows a line drawn at the current character-shear angle and an image-font character drawn after an application set the character mode to 2:

Figure 33.28
Image Font and Character Shear

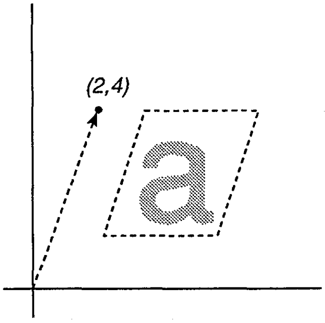


Figure 33.29 shows a line drawn at the current character angle and a string of image-font text drawn after an application set the character mode to 2:

Figure 33.29
Image Font and Character Angle

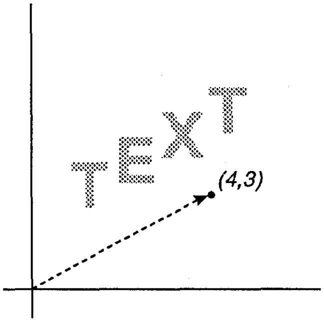
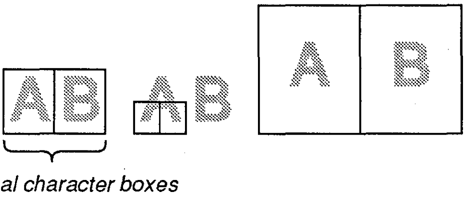


Figure 33.30 shows two letters of image-font text as they would appear in different sizes of character box after an application set the character mode to 2:

Figure 33.30
Image Font and Three Sizes of Character Box



You can determine which character mode is set by calling the **GpiQueryCharMode** function. You can set a new character mode by calling the **GpiSetCharMode** function.

If the current font is an outline font, MS OS/2 always draws text using the current character attributes—regardless of the current character mode.

Figure 33.31 shows a vector that corresponds to the current character-shear angle and a character of outline-font text:

Figure 33.31
Outline Font and Character Shear

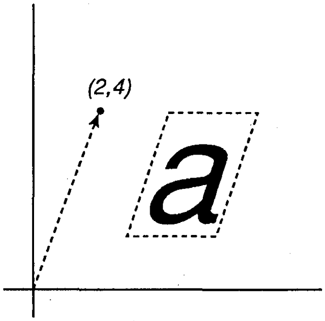


Figure 33.32 shows a vector that corresponds to the current character angle and a string of outline-font text:

Figure 33.32
Outline Font and Character Angle

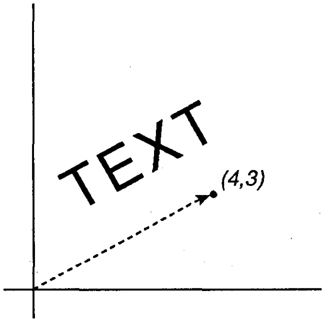
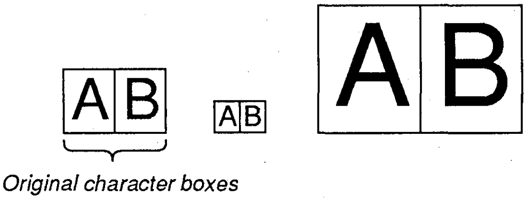


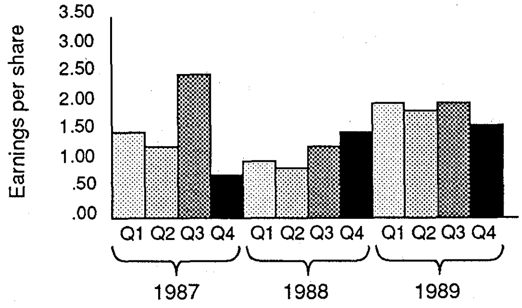
Figure 33.33 shows two letters of outline-font text as they would appear in different sizes of character box:

Figure 33.33
Outline Font and Three Sizes of Character Box



In addition to the character-attribute functions, you can use the Presentation Manager transformations to scale, rotate, translate, shear, and reflect text output drawn with an outline font. Figure 33.34 shows a bar graph labeled with outline text that has been rotated 90 degrees:

Figure 33.34
Graph with Outline Font



33.2.6 Font Files and Dynamic-Link Libraries

You can use Font Editor to alter and customize image-font files (files with a *.fnt* extension). After creating a custom font tailored to your application, you need to turn it into a dynamic-link library that your application can load. Once your application loads this library, it can use any font within it. A dynamic-link library is a collection of code and data segments that applications can access at run time. The code and data in a dynamic-link library is shareable—several applications can access it simultaneously. Dynamic-link libraries that contain fonts are unique; they contain data segments and empty (or useless) code segments. The MS OS/2 naming convention for a dynamic-link library requires that the library name end with a *.fon* extension.

The following text explains how you can create a dynamic-link library that contains your custom font. To create a dynamic-link library, you must use **masm**, the Microsoft Macro Assembler; **link**, the Microsoft Segmented-Executable linker; and **rc**, the MS OS/2 Resource Compiler. For this example, assume that your custom font file is named *newfont.fnt*.

After creating your custom font file, you need to create a special assembler file with your editor. You can use the following code fragment as the source code for this file:

```
code segment word    ;makes dummy code segment aligned on word boundary
db "empty_segment"   ;initializes a string in dummy segment
code ends            ;dummy segment ends here
end                  ;terminates source file
```

Call this file *generic.asm*. Once you have created *generic.asm*, assemble it with the following command:

```
masm generic
```

After assembling the file, you create a module-definition file. Call this file *generic.def*. It should contain the following statements:

```
LIBRARY generic
SEGMENTS  CODE MOVEABLE
```

The first statement tells the linker that you are creating a library with the module name *generic*. The second statement tells the linker that the segments in this library are movable code segments.

Upon creating *generic.def*, you start the linker with the following command:

```
link generic,,,,generic.def
```

This command creates an empty executable file called *generic.exe*.

After creating the empty executable file (which is the template for a dynamic-link library), you should create a resource file and call it *generic.rc*. For example, if your font file is called *newfont.fnt*, you would place the following statement in *generic.rc*:

```
FONT    200     newfont.fnt
```

This statement assigns an identifier, 200, to the font resource *newfont.fnt*.

Finally, you should use Resource Compiler to add the font file (*newfont.fnt*) to the empty dynamic-link library:

```
rc  generic.rc
```

The executable file, *generic.exe*, now contains your custom fonts. You still must rename this file *generic.fon* and copy it to a directory pointed to by the **libpath** command in your *config.sys* file. After you have done so, you can load the fonts into your application by calling the **GpiLoadFonts** function and passing it a pointer to the library name, generic, as the second argument.

33.2.7 Selecting New Fonts

If you need to use a font other than the default system font in your application, you can select a new one by calling the **GpiCreateLogFont** function. There are two kinds of fonts that you can select: public fonts and private fonts.

33.2.7.1 Public Fonts

Public fonts are those that a user loads by using the MS OS/2 Presentation Manager Control Panel. There are three dynamic-link libraries that contain Courier, Helvetica, and Times Roman fonts. If you load each library by using Control Panel, a total of 76 public fonts are available to any application that you run. These fonts are available in both outline and image formats in point sizes ranging from 8 to 24 points.

You can determine how many public fonts are currently loaded by calling the **GpiQueryFonts** function and passing it the QF_PUBLIC flag as the second argument, a NULL pointer as the third argument, and a count of 0 as the fourth argument, as shown in the following code fragment:

```
FONTMETRICS fm, afm[80];
LONG lCount, lFontCount;
HPS hps;
```

```
lFontCount = GpiQueryFonts(hps,
    QF_PUBLIC,
    NULL,   /* queries all public fonts */
    &lCount,
    (LONG) (sizeof(fm)),
    (PFONTMETRICS) afm);
```

You can determine the characteristics of the loaded public fonts by calling **GpiQueryFonts** and passing it the QF_PUBLIC flag, the count of available fonts returned by the first call, and the address of an array of **FONTMETRICS** structures, as shown in the following code fragment. MS OS/2 copies the attributes of the fonts into the array of **FONTMETRICS** structures, which you can then examine in order to select a font.

```
lFontCount = GpiQueryFonts(hps,
    QF_PUBLIC,
    NULL,
    &lCount,
    (LONG) (sizeof(fm)),
    (PFONTMETRICS) afm);
```

Once you determine which font you need, call the **GpiCreateLogFont** function, which copies various fields from the **FONTMETRICS** structure of the desired font into their corresponding fields in a **FATTRS** structure. The fields in the **FATTRS** structure describe the face name, code page, maximum baseline extent, and average character width of the font you would like to use. A special field, **lMatch**, contains a unique identifier that MS OS/2 uses to match your request to a font. Another field in the **FATTRS** structure specifies whether the font should be an image font or an outline font. You can use other fields to request that MS OS/2 synthesize an italic, underscored, strikeout, or bold font.

If MS OS/2 returns the value 2 after the call to **GpiCreateLogFont**, the function was successful. You can begin using the font *after* you assign it to the application's presentation space by passing the local identifier (*lcid*) from **GpiCreateLogFont** to the **GpiSetCharSet** function.

33.2.7.2 Private Fonts

Private fonts are fonts that an application loads exclusively for its own use. An application loads a private font when it calls the **GpiLoadFonts** function and passes it the name of the dynamic-link library that contains the fonts. (Note that in order to load a dynamic-link library of fonts, the library must be in one of the directories pointed to by the **libpath** command in the *config.sys* file.)

After the application loads the dynamic-link library of fonts, it can determine the characteristics of the fonts in that library by calling the **GpiQueryFonts** function.

You can determine how many private fonts are currently loaded by calling **GpiQueryFonts** passing it the QF_PRIVATE flag as the second argument, a NULL pointer as the third argument, and a count of 0 as the fourth argument, as shown in the following code fragment.

```
FONTMETRICS fm, afm[80];
LONG lCount;

lFontCount = GpiQueryFonts(hps,
    QF_PRIVATE,
    NULL,   /* queries all private fonts */
    &lCount,
    (LONG) (sizeof(fm)),
    (PFONTMETRICS) afm);
```

You can determine the characteristics of the loaded private fonts by calling **GpiQueryFonts** and passing it the QF_PRIVATE flag, the count of available fonts returned by the first call, and the address of an array of **FONTMETRICS** structures, as shown in the following code fragment. MS OS/2 copies the font attributes into the array of **FONTMETRICS** structures, which you can then examine in order to select a font.

```
lFontCount = GpiQueryFonts(hps,
    QF_PRIVATE,
    NULL,
    &lCount,
    (LONG) (sizeof(fm)),
    (PFONTMETRICS) afm);
```

Once you determine which font you need, call the **GpiCreateLogFont** function, which copies various fields from the **FONTMETRICS** structure into their corresponding fields in a **FATTRS** structure. The fields in the **FATTRS** structure describe the face name, code page, maximum baseline extent, and average character width of the font you would like to use. A special field, **lMatch**, contains a unique identifier that MS OS/2 uses to match your request to a font. Other fields in the **FATTRS** structure specify whether the font is an outline font or an image font and whether it is proportional or fixed. You can use another of the fields to request that MS OS/2 synthesize an italic, underscored, strikeout, or bold font. The **FATTRS** structure has the following form:

```
typedef struct _FATTRS {          /* fat */
    USHORT  usRecordLength;
    USHORT  fsSelection;
    LONG    lMatch;
    CHAR    szFacename[FACESIZE];
    USHORT  idRegistry;
    USHORT  usCodePage;
    LONG    lMaxBaselineExt;
    LONG    lAveCharWidth;
    USHORT  fsType;
    USHORT  fsFontUse;
} FATTRS;
```

If MS OS/2 returns the value 2 after the call to **GpiCreateLogFont**, the function was successful. You can begin using the font *after* you assign it to the application's presentation space by passing the local identifier (*lcid*) from **GpiCreateLogFont** to the **GpiSetCharSet** function. If MS OS/2 returns 1 after you call **GpiCreateLogFont**, the function was not successful and the new font is the default system font.

33.3 Using Fonts and Character Primitives

You can use the font and character functions to perform the following tasks:

- Select a font from the public fonts loaded with Control Panel.
- Select a font from a library of private fonts loaded by the application.
- Draw a string of text using the selected font.
- Scale, translate, and rotate a string of text from an outline font.
- Scale, shear, and alter the direction of a string of text from image and outline fonts.

33.3.1 Selecting a Public Font

To select a public font, you must perform the following tasks:

- Call the **GpiQueryFonts** function, passing the QF_PUBLIC flag, a NULL pointer to the font's face name, and a count of 0 to determine the number of available public fonts.
- Copy this number (the **GpiQueryFonts** return value) into an integer variable.
- Call **GpiQueryFonts** again, passing the QF_PUBLIC flag, a NULL pointer to the font's face name, and the count returned by the previous call.
- Examine the metrics, looking for the face name and attributes of the font that your application needs.
- Copy the appropriate metrics from the font that suits your application into a **FATTRS** structure.
- Initialize a local identifier (*lcid*) for the new font.
- Call the **GpiCreateLogFont** function, passing a local identifier for the font, the address of an empty array of eight characters, and the address of the **FATTRS** structure.
- Examine the return value from **GpiCreateLogFont**. If the function was successful, it should be 2.
- Pass the local identifier to the **GpiSetCharSet** function, assigning the font to your application's presentation space.

The following code fragment shows how to select a Helvetica public font:

```
i = 0;
while (!strcomp(afm[i++].szFacename, "Helv"));
lFontCount = GpiQueryFonts(hps,
    QF_PUBLIC,
    NULL,
    &lCount,
    (LONG) (sizeof(fm)),
    (PFONTMETRICS) afm);
lCount = lFontCount;
lFontCount = GpiQueryFonts(hps,
    QF_PUBLIC,
    NULL,
    &lCount,
    (LONG) (sizeof(fm)),
    (PFONTMETRICS) afm);
fat.usRecordLength = sizeof(fat);
fat.fsSelection = afm[i].fsSelection;
fat.lMatch = afm[i].lMatch;
strcpy(fat.szFacename, afm[i].szFacename);
fat.idRegistry = afm[i].idRegistry;
fat.usCodePage = afm[i].usCodePage;
fat.lMaxBaselineExt = afm[i].lMaxBaselineExt;
fat.lAveCharWidth = afm[i].lAveCharWidth;
fat.fsType = afm[i].fsType;
fat.fsFontUse = 0;
GpiCreateLogFont(hps,
    (PSTR8) chName,
    lcid,
    (PFATTRS) &fat);
GpiSetCharSet(hps, lcid);
```

33.3.2 Drawing Text

Before drawing text, you must determine which of the four text-output functions you should use. The following list describes the specific purpose of each text-output function:

Function name	Purpose
GpiCharString	This function draws a string of text starting at the current position.
GpiCharStringAt	This function draws a string of text starting at a point that you pass as a function argument. It is identical to a function sequence of **GpiMove, GpiCharString**.
GpiCharStringPos	This function alters the intercharacter spacing in a string of text in order to shade a special rectangle that surrounds a string of text, to draw a string of text in halftones, or to clip a string of text to a special rectangle. This function starts drawing the text at the current position.
GpiCharStringPosAt	This function alters the intercharacter spacing in a string of text in order to shade a special rectangle that surrounds a string of text, to draw a string of text in halftones, or to clip a string of text to a special rectangle. This function starts drawing the text at a point that you pass as one of the function arguments.

The following code fragment shows how to load an array of characters with the string "MS OS/2 Presentation Manager", set the current position to the point (100,100) by calling the **GpiMove** function, and then draw the string by calling the **GpiCharString** function:

```
ptl.x = 100; ptl.y = 100;
GpiMove (hps, &ptl);
GpiCharString(hps, 28L, "MS OS/2 Presentation Manager");
```

33.3.3 Transforming Text from an Outline Font

The following code fragment shows how to rotate a string of text 90 degrees counterclockwise by using the model transformation:

```
matlfTransform.fxM11 = MAKEFIXED(1, 0);       /* translates text */
matlfTransform.fxM12 = MAKEFIXED(0, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 0);
matlfTransform.fxM22 = MAKEFIXED(1, 0);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = -300L;
matlfTransform.lM32 = -100L;
matlfTransform.lM33 = 1L;
GpiSetModelTransformMatrix(hps, 9L, &matlfTransform,
    TRANSFORM_REPLACE);
```

```
matlfTransform.fxM11 = MAKEFIXED(0, 0);        /* rotates text */
matlfTransform.fxM12 = MAKEFIXED(1, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = -MAKEFIXED(1, 0);
matlfTransform.fxM22 = MAKEFIXED(0, 0);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = 0L;
matlfTransform.lM32 = 0L;
matlfTransform.lM33 = 1L;
GpiSetModelTransformMatrix(hps, 9L, &matlfTransform, TRANSFORM_ADD);

matlfTransform.fxM11 = MAKEFIXED(1, 0);        /* translates again */
matlfTransform.fxM12 = MAKEFIXED(0, 0);
matlfTransform.lM13 = 0L;
matlfTransform.fxM21 = MAKEFIXED(0, 0);
matlfTransform.fxM22 = MAKEFIXED(1, 0);
matlfTransform.lM23 = 0L;
matlfTransform.lM31 = 300L;
matlfTransform.lM32 = 100L;
matlfTransform.lM33 = 1L;
GpiSetModelTransformMatrix(hps, 9L, &matlfTransform, TRANSFORM_ADD);

ptl.x = 100;
ptl.y = 100;
GpiMove(hps, &ptl);
GpiCharString(hps, 28L, "MS OS/2 Presentation Manager");
```

33.3.4 Transforming Text from an Image Font

The following code fragment shows how to double the size of the character box, set the character mode to 2, and then print a string of text:

```
GpiQueryCharBox(hps, &sizfxBox);
sizfxBox.cx = 2 * sizfxBox.cx;
sizfxBox.cy = 2 * sizfxBox.cy;
GpiSetCharBox(hps, &sizfxBox);
ptl.x = 100;
ptl.y = 100;
GpiMove(hps, &ptl);
GpiCharString(hps, 28L, "MS OS/2 Presentation Manager");
```

33.4 Summary

The following list summarizes the MS OS/2 font and character-primitive functions:

GpiCharString Draws a string of text starting at the current position. It sets the current position to the end of the string at the point where the next character would be drawn. All text is drawn using the font assigned to the application's presentation space.

GpiCharStringAt Draws a string of text starting at a point that you specify as a function argument. It updates the current position after drawing each character in the string and, upon completion, sets the current position to the point where the next character would be drawn. All text is drawn using the font assigned to the application's presentation space.

GpiCharStringPos Draws a string of text starting at the current position. It sets the current position to the end of the string at the point where the next character would be drawn. All text is drawn using the font assigned to the application's presentation space. In addition to drawing a string of text, you can use this function to do the following:

- Adjust spacing between characters in the string.
- Clip the text to a small rectangle.

GpiCharStringPosAt Draws a string of text starting at a point that you specify as one of the function arguments. It sets the current position to the end of the string at the point where the next character would be drawn. All text is drawn using the font assigned to the application's presentation space. In addition to drawing a string of text, you can use this function to do the following:

- Adjust spacing between characters in the string.
- Shade the text by drawing it in halftones.
- Clip the text to a small rectangle.

GpiCreateLogFont Passes a description of a font to MS OS/2, which attempts to select the closest possible match from available fonts in MS OS/2 or on the device associated with the current presentation space. If the font you request is not available, MS OS/2 selects the default system font. A **FATTRS** structure contains a description of the requested font. Typically, applications call the **GpiQueryFonts** function and examine the metrics for a number of fonts in the system and then load the **FATTRS** structure with values from the metrics of the desired font. If MS OS/2 is successful and finds a font that matches the values in **FATTRS**, the function returns a value of 2; if MS OS/2 does not find a matching font and substitutes the default system font instead, the function returns a value of 1; if the function fails completely, it returns 0.

GpiDeleteSetId Frees a local identifier (*lcid*). A local identifier is a long integer value that identifies a particular font that you are using in your application. Once you free a local identifier from a particular font, that font is no longer available for use.

GpiLoadFonts Loads private fonts from a dynamic-link library of fonts so that an application can access them by calling the **GpiQueryFonts** and **GpiCreateLogFont** functions. You identify a specific dynamic-link library by passing it the appropriate path and filename as arguments.

GpiQueryCharAngle Retrieves the current value of the character-baseline angle. If the current font is an outline font, the baseline of each character's cell is drawn parallel to this angle. If the current font is an image font, the character-baseline angle uniquely affects the image-font output. If the current font is an image font and the character mode is 1, MS OS/2 ignores the baseline angle. If the current font is an image font, the character mode is 2, and the baseline angle is nonzero, MS OS/2 aligns text with the baseline angle. However, the baseline of each character remains parallel to the *x*-axis.

GpiQueryCharBox Retrieves the dimensions (in world coordinates) of the character box. If the current font is an outline font, the character-cell dimensions are reduced or expanded to match the character-box dimensions. If the current font is an image font, the character-box dimensions uniquely affect the image-font output. If the current font is an image font and the character mode is 1, MS OS/2 ignores the character-box dimensions when it draws text. If the current font is an image font and the character mode is 2, MS OS/2 increases or reduces the spacing between characters (based on the box dimensions) when it draws text.

GpiQueryCharMode Retrieves the current character-mode attribute. There are three character modes; each affects text output with an image font. In mode 1, MS OS/2 ignores the character angle, box, and shear attributes when drawing text with an image font. In mode 2, MS OS/2 uses each of the attributes when drawing text with an image font. In mode 3, MS OS/2 issues an error if the application attempts to draw text using an image font. An application can use outline fonts for text output in character mode 3 only.

GpiQueryCharSet Retrieves the local identifier assigned to the current font.

GpiQueryCharShear Retrieves the current character-shear attribute. The character-shear attribute is a point that defines the angle between the *x*-axis of a coordinate system and a line drawn from a system's origin through the point. If the current font is an outline font, the sides of the character box are drawn parallel to this angle and the character within the box is slanted accordingly. If the current font is an image font and the character mode is 1, MS OS/2 ignores this attribute when drawing text. If the current font is an image font and the character mode is 2, MS OS/2 draws the sides of the character box parallel to the angle; the character itself remains upright.

GpiQueryCharStringPos Retrieves an array of points that specify the position (in world coordinates) that MS OS/2 would assign to each character in a string of text if the string were drawn by the **GpiCharStringPos** function.

GpiQueryCharStringPosAt Retrieves an array of points that specify the position (in world coordinates) that MS OS/2 would assign to each character in a string of text if the string were drawn by the **GpiCharStringPosAt** function.

GpiQueryCp Retrieves the current code-page identifier. A code page is a character set that contains 256 characters. Each character in the code page is assigned a value (called a code point) in the range 0 through 255. A code-page identifier is a three-digit value in the range 0 through 999.

GpiQueryDefCharBox Retrieves the default character-box dimensions (in world coordinates). The default dimensions are assigned by the font's designer.

GpiQueryFontFileDescriptions Retrieves the types and face names of fonts in a file, if that file is a font file.

GpiQueryFontMetrics Retrieves the font metrics for the current logical font.

GpiQueryFonts Retrieves metrics for all of the public or private fonts if the third argument is a NULL pointer. Otherwise, it returns the metrics for fonts with a particular face name. The function returns the metrics in an array of **FONTMETRICS** structures. A public font is one that MS OS/2 loads when a user turns the computer on. A private font is one that an application loads.

GpiQueryKerningPairs Retrieves an array of **KERNINGPAIRS** structures. Kerning is space that appears between a pair of characters when MS OS/2 draws them. By adjusting the kerning between certain characters, you can make text more readable. The **KERNINGPAIRS** structure contains three fields: The first two identify the characters, and the third specifies the amount of kerning (measured in world coordinates). If this last field contains a negative number, the kerning is reduced by that amount; if the last field contains a positive number, the kerning is increased by that amount.

GpiQueryNumberSetIds Retrieves the number of current local identifiers that an application is using. You can use this value when you call the **GpiQuerySetIds** function.

GpiQueryTextBox Retrieves the dimensions of a rectangle surrounding a string of text that an application draws using the **GpiCharStringPos** or **GpiCharStringPosAt** function. You can use these dimensions to draw a shaded rectangle behind the text.

GpiQueryWidthTable Retrieves width values you can use to determine the average character width of characters in the current logical font. To determine this average character width, you multiply the width values by the factors given in the description of the **FONTMETRICS** structure.

GpiSetCharAngle Sets the current value of the character-baseline angle. If the current font is an outline font, the baseline of each character's cell is drawn parallel to this angle. If the current font is an image font, the character-baseline angle uniquely affects the image-font output. If the current font is an image font and the character mode is 1, MS OS/2 ignores the baseline angle. If the current font is an image font, the character mode is 2, and the baseline angle is nonzero, MS OS/2 aligns text with the baseline angle. However, the baseline of each character remains parallel to the *x*-axis.

GpiSetCharBox Sets the dimensions (in world coordinates) of the character box. If the current font is an outline font, the character-cell dimensions are reduced or expanded to match the dimensions of the character box. If the current font is an image font, the dimensions of the character box uniquely affect the image-font output. If the current font is an image font and the character mode is 1, MS OS/2 ignores the character-box dimensions when drawing text. If the current font is an image font and the character mode is 2, MS OS/2 increases or reduces the spacing between characters (based on the box dimensions) when drawing text strings.

GpiSetCharMode Sets the current character-mode attribute. There are three character modes; each affects text output with an image font. In mode 1, MS OS/2 ignores the angle, box, and shear attributes when drawing text with an image font. In mode 2, MS OS/2 uses each of the attributes when drawing text with an image font. In mode 3, MS OS/2 issues an error if the application attempts to draw text using an image font. An application can use outline fonts for text output in character mode 3 only.

GpiSetCharSet Assigns a font to a presentation space. The font is identified by a local identifier (*lcid*).

GpiSetCharShear Sets the current character-shear attribute. The character-shear attribute is a point that defines an angle between the *x*-axis of a coordinate system and a line drawn from a system's origin through the point. If the current font is an outline font, the sides of the character box are drawn parallel to this angle and the character within the box is slanted accordingly. If the current font is an image font and the character mode is 1, MS OS/2 ignores this attribute when drawing text. If the current font is an image font and the character mode is 2, MS OS/2 draws the sides of the character box parallel to the angle; the character itself remains upright.

GpiSetCp Sets the current code-page identifier. A code page is a character set that contains 256 characters. Each character in the code page is assigned a value (called a code point) in the range 0 through 255. A code-page identifier is a three-digit value in the range 0 through 999.

GpiUnloadFonts Unloads font definitions that were loaded previously from a resource file.

Chapter

34

Color and Mix Modes

34.1 Introduction 469

34.2 About Color and Mix Modes 469

34.2.1 Color and RGB Values 470

34.2.2 Color Tables 470

34.2.3 Color Output and Mix Modes 471

34.2.4 Dithering 474

34.3 Using Colors and Mix Modes 474

34.3.1 Creating a Logical Color Table 474

34.3.2 Determining the Color-Table Format and Index Values 475

34.3.3 Determining an Index Value for an RGB Value 475

34.3.4 Setting the Primitive Color Attributes 475

34.4 Summary 476

34.1 Introduction

This chapter describes color and mix modes and their use in MS OS/2 applications. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Line and arc primitives
- Area primitives
- Character primitives
- Marker primitives

34.2 About Color and Mix Modes

Color and mix modes are two primitive attributes. The color attribute describes a line, arc, character, marker, area, or image color before it is combined with the color on the drawing surface. The mix-mode attribute describes how the system combines a primitive's color with the color of the drawing surface. Some primitives have foreground and background color attributes. For instance, the character primitive has a foreground color attribute that specifies the color of the character and a background color attribute that specifies the color surrounding the character. The primitives that have foreground and background attributes also have foreground and background mix modes.

To understand color, it is important to understand several principles of color devices. Most color devices can generate three fundamental colors: red, green, and blue. On some devices, each of these three colors corresponds to a color plane—for example, a red pel is visible when a pel in the red plane is on and the corresponding pels in the green and blue planes are off. On other devices, there is only one color plane, and each pel contains a red, a green, and a blue section—for example, a red pel is visible when the red section of that pel is on and the green and blue sections are off. By combining the three fundamental colors, a device can obtain five additional colors, for a total of eight. These eight colors, and the combinations that produce them, are as follows:

Resulting color	Combined colors
Black	Red, green, and blue are off.
Red	Red is on; green and blue are off.
Green	Green is on; red and blue are off.
Blue	Blue is on; red and green are off.
Pink	Red and blue are on; green is off.
Cyan	Blue and green are on; red is off.
Yellow	Red and green are on; blue is off.
White	Red, green, and blue are on.

34.2.1 Color and RGB Values

In MS OS/2, the red, green, and blue components of a color are either stored in an RGB structure or stored as a long integer (32-bit) value. The RGB structure has the following format:

```
typedef struct _RGB {        /* rgb */
    BYTE bBlue;
    BYTE bGreen;
    BYTE bRed;
} RGB;
```

The RGB value has the following format:

```
0x00RRGGBB.
```

Each field in the RGB structure and each of the last three bytes in the RGB value specify a color intensity in the range 0 through 255, 0 being the lowest intensity, 255 the highest. If a field or byte contains 0, its corresponding color is not visible; if a field or byte contains 128, the color is pale; and if a field or byte contains 255, the color is as intense as the device allows. If all the fields or bytes are set to 0, the corresponding color is black. Similarly, if all the fields or bytes are set to 255, the corresponding color is white.

The following list shows the RGB value associated with each of the eight fundamental colors:

Color	Associated RGB value
White	0x00FFFFFF
Yellow	0x00FFFF00
Pink	0x00FF00FF
Cyan	0x0000FFFF
Blue	0x000000FF
Green	0x0000FF00
Red	0x00FF0000
Black	0x00000000

34.2.2 Color Tables

A color table is an array of RGB values. There are two kinds of color tables: physical (used by device drivers) and logical (used by applications). The physical color table contains RGB values representing the available colors on a device. The logical color table contains RGB values representing the colors that an application would prefer to use. You can determine which colors are in the physical color table by calling the **GpiQueryRealColors** function. You can determine which colors are in the current logical color table by calling the **GpiQueryLogColorTable** function.

The following list contains the index values and colors found in a logical color table:

Index	Color
0	Device's background color (white)
1	Blue
2	Red
3	Pink
4	Green
5	Cyan
6	Yellow
7	Neutral color (black)
8	Dark gray
9	Pale blue
10	Pale red
11	Pale pink
12	Dark green
13	Dark cyan
14	Brown
15	Pale gray

The value you place in the color field of a primitive-attribute structure (by using **GpiSetAttrs**) is an index into the logical color table. When MS OS/2 draws the primitive, it searches the physical color table for a color that is the closest approximation to this color index. MS OS/2 then uses this approximate color to draw with.

You can replace the default logical color table with a new color table by calling the **GpiCreateLogColorTable** function. You can also use this function to reset the logical color table to its original values. Either way, once you create a new logical color table (or reset it to its original values), it becomes part of the application's presentation space.

For a given device, you can determine the maximum size of the logical color table by using the **DevQueryCaps** function. This function also returns the maximum number of distinct colors available on a particular device.

34.2.3 Color Output and Mix Modes

When an application draws a line, arc, character, marker, area, or image, MS OS/2 uses a mix mode to determine the color that appears on a video display, printer, or plotter. A mix mode is a bitwise operation on the color indices in a device's physical color table. For example, suppose an application has set the **lColor** field in the **LINEBUNDLE** structure to CLR_BLUE and the **usMixMode** field in the same structure to FM_OR. The current drawing-surface color is

CLR_BACKGROUND. The CLR_BLUE entry corresponds to the color blue in both the presentation space's logical color table and the device's physical color table. The CLR_BACKGROUND entry corresponds to the color white in both the presentation space's logical color table and the device's physical color table. The index value for blue in the device's table is 0x0001, and the index value for white in the device's table is 0x0000. To determine the color of a line, MS OS/2 performs a bitwise OR operation on the two index values, as follows:

```
0x0001
0x0000
0x0001   (Result of bitwise OR)
```

In this case, the result is 0x0001, the index value for blue in the device's color table. This means that a blue line appears when the application calls the **GpiLine** or **GpiPolyLine** function.

Some of the primitive bundles contain foreground and background colors and mix modes. For instance, the **AREABUNDLE** structure contains a foreground color that corresponds to the foreground color of the area fill pattern. The **AREABUNDLE** structure also contains a background color that corresponds to the background color that appears behind the area fill pattern. **AREABUNDLE** also contains foreground and background mix modes that specify bitwise operations on index values in the device's physical color table.

There are 16 foreground mix modes. For each mix mode, the index value for the foreground and current drawing-surface colors (in the device's physical color table) are combined by using one of the bitwise operators. The different foreground mix modes are described in the following list:

Mix mode	Description
FM_AND	Final color's index value is determined by a bitwise AND operation on the foreground color's index value and the drawing surface's index value.
FM_INVERT	Final color's index value is always the inverse of the drawing surface's index value.
FM_LEAVEALONE	Final color's index value is that of the drawing-surface color.
FM_MASKSRCNOT	Final color's index value is determined by inverting the drawing surface's index value and performing a bitwise AND operation on this value and the foreground color's index value.
FM_MERGENOTSRC	Final color's index value is determined by performing a bitwise AND operation on the drawing surface's index value and the inverse of the foreground color's index value.
FM_MERGESRCNOT	Final color's index value is determined by performing a bitwise AND operation on the foreground color's index value and the inverse of the drawing-surface color's index value.

Mix mode	Description
FM_NOTCOPYSRC	Final color's index value is the inverse of the foreground color's index value.
FM_NOTMASKSRC	Final color's index value is the inverse of the FM_AND result.
FM_NOTMERGESRC	Final color's index value is always the inverse of the FM_OR result.
FM_NOTXORSRC	Final color's index value is always the inverse of the FM_XOR result.
FM_ONE	Final color's index value is always 1.
FM_OR	Final color's index value is determined by a bitwise OR operation on the foreground color's index value and the drawing surface's index value.
FM_OVERPAINT	Final color's index value is that of the foreground color.
FM_SUBTRACT	Final color's index value is determined by inverting the foreground color's index value and performing a bitwise AND operation on this value and the drawing surface's index value.
FM_XOR	Final color's index value is determined by a bitwise XOR operation on the foreground color's index value and the drawing surface's index value.
FM_ZERO	Final color's index value is always zero.

There are four background mix modes. For each mix mode, the index value for the background color and the current drawing-surface color (in the device's physical color table) are combined using one of the bitwise operators. The different background mix modes are described in the following list:

Mix mode	Description
BM_LEAVEALONE	Final color's index value is that of the drawing-surface color.
BM_OR	Final color's index value is determined by a bitwise OR operation on the background color's index value and the drawing surface's index value.
BM_OVERPAINT	Final color's index value is that of the background color.
BM_XOR	Final color's index value is determined by a bitwise XOR operation on the background color's index value and the drawing surface's index value.

34.2.4 Dithering

If you request a color that isn't available in the physical color table, MS OS/2 obtains the closest match by a process called dithering. For example, if the physical color table does not contain a certain shade of gray, MS OS/2 can create what appears to be a gray by mixing black pels and white pels. Similarly, if the physical color table does not contain a light green color but does contain a yellow and a green, MS OS/2 can create what appears to be light green by mixing yellow pels and green pels.

Dithering is especially important on monochrome devices. By combining various combinations of black pels with white pels, MS OS/2 can create numerous shades of gray.

34.3 Using Colors and Mix Modes

You can use the color and mix-mode functions to perform the following tasks:

- Create a logical color table.
- Determine the format and the starting and ending index values of the current logical color table.
- Determine the index value for an entry in the logical color table that is the closest match to an RGB value.
- Determine the RGB value associated with a particular entry in the logical color table.
- Determine and set the current foreground and background colors.
- Determine and set the current foreground and background mix modes.

34.3.1 Creating a Logical Color Table

To create a logical color table, you must perform the following tasks:

1. Create an array of RGB values that will replace the existing logical color table.
2. Call the **GpiCreateLogColorTable** function, using the LCOL_RESET and LCOLF_CONSECRGB flags.

The following code fragment demonstrates this process:

```
LONG alTable[] = {
    0xFFFFFF,  /* white */
    0xEDEDED,
    0xDBDBDB,
    0xC9C9C9,
    0xB7B7B7,
    0xA6A6A6,
    0x949494,
    0x828282,
    0x707070,
    0x5E5E5E,
    0x4D4D4D,
    0x3B3B3B,
    0x292929,
    0x171717,
    0x050505,
    0x000000 }; /* black */
```

```
GpiCreateLogColorTable(hps,
    LCOL_RESET,       /* begins with default         */
    LCOLF_CONSECRGB,  /* consecutive RGB values      */
    OL,               /* starting index in table     */
    16L,              /* number of elements in table */
    alTable);
```

34.3.2 Determining the Color-Table Format and Index Values

To determine the format and the starting and ending index values of the current logical color table, you can call the **GpiQueryColorData** function. The following code fragment calls **GpiQueryColorData** to determine whether the default logical color table is loaded and, if so, loads a new table:

```
LONG lClrData[3];
    .
    .
    .
GpiQueryColorData(hps, 3L, lClrData);
if (lClrData == LCOLF_DEFAULT)
    GpiCreateLogColorTable(hps,
        LCOL_RESET,       /* begins with default         */
        LCOLF_CONSECRGB,  /* consecutive RGB values      */
        OL,               /* starting index in table     */
        16L,              /* number of elements in table */
        alTable);
```

34.3.3 Determining an Index Value for an RGB Value

To determine the logical-color-table index value associated with an RGB value, you can call the **GpiQueryColorIndex** function. The following code fragment shows how to determine which index value matches the RGB value for pink (0x00FF00FF) and then use that index entry to set the foreground color to pink for each of the primitive attributes:

```
LONG lIndex = 255;     /* logical-color-table index */

lIndex = GpiQueryColorIndex(hps, LCOLOPT_REALIZED, 0x00FF00FF);
if ((lIndex >= 0) && (lIndex <= 15))     /* checks for valid index */
    GpiSetColor(hps, lIndex);
```

34.3.4 Setting the Primitive Color Attributes

To set the color attributes for a single primitive, you can call the **GpiSetAttrs** function, or you can set the color attributes for all of the primitives in a presentation space by calling the **GpiSetColor** and **GpiSetBackColor** functions.

The following code fragment shows how to use **GpiSetAttrs** to set the line-primitive color attribute to dark gray:

```
LINEBUNDLE lbnd; /* line-primitive attribute bundle */

lbnd.lColor = CLR_DARKGRAY;
GpiSetAttrs(hps, PRIM_LINE, LBB_COLOR, OL, &lbnd);
```

The next code fragment shows how to use **GpiSetColor** to set the foreground color attribute for all of the primitives to dark gray:

```
GpiSetColor(hps, CLR_DARKGRAY);
```

34.4 Summary

The following list summarizes the MS OS/2 color and mix-mode functions:

GpiCreateLogColorTable Creates a table of colors for an application. After you call this function, MS OS/2 creates the closest match to the ideal colors in your logical color table by mapping colors from the device's physical color table and then dithering them.

GpiQueryBackColor Retrieves the current background color. If the background colors were set by the **GpiSetBackColor** function, the color returned by **GpiQueryBackColor** is the background color for line, character, marker, and bitmap operations. If the background colors were set by the **GpiSetAttrs** function, the color returned by **GpiQueryBackColor** is the background color for character output only.

GpiQueryBackMix Retrieves the current background mix mode. If the background mix mode was set by the **GpiSetBackMix** function, the mix mode returned by **GpiQueryBackMix** is the background mix mode for line, character, marker, and bitmap operations. If the background mix mode was set by the **GpiSetAttrs** function, the mix mode returned by **GpiQueryBackMix** is the background mix mode for character output only.

GpiQueryColor Retrieves the current foreground color. If the foreground color was set by the **GpiSetColor** function, the color returned by **GpiQueryColor** is the foreground color for line, character, marker, and bitmap operations. If the foreground color was set by the **GpiSetAttrs** function, the color returned by **GpiQueryColor** is the foreground color for character output only.

GpiQueryColorData Retrieves information about the current logical color table. This information includes the format and the starting and ending index values of the current logical color table.

GpiQueryColorIndex Retrieves the index value for the logical-color-table entry that is the closest possible match to a specified RGB value.

GpiQueryLogColorTable Retrieves information about the current logical color table.

GpiQueryMix Retrieves the value of the current foreground-color mix-mode attribute.

GpiQueryNearestColor Retrieves the RGB value for an available color on the current output device that is the closest possible color match to a specified RGB value.

GpiQueryRGBColor Retrieves the RGB value associated with a logical-color-table index.

GpiQueryRealColors Retrieves the RGB values for actual colors in the device's color table.

GpiRealizeColorTable Not supported in the current release of Presentation Manager for MS OS/2.

GpiUnrealizeColorTable Not supported in the current release of Presentation Manager for MS OS/2.

GpiSetBackColor Sets the background color for line, character, marker, and bitmap operations.

GpiSetBackMix Sets the background mix mode for line, character, marker, and bitmap operations.

GpiSetColor Sets the foreground color for line, character, marker, and bitmap operations.

GpiSetMix Sets the foreground mix mode for line, character, marker, and bitmap operations.

Chapter
35

Paths

35.1 Introduction 481

35.2 About Paths 481
35.2.1 Geometric Lines 483
35.2.2 Polygons and Other Shapes 484
35.2.3 Fill Modes 485
35.2.3.1 Alternate Mode 485
35.2.3.2 Winding Mode 486
35.2.4 Clip Paths 487
35.2.5 Path Attributes 487

35.3 Using Paths 490
35.3.1 Drawing a Geometric Line 490
35.3.2 Drawing a Filled Polygon 491
35.3.3 Creating a Clip Path 491

35.4 Summary 492

35.1 Introduction

This chapter describes a graphics object called a path. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Line and arc primitives
- Color and mix modes
- Area primitives
- Clipping

35.2 About Paths

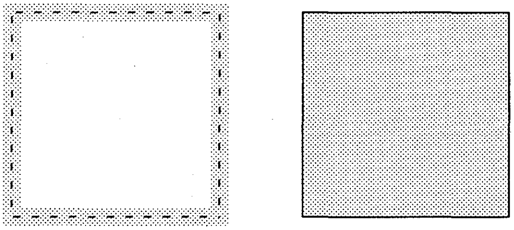
A path is a figure that is filled or outlined. You can use paths to draw geometric (wide) lines, create nonrectangular clipping regions, and outline and/or fill irregular shapes and polygons.

Paths provide the only means of drawing lines that are wider than one pel (pixel); the only means of clipping to circular, elliptical, or other nonrectangular regions; and the only means of generating filled or outlined irregular shapes and polygons (including triangles, trapezoids, rhomboids, and other nonrectangular regions). To generate filled or outlined rectangles, you can use the **GpiBox** function, and to generate filled or outlined circles and ellipses, you can use the **GpiFullArc** function.

A path is similar to another graphics object called an area, but there are fundamental differences between the two objects. The interior of an area is always filled, while the interior of a path can be empty. Even when the interior of a path is empty, the borders are visible. Paths with empty interiors are called stroked paths because their visible borders are drawn, or "stroked." Another difference is in the use of paths and areas. Applications perform clipping operations by using paths but not by using areas. For more information about areas, see Chapter 36, "Area Primitives."

There are two operations that an application can perform on a path: stroke and fill. You use a stroked path to draw geometric lines and to outline polygons. You use a filled path to fill the interior of polygons. Figure 35.1 shows a rectangle that is drawn first as a stroked path and then as a filled path:

Figure 35.1
Stroked and Filled Rectangles



When MS OS/2 strokes a path, it generates a geometric line along the original line that defined the path. The end result is like the stroke of a brush. When MS OS/2 fills a path, it fills the region surrounded by the original line that defined the path. If this original line did not close the region, MS OS/2 automatically closes the region and fills the path.

The functions that generate a path are always enclosed in a path bracket. The **GpiBeginPath** function defines the beginning of a path bracket, and the **GpiEndPath** function defines the end of a path bracket. The functions you can use within a path bracket are as follows:

- **GpiBeginElement**
- **GpiBox**
- **GpiCallSegmentMatrix**
- **GpiCharString**
- **GpiCharStringAt**
- **GpiCharStringPos**
- **GpiCharStringPosAt**
- **GpiCloseFigure**
- **GpiComment**
- **GpiCreateLogFont**
- **GpiDeleteElement** (retain mode only)
- **GpiDeleteElementRange** (retain mode only)
- **GpiDeleteElementsBetweenLabels** (retain mode only)
- **GpiDeleteSetId**
- **GpiElement**
- **GpiEndElement**
- **GpiEndPath**
- **GpiFullArc**
- **GpiGetData**
- **GpiLabel**
- **GpiLine**
- **GpiMarker**
- **GpiMove**
- **GpiOffsetElementPointer**
- **GpiPartialArc**
- **GpiPointArc**
- **GpiPolyFillet**
- **GpiPolyFilletSharp**
- **GpiPolyLine**

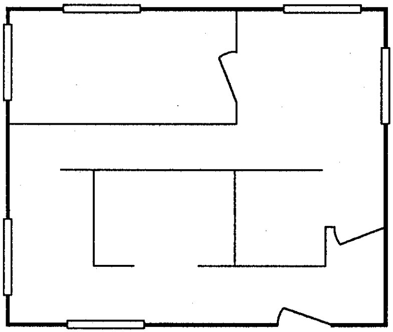
- **GpiPolyMarker**
- **GpiPolySpline**
- **GpiPop**
- **GpiPutData**
- **GpiQueryArcParams** (not valid in retain mode)
- **GpiQueryAttrMode**
- **GpiQueryCurrentPosition** (not valid in retain mode)
- **GpiSetArcParams**
- **GpiSetAttrMode**
- **GpiSetAttrs**
- **GpiSetCharAngle**
- **GpiSetCharBox**
- **GpiSetCharDirection**
- **GpiSetCharMode**
- **GpiSetCharSet**
- **GpiSetCharShear**
- **GpiSetColor**
- **GpiSetCp**
- **GpiSetCurrentPosition**
- **GpiSetEditMode**
- **GpiSetElementPointer** (not valid in retain mode)
- **GpiSetElementPointerAtLabel** (not valid in retain mode)
- **GpiSetLineEnd**
- **GpiSetLineJoin**
- **GpiSetLineType**
- **GpiSetLineWidth**
- **GpiSetMarker**
- **GpiSetMarkerBox**
- **GpiSetMarkerSet**
- **GpiSetMix**
- **GpiSetModelTransformMatrix**

35.2.1 Geometric Lines

A geometric line is a stroked path that your application can scale by using one of the scaling transformations. Geometric lines can be wider than one pel. When you draw geometric lines, you use the same functions you would use to draw normal lines: **GpiMove**, **GpiLine**, **GpiPolyLine**, **GpiPartialArc**, and so on. But you must enclose these functions in a path bracket.

In the floor plan shown in Figure 35.2, the outer walls were drawn using geometric lines. All of the other objects were drawn using normal lines.

Figure 35.2
Geometric Lines and Normal Lines



After constructing a path bracket for a geometric line, you must specify a line-end style, a line-join style, and a line width. To specify the line-end style, you call the **GpiSetLineEnd** function; to specify the line-join style, you call the **GpiSetLineJoin** function; and to specify the line width, you call the **GpiSetLineWidthGeom** function. Once you have completed these steps, you can draw the line by calling the **GpiStrokePath** function.

35.2.2 Polygons and Other Shapes

As noted earlier, the **GpiFullArc** function can generate filled or outlined circles and ellipses, and the **GpiBox** function can generate filled or outlined rectangles. But to generate filled or outlined regions that are not circular, elliptical, or rectangular, you should use the path functions.

Figure 35.3 shows outlines of a trapezoid, a rhomboid, and a rhombus, each constructed by calling the **GpiBeginPath**, **GpiPolyLine**, **GpiEndPath**, and **GpiStrokePath** functions:

Figure 35.3
Stroked Paths

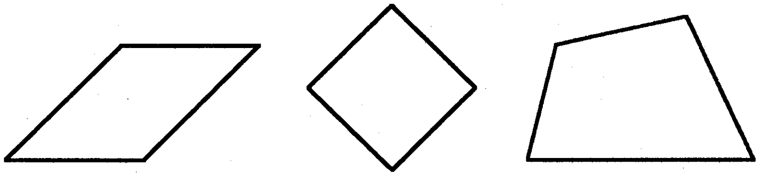
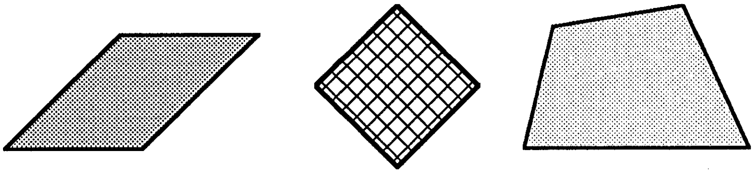


Figure 35.4 shows the same quadrilaterals, only this time they were filled by calling the **GpiFillPath** function:

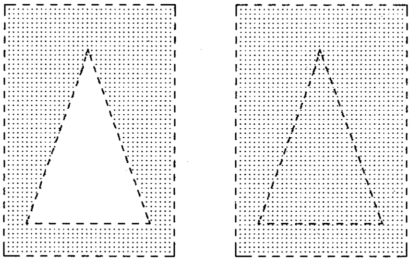
Figure 35.4
Filled Paths



35.2.3 Fill Modes

To fill a path, MS OS/2 uses one of two methods, called fill modes. You can select the mode—either alternate or winding—when you call **GpiFillPath**. If the path consists of multiple intersecting regions, the mode will affect the final appearance of the path. Figure 35.5 shows two identical paths that were filled using the two modes. Each path consists of a triangle drawn within a rectangle. The path on the left was filled using the alternate mode, the path on the right was filled using the winding mode.

Figure 35.5
Fill Modes



35.2.3.1 Alternate Mode

When an application specifies the alternate mode, MS OS/2 performs a test on pels inside the path's boundary lines. This test involves the following steps:

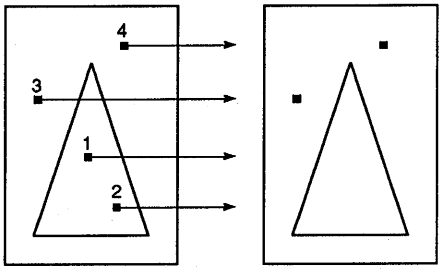
1 Select a pel within the path's boundary lines.

2 Draw an imaginary ray, in the *positive x*-direction, from that pel towards infinity.

3 Highlight the pel if the ray intersects the boundary lines of the path an *odd* number of times.

Figure 35.6 shows how MS OS/2 would perform this test on the path shown in Figure 35.5:

Figure 35.6
Alternate-Mode Test



35.2.3.2 Winding Mode

When an application specifies winding mode, MS OS/2 again performs a test on pels inside the path's boundary lines. The winding-mode test, which differs from the alternate-mode test, involves the following steps:

1 Determine the direction in which each boundary line was drawn.

2 Select a pel within the path's boundary lines.

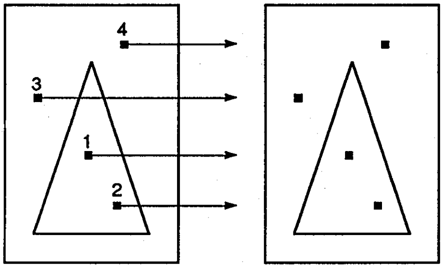
3 Draw an imaginary ray, in the *positive x*-direction, from that pel towards infinity.

4 Each time the ray intersects a boundary line with a *positive y*-component, increment a count value, and each time the ray intersects a boundary line with a *negative y*-component, decrement the count value.

5 Highlight the pel if the count value is nonzero.

Figure 35.7 shows how MS OS/2 would perform this test on the path shown in Figure 35.5:

Figure 35.7
Winding-Mode Test

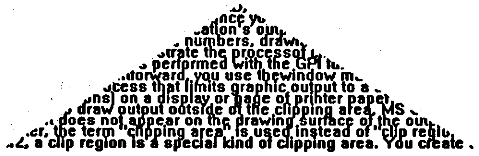


35.2.4 Clip Paths

Clipping is the process of discarding output that shouldn't appear in a window or on a page of printer paper. Normally, a clip region is a rectangle or a polygon defined by intersecting rectangles.

You can create rectangular clip regions by calling the **GpiCreateRegion** and **GpiSetClipRegion** functions. But you can also create nonrectangular clip regions, called clip paths, by generating a path that matches the nonrectangular region and then setting that path to a clip region by calling the **GpiSetClipPath** function. Figure 35.8 shows the result of clipping text by using a triangular clip path:

Figure 35.8
Triangular Clip Path



nce yo
ation's own
numbers, drawn
strate the process of
performed with the GPI to
forward, you use the window m
cess that limits graphic output to a
ns) on a display or page of printer paper
draw output outside of the clipping area, MS
does not appear on the drawing surface of the out
er, the term "clipping area" is used instead of "clip regio
a clip region is a special kind of clipping area. You create

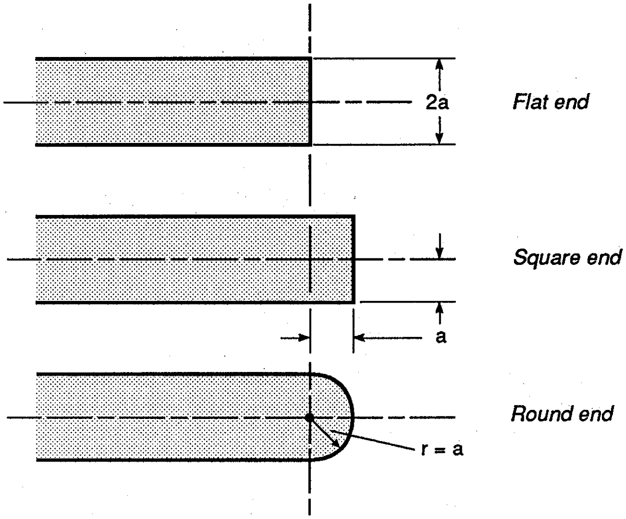
35.2.5 Path Attributes

Path attributes describe or identify a path's fill pattern, its color, and how it is drawn. If the path is a geometric line, several additional attributes describe its width, line-end style, and line-join style.

The line-width attribute specifies the width (in world units) of the geometric line. A world unit is a unit of measure in the world coordinate system. To set the width of a geometric line, you can use the **GpiSetLineWidthGeom** function.

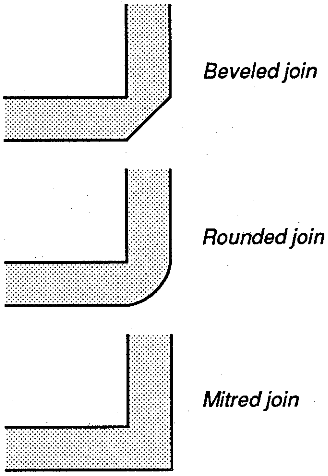
The line-end-style attribute specifies the shape of the end of the geometric line. You can draw geometric lines with square, flat, or round ends. To set the line-end style, you can use the **GpiSetLineEnd** function. (The round and square ends extend past the starting and ending points of a line; a flat end does not.) Figure 35.9 shows the three line-end styles:

Figure 35.9
Geometric Line Ends



The line-join-style attribute specifies the shape formed by two intersecting geometric lines. You can select a beveled, rounded, or mitred line-join style. To set the line-join style, you can use the **GpiSetLineJoin** function. Figure 35.10 shows the three line-join styles:

Figure 35.10
Geometric Line Joins



The pattern that fills a path or geometric line can be a bitmap that measures 8 bits by 8 bits or a character from one of the character sets. MS OS/2 uses the pattern or character to fill the interior of the path that defines the geometric line. If the pattern is a bitmap, you can alter its appearance by shifting its reference point, a point that the system uses to align the bitmap each time it copies the bitmap into the pattern.

The color of the fill pattern depends on the foreground and background mix modes as well as the foreground and background pattern colors. The mix modes tell MS OS/2 how to combine the foreground and background pattern colors with the existing color on the drawing surface. For more information about foreground and background mix modes, see Chapter 34, "Color and Mix Modes."

When you create a presentation space, the geometric-line attributes are set to the following default values:

Geometric-line attribute	Default value
Line width	1 world unit
Line-end style	LINEEND_FLAT
Line-join style	LINEJOIN_BEVEL
Pattern symbol	Solid rectangle
Foreground color	CLR_NEUTRAL (black on most devices)
Background color	CLR_BACKGROUND (white on most devices)
Foreground mix mode	FM_OVERPAINT
Background mix mode	BM_LEAVEALONE
Reference point	(x = 0, y = 0)

You can retrieve the current geometric-line attributes by calling the **GpiQueryAttrs** function. This function copies the current attributes into one of the five attribute-bundle structures: **AREABUNDLE**, **CHARBUNDLE**, **IMAGEBUNDLE**, **LINEBUNDLE**, and **MARKERBUNDLE**. The **LINEBUNDLE** and **AREABUNDLE** structures contain the geometric-line attributes that affect the appearance of paths. The **LINEBUNDLE** structure has the following form:

```
typedef struct _LINEBUNDLE {    /* lbnd */
    LONG    lColor;
    LONG    lReserved;
    USHORT  usMixMode;
    USHORT  usReserved;
    FIXED   fxWidth;
    LONG    lGeomWidth;    /* geometric line width      */
    USHORT  usType;
    USHORT  usEnd;         /* geometric line-end style  */
    USHORT  usJoin;        /* geometric line-join style */
} LINEBUNDLE;
```

The **AREABUNDLE** structure has the following form:

```
typedef struct _AREABUNDLE { /* pbnd */
    LONG   lColor;          /* pattern foreground color    */
    LONG   lBackColor;      /* pattern background color    */
    USHORT usMixMode;       /* pattern mix mode            */
    USHORT usBackMixMode;   /* background pattern mix mode */
    USHORT usSet;           /* bitmap local identifier     */
    USHORT usSymbol;        /* character identifier        */
    POINTL ptlRefPoint ;    /* pattern reference point     */
} AREABUNDLE;
```

To change the current attributes, use the **GpiSetAttrs** function.

35.3 Using Paths

You can use path functions to perform the following tasks:

- Draw a geometric line.
- Draw a filled polygon.
- Create a clip path.

35.3.1 Drawing a Geometric Line

To draw a geometric line, you must perform the following steps:

1. Set the geometric-line width by calling the **GpiSetLineWidthGeom** function.
2. Set the geometric-line-end style calling the **GpiSetLineEnd** function.
3. Start a path by calling the **GpiBeginPath** function.
4. Draw the line(s) by calling the **GpiMove** and **GpiLine** functions.
5. End the path by calling the **GpiEndPath** function.
6. Draw the line by calling the **GpiStrokePath** function.

The following code fragment shows how to draw a straight line that is 10 units wide and has round ends:

```
GpiSetLineWidthGeom(hps, 10L);      /* sets line width to 10  */
GpiSetLineEnd(hps, LINEEND_ROUND);  /* sets line end to round */
GpiBeginPath(hps, 1L);              /* begins path            */
ptl.x = 7; ptl.y = 15;
GpiMove(hps, &ptl);                 /* sets current position  */
ptl.x = 450; ptl.y = 15;
GpiLine(hps, &ptl);                 /* draws line             */
GpiEndPath(hps);                    /* ends path              */
GpiStrokePath(hps, 1L, 0L);         /* draws wide line        */
```

35.3.2 Drawing a Filled Polygon

To draw a filled polygon, you must perform the following steps:

1 Start a path by calling the **GpiBeginPath** function.

2 Move to the starting point by calling the **GpiMove** function.

3 Draw the boundary lines by calling the appropriate line-drawing function.

4 End the path by calling the **GpiEndPath** function.

5 Specify a fill mode and fill the path by calling the **GpiFillPath** function.

The following code fragment shows how to draw an empty triangle within a solid rectangle:

```
POINTL aptl1 [4] = {           /* array of points for triangle  */
    50, 50,
    100, 100,
    150, 50,
    50, 50 };

POINTL aptl2[5] = {            /* array of points for rectangle */
    25, 25,
    25, 200,
    200, 200,
    200, 25,
    25, 25 };

GpiBeginPath(hps, 1L);    /* begins path                 */
GpiMove(hps, aptl1);      /* sets current position       */
GpiPolyLine(hps, 4,
    aptl1);               /* plots points for triangle   */
GpiMove(hps, aptl2);      /* sets current position       */
GpiPolyLine(hps, 5,
    aptl2);               /* plots points for rectangle  */
GpiEndPath(hps);          /* ends path                   */
GpiFillPath(hps, 1L,
    FPATH_ALTERNATE);     /* draws triangle and rectangle */
```

35.3.3 Creating a Clip Path

To create a clip path, you must perform the following steps:

1 Start a path by calling the **GpiBeginPath** function.

2 Draw the border of the path by calling the appropriate line or arc functions.

3 End the path by calling the **GpiEndPath** function.

4 Create the clip path by calling the **GpiSetClipPath** function.

The following code fragment shows how to create a triangular clip path and then write text into it:

```
POINTL aptl[4] = {            /* array of points for triangle */
    35, 45,
    100, 100,
    200, 45,
    35, 45 };

GpiBeginPath(hps, 1L);       /* begins path bracket             */
GpiMove(hps, aptl);          /* sets current position           */
GpiPolyLine(hps, 4, aptl);   /* plots points for triangle       */
GpiEndPath(hps);             /* ends path bracket               */

GpiSetClipPath(hps, 1L, SCP_ALTERNATE | SCP_AND); /* sets clip path */

ptlStart.x = 50;
for (i = 50; i < 110; i = i + 10) {
    ptlStart.y = i;

    /* write the text */

    GpiCharStringAt(hps, &ptlStart, 18, "?!String of text!?");
  }
```

35.4 Summary

The following list summarizes the MS OS/2 path functions:

GpiBeginPath Begins a path definition.

GpiEndPath Ends a path definition by closing an open path.

GpiFillPath Fills the interior of a path. MS OS/2 uses the attributes in the current area bundle when filling the interior.

GpiModifyPath Modifies a path. Once you have modified the path, you can use it to draw geometric-line output by calling the **GpiFillPath** function.

GpiStrokePath Draws a geometric line.

Chapter

36

Area Primitives

36.1 Introduction 495

36.2 About Areas and Area Primitives 495

36.2.1 Creating Areas 496

36.2.2 Outlining Areas 499

36.2.3 Filling Areas 500

36.3 Using Areas and Area Primitives 501

36.3.1 Drawing a Single Closed Figure 502

36.3.2 Drawing Multiple Intersecting Closed Figures 502

36.3.3 Drawing Multiple Disjoint Closed Figures 502

36.3.4 Creating a Custom Fill Pattern from a Bitmap 502

36.3.5 Creating a Custom Fill Pattern from a Font Character 503

36.4 Summary 504

36.1 Introduction

This chapter describes areas and area primitives. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Color and mix modes
- Fonts and text output
- Paths

36.2 About Areas and Area Primitives

An area primitive is one or more closed figures that are filled or filled *and* outlined. You can define an area primitive as a distinct closed figure, as multiple intersecting or disjoint closed figures, or as some combination of these. An area primitive is the result of a collection of functions that define and draw the area. Figure 36.1 shows two areas—a five-pointed star in the lower-left corner, and two intersecting boxes in the upper-right corner:

Figure 36.1
Two Areas

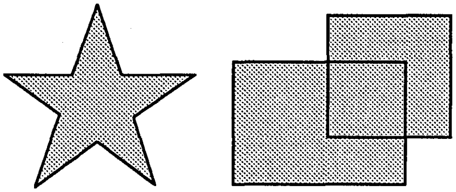
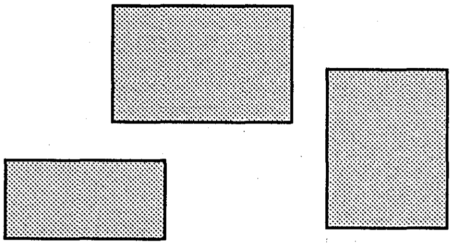


Figure 36.2 shows a single area that contains three disjoint boxes:

Figure 36.2
Disjoint Area



36.2.1 Creating Areas

An area bracket contains the functions that define an area primitive. A **GpiBeginArea** function defines the beginning of an area bracket; a **GpiEndArea** function defines the end of the area bracket and tells MS OS/2 to draw the area. An area bracket has the following form:

```
GpiBeginArea();                /* start of area bracket         */
    .
    .                          /* line and arc functions        */
    .
GpiEndArea();                  /* end of area bracket, draws area */
```

Once you define the beginning of an area bracket, you can define its shape and its location in your application's world space by using line and arc functions. These area functions are described in the following list:

Area functions	Description
GpiBox	Draws a rectangle.
GpiFullArc	Draws an ellipse.
GpiLine	Draws a straight line.
GpiPartialArc	Draws part of an ellipse defined by an angle.
GpiPointArc	Draws part of an ellipse defined by three points.
GpiPolyFillet	Draws a series of connected fillets.
GpiPolyFilletSharp	Draws a series of fillets, with each curve approximating an ellipse, a hyperbola, or a parabola.
GpiPolyLine	Draws a series of connected straight lines.
GpiPolySpline	Draws a series of connected splines.

Always use the DRO_OUTLINE constant if you call either the **GpiBox** or **GpiFullArc** function to define the shape of an area; if you use the DRO_FILL or DRO_OUTLINEFILL constant, MS OS/2 returns an error message.

If you do not close a figure before calling **GpiEndArea**, MS OS/2 automatically closes it for you by drawing a line from the end point of the last line or curve drawn to the starting point of the first line or curve drawn. The current position is always updated to the end point of the last line drawn.

In addition to using line and arc functions in an area bracket, you can also use the following functions:

- **GpiBeginElement**
- **GpiBox**
- **GpiCallSegmentMatrix**
- **GpiComment**

- **GpiDeleteElement** (retain mode only)
- **GpiDeleteElementRange** (retain mode only)
- **GpiDeleteElementsBetweenLabels** (retain mode only)
- **GpiElement**
- **GpiEndArea**
- **GpiEndElement**
- **GpiFullArc**
- **GpiGetData**
- **GpiLabel**
- **GpiLine**
- **GpiMove**
- **GpiOffsetElementPointer**
- **GpiPartialArc**
- **GpiPointArc**
- **GpiPolyFillet**
- **GpiPolyFilletSharp**
- **GpiPolyLine**
- **GpiPolySpline**
- **GpiPop**
- **GpiPutData**
- **GpiQueryArcParams** (not valid in retain mode)
- **GpiQueryAttrMode**
- **GpiQueryAttrs** (not valid in retain mode)
- **GpiQueryBackColor** (not valid in retain mode)
- **GpiQueryBackMix**
- **GpiQueryBoundaryData**
- **GpiQueryCharAngle** (not valid in retain mode)
- **GpiQueryCharBox** (not valid in retain mode)
- **GpiQueryCharDirection** (not valid in retain mode)
- **GpiQueryCharMode** (not valid in retain mode)
- **GpiQueryCharSet** (not valid in retain mode)
- **GpiQueryCharShear** (not valid in retain mode)
- **GpiQueryCharStringPos** (not valid in retain mode)
- **GpiQueryCharStringPosAt** (not valid in retain mode)
- **GpiQueryClipBox**
- **GpiQueryClipRegion**

- **GpiQueryColor** (not valid in retain mode)
- **GpiQueryColorData**
- **GpiQueryColorIndex**
- **GpiQueryCp**
- **GpiQueryCurrentPosition** (not valid in retain mode)
- **GpiQueryDefaultViewMatrix**
- **GpiQueryDefCharBox**
- **GpiQueryDevice**
- **GpiQueryDeviceBitmapFormats**
- **GpiQueryEditMode**
- **GpiQueryElement** (valid only in retain mode)
- **GpiQueryElementPointer** (valid only in retain mode)
- **GpiQueryElementType** (valid only in retain mode)
- **GpiQueryFontFileDescriptions**
- **GpiQueryFontMetrics**
- **GpiQueryFonts**
- **GpiQueryGraphicsField**
- **GpiQueryInitialSegmentAttrs**
- **GpiQueryKerningPairs**
- **GpiQueryLineEnd** (not valid in retain mode)
- **GpiQueryLineJoin** (not valid in retain mode)
- **GpiQueryLineType** (not valid in retain mode)
- **GpiQueryLineWidth** (not valid in retain mode)
- **GpiQueryLineWidthGeom** (not valid in retain mode)
- **GpiQueryLogColorTable**
- **GpiQueryMarker** (not valid in retain mode)
- **GpiQueryMarkerBox** (not valid in retain mode)
- **GpiQueryMarkerSet** (not valid in retain mode)
- **GpiQueryMix** (not valid in retain mode)
- **GpiQueryModelTransformMatrix** (not valid in retain mode)
- **GpiQueryNearestColor**
- **GpiQueryNumberSetIds**
- **GpiQueryPageViewport**
- **GpiQueryPattern** (not valid in retain mode)
- **GpiQueryPatternRefPoint** (not valid in retain mode)
- **GpiQueryPatternSet** (not valid in retain mode)

- **GpiQueryPel**
- **GpiQueryPickAperturePosition**
- **GpiQueryPickApertureSize**
- **GpiQueryRealColors**
- **GpiQueryRegionBox**
- **GpiQueryRegionRects**
- **GpiQueryRGBColor**
- **GpiQuerySegmentAttrs**
- **GpiQuerySegmentNames**
- **GpiQuerySegmentPriority**
- **GpiQuerySegmentTransformMatrix**
- **GpiQuerySetIds**
- **GpiQueryStopDraw**
- **GpiQueryTag** (not valid in retain mode)
- **GpiQueryViewingLimits** (not valid in retain mode)
- **GpiQueryViewingTransformMatrix**
- **GpiQueryWidth**
- **GpiSetArcParams**
- **GpiSetAttrMode**
- **GpiSetAttrs**
- **GpiSetColor**
- **GpiSetCurrentPosition**
- **GpiSetEditMode**
- **GpiSetElementPointer** (not valid in retain mode)
- **GpiSetElementPointerAtLabel** (not valid in retain mode)
- **GpiSetLineEnd**
- **GpiSetLineJoin**
- **GpiSetLineType**
- **GpiSetLineWidth**
- **GpiSetMix**
- **GpiSetModelTransformMatrix**
- **GpiSetSegmentTransformMatrix**

36.2.2 Outlining Areas

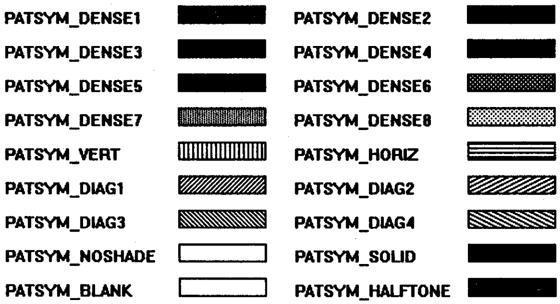
If you want MS OS/2 to outline an area, use the BA_BOUNDARY flag when you call the **GpiBeginArea** function. MS OS/2 then draws the outline, using a narrow line that is 1 pel wide. The default outline color is black (CLR_NEUTRAL) on most displays and printers and the default line style is solid (LINETYPE_SOLID). You can change the line color and line styles by calling the **GpiSetAttrs** function, or to determine the current line color and line styles, you can call the **GpiQueryAttrs** function. For more information about lines and line primitives, see Chapter 32, "Line and Arc Primitives."

If you do not want MS OS/2 to outline an area, use the BA_NOBOUNDARY flag when you call **GpiBeginArea**.

36.2.3 Filling Areas

MS OS/2 fills an area by repeatedly drawing an eight-by-eight bitmap called a fill pattern. The default fill pattern is solid (PATSYM_DENSE1), and the default fill-pattern color is black (CLR_NEUTRAL). MS OS/2 provides 15 predefined fill patterns, in addition to the default pattern. Figure 36.3 shows each of the 16 predefined patterns:

Figure 36.3
Predefined Fill Patterns



To determine which of the 16 patterns is currently selected, you can call the **GpiQueryPattern** function, or to select a new pattern, you can call the **GpiSetPattern** function. You can also create custom patterns by using bitmaps or characters and symbols from an image font. If you do create a custom pattern, remember that MS OS/2 uses only the first eight bits in the first eight rows of the bitmap, starting in the lower-left corner.

You can change the foreground and background colors and mix modes for the fill pattern by calling the **GpiSetAttrs** function. The foreground color corresponds to the color of the bits that are set in the fill-pattern bitmap. The background color corresponds to the color of the bits that are not set in the fill-pattern bitmap. The foreground mix mode specifies how MS OS/2 should combine the foreground color with the color on the drawing surface; the background mix mode specifies how MS OS/2 should combine the background color with the color on the drawing surface. You can determine the current foreground and background color and mix-mode settings by calling the **GpiQueryAttrs** function. For more information about color and mix modes, see Chapter 34, "Color and Mix Modes."

To create a custom pattern from a hard-coded bitmap, you can call the **GpiSetPatternSet** function, passing it the local identifier (*lcid*) for the bitmap. This is the identifier you assigned when you called the **GpiSetBitmapId** function.
For more information about creating bitmaps, see Chapter 38, "Bitmaps."

To create a custom pattern from a character or a symbol in a font, you can call the **GpiSetPatternSet** function, passing it the local identifier for the font. This is the same identifier that you passed to the **GpiCreateLogFont** function when you created the font. You then need to specify the code point for a character or symbol in that font by calling the **GpiSetPattern** function and passing the value of the code point. For more information about selecting and using fonts, see Chapter 33, "Fonts and Character Primitives."

You can alter the alignment of the fill pattern in an area by altering the pattern reference point. The default pattern reference point is (0,0). When the default reference point is set, MS OS/2 aligns the lower-left corner of the fill pattern with the point (0,0) in the application's world space and begins filling the area. If you adjust the pattern reference point to (3,2), MS OS/2 aligns the lower-left corner of the fill pattern with the point (3,2) in the application's world space and begins filling the area. By moving the reference point from (0,0) to (3,2), the fill pattern appears shifted up 2 pels and to the right 3 pels.

A special structure called the **AREABUNDLE** structure contains seven fields that specify the current fill-pattern colors and mix modes, the local identifier for the bitmap or font, the character code point, and the pattern reference point.

The **AREABUNDLE** structure has the following form:

```
typedef struct _AREABUNDLE {     /* pbnd                       */
    LONG     lColor;             /* foreground color           */
    LONG     lBackColor;         /* background color           */
    USHORT   usMixMode;          /* foreground mix mode        */
    USHORT   usBackMixMode;      /* background mix mode        */
    USHORT   usSet;              /* bitmap or font lcid        */
    USHORT   usSymbol;           /* character or symbol code point */
    POINTL   ptlRefPoint;        /* pattern reference point    */
} AREABUNDLE
```

To determine the current values in these fields, you can call the **GpiQueryAttrs** function, or to change the values, call the **GpiSetAttrs** function.

When filling an area, MS OS/2 uses not only a fill pattern but a fill mode, as well. The two fill modes, alternate and winding, affect how the area is filled. For more information about fill modes, see Chapter 35, "Paths."

36.3 Using Areas and Area Primitives

You can use area functions to perform the following tasks:

- Draw a single closed figure.
- Draw multiple intersecting closed figures.
- Draw multiple disjoint closed figures.
- Draw any combination of the three.
- Create a custom fill pattern from a bitmap.
- Create a custom fill pattern from a font character.

36.3.1 Drawing a Single Closed Figure

The following code fragment shows how to use area functions to draw a single closed figure that is filled using the alternate mode. The closed figure in this example is a five-pointed star.

```
/* Initialize the array of points for the 5-pointed star. */

aptl[0].x = 37; aptl[0].y = 82;
aptl[1].x = 400; aptl[1].y = 195;
aptl[2].x = 40; aptl[2].y = 320;
aptl[3].x = 260; aptl[3].y = 10;
aptl[4].x = 260; aptl[4].y = 390;
aptl[5].x = 37; aptl[5].y = 82;

/* Draw the star. */

GpiBeginArea(hps, BA_ALTERNATE);
GpiMove(hps, aptl);
GpiPolyLine(hps, 6L, aptl);
GpiEndArea(hps);
```

36.3.2 Drawing Multiple Intersecting Closed Figures

The following code fragment shows how to use area functions to draw two intersecting boxes that are filled using the winding mode:

```
GpiBeginArea(hps, BA_WINDING);
ptl.x = 500; ptl.y = 300;
GpiMove(hps, &ptl);
ptl.x = 700; ptl.y = 500;
GpiBox(hps, DRO_OUTLINE, &ptl, OL, OL);
ptl.x = 580; ptl.y = 370;
GpiMove(hps, &ptl);
ptl.x = 780; ptl.y = 570;
GpiBox(hps, DRO_OUTLINE, &ptl, OL, OL);
GpiEndArea(hps);
```

36.3.3 Drawing Multiple Disjoint Closed Figures

The following code fragment shows how to use area functions to draw two disjoint boxes that are filled using the winding mode:

```
GpiBeginArea(hps, BA_WINDING);
ptl.x = 100; ptl.y = 200;
GpiMove(hps, &ptl);
ptl.x = 200; ptl.y = 400;
GpiBox(hps, DRO_OUTLINE, &ptl, OL, OL);
ptl.x = 580; ptl.y = 470;
GpiMove(hps, &ptl);
ptl.x = 780; ptl.y = 570;
GpiBox(hps, DRO_OUTLINE, &ptl, OL, OL);
GpiEndArea(hps);
```

36.3.4 Creating a Custom Fill Pattern from a Bitmap

The following code fragment shows how to create a custom fill pattern by using a hard-coded bitmap, which in this example creates a cross-hatch pattern:

```
CreatePattern();
GpiSetPatternSet(hps, lcidCustom);
     .
     .
     .
CreatePattern()
```

```
{
    /* Bitmap information */

    HBITMAP hbm;               /* bitmap handle                      */
    BITMAPINFOHEADER bmp;      /* structure for bitmap information */
    PBITMAPINFO pbmi;          /* pointer to structure for data     */
    CHAR cBuffer[80];          /* structure template                 */

    bmp.cbFix = 12;          /* length of structure */
    bmp.cx = 8;              /* 8 pels wide          */
    bmp.cy = 8;              /* 8 pels high          */
    bmp.cPlanes = 1;         /* 1 plane              */
    bmp.cBitCount = 1;       /* 1 bit per pel        */

    /*
     * Initialize the bitmap data by loading "cBuffer" with the bitmap
     * information.
     */

    pbmi = (PBITMAPINFO) cBuffer;
    pbmi->cbFix = 12;         /* length of structure */
    pbmi->cx = 8;             /* 8 bits wide          */
    pbmi->cy = 8;             /* 8 bits high          */
    pbmi->cPlanes = 1;        /* 1 plane              */
    pbmi->cBitCount = 1;      /* 1 bit per pel        */

    /* Initialize first two color-table entries to black and white. */

    pbmi->argbColor[0].bRed = 0;          /* Color[0] = black */
    pbmi->argbColor[0].bGreen = 0;        /* Color[0] = black */
    pbmi->argbColor[0].bBlue = 0;         /* Color[0] = black */
    pbmi->argbColor[1].bRed = 255;        /* Color[1] = white */
    pbmi->argbColor[1].bGreen = 255;      /* Color[1] = white */
    pbmi->argbColor[1].bBlue = 255;       /* Color[1] = white */

    /* Create a bitmap and retrieve its handle. */

    hbm = GpiCreateBitmap(hps,
        &bmp,
        CBM_INIT,
        (PBYTE) abPattern5,    /* array of bits */
        pbmi);

    /* Tag the bitmap just created with a custom identifier (lcid). */

    GpiSetBitmapId(hps, hbm, lcidCustom);
}
```

36.3.5 Creating a Custom Fill Pattern from a Font Character

The following code fragment shows how to create a custom fill pattern by using a character from a font:

```
LoadFont();
GpiSetPatternSet(hps, lcidCustom);
GpiSetPattern(hps, lCodePoint);
    .
    .
    .
LoadFont()
{
    /* Determine the number of loaded public fonts. */

    cFonts = GpiQueryFonts(hps, QF_PUBLIC, NULL, &lCount,
        (LONG) (sizeof(fm)), (PFONTMETRICS) afm);

    /* Load the metrics for all public fonts into afm. */

    cFonts = GpiQueryFonts(hps, QF_PUBLIC, NULL, &lCount,
        (LONG) (sizeof(fm)), (PFONTMETRICS) afm);
    /* Grab the first image font with a point size larger than 8. */

    for(i = 1;
        (afm[i].fsDefn % 2 != 0) &&
        (afm[i].sNominalPointSize / 10 >= 8);
        i++);

    /* Load the FATTRS structure with corresponding metrics. */

    fat.usRecordLength = sizeof(fat);
    fat.fsSelection = afm[i].fsSelection;
    fat.lMatch = afm[i].lMatch;
    strcpy(fat.szFacename, afm[i].szFacename);
    fat.idRegistry = afm[i].idRegistry;
    fat.usCodePage = afm[i].usCodePage;
    fat.lMaxBaselineExt = afm[i].lMaxBaselineExt;
    fat.lAveCharWidth = afm[i].lAveCharWidth;
    fat.fsType = afm[i].fsType;
    fat.fsFontUse = 0;

    /* Ask MS OS/2 to select this font and assign it a custom lcid. */

    GpiCreateLogFont(hps, (PSTR8) cBuffer, lcidCustom, (PFATTRS) &fat);
    GpiSetCharSet(hps, lcidCustom);
}
```

36.4 Summary

The following list summarizes the MS OS/2 area functions:

GpiBeginArea Starts an area bracket. Its second argument is a combination of two of four possible flags. Two of these flags affect how MS OS/2 draws the border of the area: The BA_NOBOUNDARY flag specifies that MS OS/2 should fill the interior of the area but not its border; the BA_BOUNDARY flag specifies that MS OS/2 should fill the interior of the area *and* its border. The other two flags determine which mode MS OS/2 uses to fill the area: The BA_WINDING flag specifies that MS OS/2 should fill the interior of the area by using the winding mode; the BA_ALTERNATE flag specifies that MS OS/2 should fill the interior of the area by using the alternate mode.

GpiEndArea Ends an area bracket. When you call **GpiEndArea**, MS OS/2 draws an area by using the flags that you specified when calling **GpiBeginArea** and the primitives specified between the **GpiBeginArea** and **GpiEndArea** function calls.

GpiQueryPattern Retrieves a value that identifies the current pattern symbol.

GpiQueryPatternRefPoint Retrieves the coordinates of the current pattern reference point.

GpiQueryPatternSet Retrieves a local identifier (*lcid*) that identifies a bitmap or a font.

GpiSetPattern Sets the pattern-symbol attribute to a value that identifies one of the 16 predefined patterns, or sets the attribute to a code point for a character or symbol in a font.

GpiSetPatternRefPoint Sets the pattern reference point. The default reference point is (0,0); when you move the reference point, you shift the entire fill pattern within the area.

GpiSetPatternSet Sets the pattern attribute to a local identifier (*lcid*) that identifies a bitmap or a font.

Chapter

37

Marker Primitives

37.1 Introduction 507

37.2 About Marker Primitives 507

37.2.1 The Default Marker and the Default Marker Set 507

37.2.2 Custom Markers 508

37.2.3 Image and Vector Markers 508

37.2.4 Marker Colors 508

37.2.5 Drawing Markers 509

37.3 Using Markers 509

37.3.1 Drawing a Series of Markers 509

37.3.2 Selecting a New Marker 509

37.3.3 Selecting a New Marker Set 510

37.3.4 Changing the Marker Color 510

37.4 Summary 510

37.1 Introduction

This chapter describes a graphics object called a marker primitive. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Color and mix modes
- Fonts

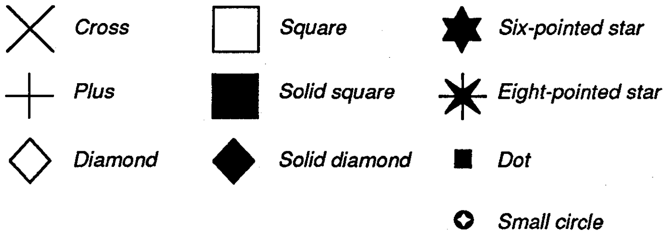
37.2 About Marker Primitives

A marker or a symbol is a character or a symbol that is always drawn centered over a point. Applications typically use marker primitives to identify points of interest such as points on a graph.

37.2.1 The Default Marker and the Default Marker Set

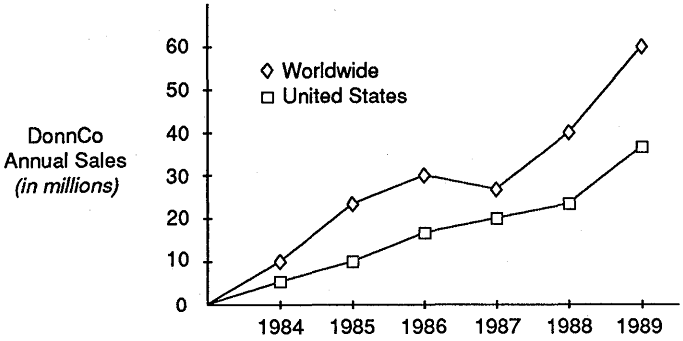
When you create a presentation space, it contains the default marker—a cross—from the default marker set. The default marker set contains eleven image characters, which are drawn by setting pels in a rectangular region called a marker box. Figure 37.1 shows ten characters from the default marker set (the eleventh character is an invisible marker):

Figure 37.1
Default Marker Set



Within the marker box, the color of the set pels defines the foreground color, whose default color is neutral (black on the display and on printers); the color of the pels that are not set defines the background color, whose default color is the background color on the device (white on the display, and the paper color on printers). The height and width of the default marker are device dependent. You can retrieve these dimensions by calling the **DevQueryCaps** function. Figure 37.2 shows a graph drawn using the default marker:

Figure 37.2
Markers Used in a Graph



37.2.2 Custom Markers

You can use any font's character set as a marker set and any character within that marker set as a marker. To retrieve the value that identifies the current marker set, call the **GpiQueryMarkerSet** function. To retrieve the value that identifies the current marker character, call the **GpiQueryMarker** function. You can call the **GpiQueryAttrs** function to retrieve both values simultaneously.

If the current marker set does not contain a symbol that suits your application, you can load a new character set and select a marker from that set. For a description of loading character sets, see Chapter 33, "Fonts and Character Primitives." Once you have loaded a new character set, you can select it as a marker set by calling the **GpiSetMarkerSet** function. After selecting the marker set, call the **GpiSetMarker** function to select a marker from the set. You can call the **GpiSetAttrs** function to set both values simultaneously.

37.2.3 Image and Vector Markers

If the current marker set contains image characters, the marker-box dimensions are fixed, since you cannot alter the size of image markers. If the current marker set contains vector characters (characters outlined by using line and arc functions), you can change the size of the marker box by calling the **GpiSetMarkerBox** or **GpiSetAttrs** function.

37.2.4 Marker Colors

When you draw a marker, three colors affect its appearance—the foreground color, the background color, and the current color of the drawing surface. For image markers, the colors were explained in Section 37.2.1. For vector markers, the foreground color defines the color of the lines, arcs, and fill patterns that draw the marker, and the background color defines the color of the remainder of the marker box. The drawing-surface color is the color of the drawing surface on the output device associated with the application's presentation space. In addition to the foreground, background, and drawing-surface colors, two mix modes—foreground and background—affect the appearance of the marker. The foreground mix mode specifies a color mix mode between the foreground marker color and the drawing-surface color. The background mix mode specifies a color mix mode between the background marker color and the drawing-surface color. (For a more complete description of colors and mix modes, see Chapter 34, "Color and Mix Modes.") You can determine the current marker colors and mix modes by calling the **GpiQueryAttrs** function. You can set new colors and mix modes by calling the **GpiSetAttrs** function.

37.2.5 Drawing Markers

You can draw a single marker or a series of markers. To draw a single marker, you must set the fields in a **POINTL** structure to correspond to the marker position in world coordinates and then call the **GpiMarker** function, passing it the address of the **POINTL** structure as the second argument. To draw a series of markers, you must set the fields in an array of **POINTL** structures to correspond to the marker positions in world coordinates and then call the **GpiPolyMarker** function, passing it the number of points in the array as the second argument and the name of the array as the third argument.

37.3 Using Markers

You can use marker functions to perform the following tasks:

- Draw a single marker or a series of markers.
- Determine the local identifier (*lcid*) for the current marker set.
- Select a character set as the new marker set.
- Determine the value that identifies the current marker.
- Select a character as the new marker.
- Set or change the size of the marker box.
- Set or change the color of a marker.

37.3.1 Drawing a Series of Markers

The following code fragment shows how to draw a graph by calling the **GpiPolyLine** and **GpiPolyMarker** functions:

```
HPS hps;           /* presentation-space handle */
POINTL aptl[6];    /* array of points           */
    .
    .
    .
aptl[0].x = 10; aptl[0].y = 15;          /* assigns points       */
aptl[1].x = 150; aptl[1].y = 30;
aptl[2].x = 200; aptl[2].y = 32;
aptl[3].x = 250; aptl[3].y = 70;
aptl[4].x = 360; aptl[4].y = 120;
aptl[5].x = 380; aptl[5].y = 98;
GpiMove(hps, aptl);                     /* sets current position */
GpiPolyMarker(hps, 6L, aptl);           /* plots points          */
GpiPolyLine(hps, 6L, aptl);             /* draws lines           */
```

37.3.2 Selecting a New Marker

The following code fragment shows how to check whether the default marker set and marker are currently being used. If they are, it replaces the default marker with a six-pointed star:

```
if ((GpiQueryMarker(hps) == MARKSYM_DEFAULT) &&
        (GpiQueryMarkerSet(hps) == LCID_DEFAULT))
    GpiSetMarker(hps, MARKSYM_SIXPOINTSTAR);
```

37.3.3 Selecting a New Marker Set

The following code fragment shows how to load a Helvetica vector font, use it as the new marker set, and select the smile-face character as the new marker primitive:

```
GpiLoadFonts(hps, "helv");                   /* loads helv.dll     */
lFontCount = GpiQueryFonts(hps,              /* loads array of fm  */
   0x00000002, "Helvetica", &lCount,
   (LONG) (sizeof(fm)), (PFONTMETRICS) afm);
fat.usRecordLength = sizeof(fat);            /* sets record length */
fat.usCodePage = 850;                        /* sets code page     */
fat.lMatch = afm[0].lMatch;                  /* uses first metrics */
fat.fsFontUse = FATTR_FONTUSE_TRANSFORMABLE; /* uses vector font   */
GpiCreateLogFont(hps, (PSTR8) buffer, ++lcid,
   (PFATTRS) &fat);
mbnd.usSet = lcid;                      /* uses font as marker set */
mbnd.usSymbol = 2;                      /* uses smile character    */
GpiSetAttrs(hps, PRIM_MARKER,
   MBB_SYMBOL | MBB_SET,
   0L, &mbnd);
```

37.3.4 Changing the Marker Color

The following code fragment shows how to set the marker foreground color to green:

```
mbnd.lColor = CLR_GREEN;
GpiSetAttrs(hps, PRIM_MARKER, MBB_COLOR, 0L, &mbnd);
```

37.4 Summary

The following list summarizes the MS OS/2 marker functions:

GpiMarker Draws a marker by using the current marker attributes.

GpiPolyMarker Draws a series of markers by using the current marker attributes.

GpiQueryAttrs Retrieves the values of the current marker attributes.

GpiQueryMarker Retrieves the current marker-symbol attribute.

GpiQueryMarkerBox Retrieves the size of the current marker box.

GpiQueryMarkerSet Retrieves the value of the current marker-set attribute.

GpiSetAttrs Sets the fields in the **MARKERBUNDLE** structure.

GpiSetMarker Sets the attribute of the marker symbol.

GpiSetMarkerBox Sets the dimensions of the marker box. (This function is only meaningful for vector markers; the width and height of raster markers are fixed.)

GpiSetMarkerSet Sets the attribute of the marker set.

Chapter

38

Bitmaps

38.1 Introduction 513

38.2 About Bitmaps 513

38.2.1 Bitmap Dimensions 514

38.2.2 Storing Color Information in Bitmaps 515

38.2.2.1 Color Planes 515

38.2.2.2 Bitcounts 515

38.2.3 Creating Bitmaps 516

38.2.4 Creating and Loading Custom Bitmaps 517

38.2.5 Drawing Bitmapped Images 518

38.2.6 Copying Images from a Display into a Bitmap 519

38.2.7 Saving Bitmaps in a File 520

38.3 Using Bitmaps 520

38.3.1 Copying an Image from a Video Display to a Bitmap 520

38.3.2 Scaling and Drawing a Bitmapped Image 522

38.3.3 Creating a Custom Fill Pattern 522

38.3.4 Loading a Bitmap from a File 523

38.3.5 Storing a Bitmap in a Metafile 523

38.4 Summary 524

38.1 Introduction

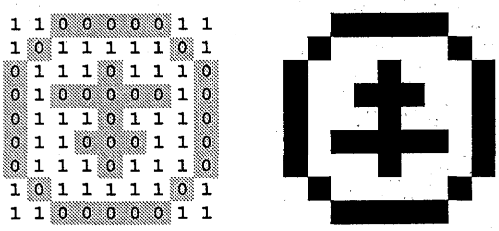
This chapter describes a graphic object called a bitmap. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Coordinate spaces
- Color and mix modes
- Area primitives
- Paths

38.2 About Bitmaps

A bitmap is an array of bits that represents an image. Applications can use bitmaps to store and display scanned images, icons, and symbols, and to create fill patterns for area primitives and paths. You can display a bitmap image on a raster output device (a raster is a rectangular matrix of pels or picture elements on a video display or dot matrix printer). A raster output device displays an image by setting adjacent pels in its matrix to colors specified in a corresponding bitmap. An image created in this way is called a "bitmapped image." Figure 38.1 shows an array of bits, on the left, and its corresponding bitmapped image, on the right:

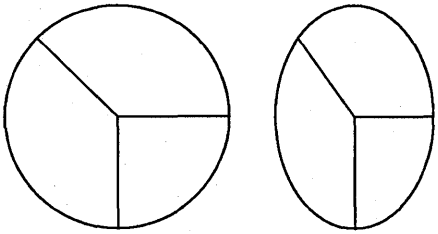
Figure 38.1
Bitmap and Image



```
1 1 0 0 0 0 0 1 1
1 0 1 1 1 1 1 0 1
0 1 1 1 0 1 1 1 0
0 1 0 0 0 0 0 1 0
0 1 1 1 0 1 1 1 0
0 1 1 0 0 0 1 1 0
0 1 1 1 0 1 1 1 0
1 0 1 1 1 1 1 0 1
1 1 0 0 0 0 0 1 1
```

Bitmapped images are device dependent. The shape of a device's pels, as well as its color capabilities, affects the appearance of a bitmapped image. For example, if the pels on one display measure 0.05 mm by 0.1 mm and the pels on a second display measure 0.1 mm by 0.3 mm, a circular bitmapped pie chart that you draw on the first display will appear elliptical on the second. If the first display supports 16 colors, the second supports 2 colors, and you created the pie chart using 12 colors, you will lose critical color information if you draw the chart on the second display. Figure 38.2 shows the pie chart drawn twice: The chart on the left was drawn on the first display; the chart on the right was drawn on the second display.

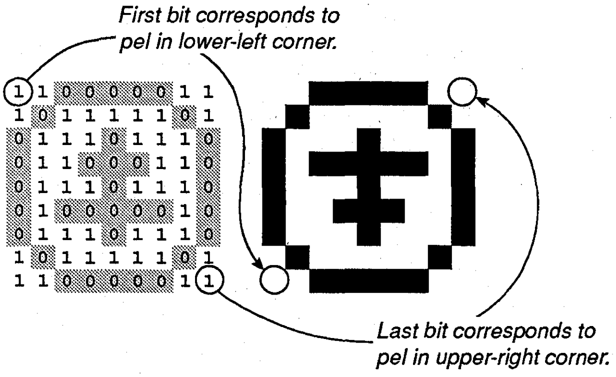
Figure 38.2
Bitmap Shown on Two Displays with Different Aspect Ratios



38.2.1 Bitmap Dimensions

Each row of pels in a bitmapped image corresponds to a row of bits in a bitmap. In MS OS/2, bitmap rows are padded so that they end on **ULONG** (32-bit) boundaries. Pels in a bitmapped image are numbered beginning in the lower-left corner, moving to the right across the first row, then proceeding up, row-by-row, from left to right to the last row. In a bitmapped image, the first pel is in the lower-left corner; the last pel is in the upper-right corner. The first byte of the bitmap contains the color information for the first pel; the last byte contains the color information for the last pel. Figure 38.3 shows the relationship between bitmap bits and pels in a bitmapped image:

Figure 38.3
Bits and Pels in a Bitmapped Image



```
1 1 0 0 0 0 0 1 1
1 0 1 1 1 1 1 0 1
0 1 1 1 0 1 1 1 0
0 1 1 0 0 0 1 1 0
0 1 1 1 0 1 1 1 0
0 1 0 0 0 0 0 1 0
0 1 1 1 0 1 1 1 0
1 0 1 1 1 1 1 0 1
1 1 0 0 0 0 0 1 1
```

First bit corresponds to pel in lower-left corner.
Last bit corresponds to pel in upper-right corner.

When you create a bitmap by using the **GpiCreateBitmap** function, you specify the bitmap width and height in terms of pels in the bitmapped image: The width is the number of pels within a row; the height is the number of rows. You should store these dimensions in the **BITMAPINFO** and **BITMAPINFOHEADER** structures and pass their addresses to the **GpiCreateBitmap** function when you call it.

38.2.2 Storing Color Information in Bitmaps

Graphics systems use one of two formats for storing color information in bitmaps. The first format uses multiple color planes. The second format uses a single plane and a multiple bit count.

38.2.2.1 Color Planes

Bitmaps are arranged in color planes. A color plane is an array of bitmap bits that contain color information. The bitmaps in each of the previous illustrations used the single-plane format, which is the standard format for bitmaps in MS OS/2 applications. In this format, adjacent bitmap bits contain indices into either a special color table of RGB values or actual **RGB** structures; all of the color information resides in a single plane. Although no device drivers for MS OS/2 Presentation Manager currently support a multi-plane format, this may change. A common multi-plane bitmap format for a bitmapped image is the 3-plane format in which one plane corresponds to the red pels, another to the green pels, and a third to the blue pels. You can determine which color-plane format a device supports by calling the **GpiQueryDeviceBitmapFormats** function and examining the first value in each pair of values that it returns.

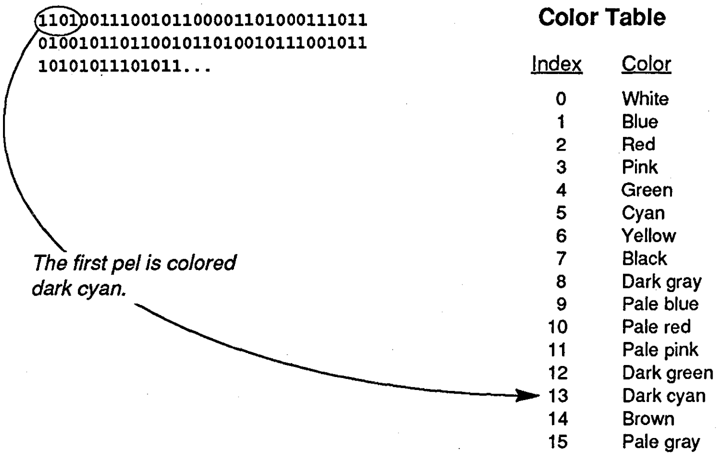
38.2.2.2 Bitcounts

A bitcount is a value that specifies how many adjacent bitmap bits correspond to each pel in a bitmapped image. There are four possible bitcounts:

- 1 bit per pel
- 4 bits per pel
- 8 bits per pel
- 24 bits per pel

If a device uses a bitcount of 1, 4, or 8 bits per pel, the bitmap bits contain index values for a bitmap color table. If the device supports a bitcount of 1 bit per pel, the color table contains two entries. If the device supports a bitcount of 4 bits per pel, the color table can contain up to 16 entries. And if the device supports a bitcount of 8 bits per pel, the color table can contain up to 256 entries. Figure 38.4 shows a bitmap using a bitcount of 4 bits per pel and an associated color table:

Figure 38.4
A Bitcount and its Associated Color Table



Index	Color
0	White
1	Blue
2	Red
3	Pink
4	Green
5	Cyan
6	Yellow
7	Black
8	Dark gray
9	Pale blue
10	Pale red
11	Pale pink
12	Dark green
13	Dark cyan
14	Brown
15	Pale gray

If the device supports a bitcount of 24 bits per pel, the bitmap bits contain the **bRed**, **bGreen**, and **bBlue** fields of **RGB** structures. There is no color table associated with a bitmap on a device that supports a format of 24 bits per pel (such a device can support over 16 million colors). You can determine the bitcount format a device supports by calling the **GpiQueryDeviceBitmapFormats** function and examining the second value in each pair of values that it returns. The first pair contains the preferred color plane and bitmap format. You should create this color table when you call the **GpiCreateBitmap** function by loading an array with the appropriate number of **RGB** structures and passing the address of this array as the **argbColor** field in the **BITMAPINFO** structure.

38.2.3 Creating Bitmaps

MS OS/2 requires that you associate a presentation space with a memory device context before you perform many of the bitmap operations. A memory device context is a special device context that lets you treat a bitmap in memory like a device. You can copy color information from another bitmap (or copy pels on the display) into a bitmap associated with a memory device context. You create a memory device context with the **DevOpenDC** function by passing as the second argument OD_MEMORY and as the last argument a handle to a compatible device context. For instance, if you are creating a memory device context that is compatible with a screen device context, you would pass the handle from the **WinOpenWindowDC** function as the last argument to **DevOpenDC**. For more information, see the examples in Section 38.3.

When you create a bitmap, you will pass **GpiCreateBitmap** information about the bitmap's color plane and bitcount formats, the dimension of the bitmap in pels, and a description of the bitmap's color table. You will pass this information in two structures: **BITMAPINFO** and **BITMAPINFOHEADER**.

The **BITMAPINFO** structure has the following form:

```
typedef struct _BITMAPINFO { /* bmi                                */
    ULONG  cbFix;            /* length of structure in bytes       */
    USHORT cx;               /* width of bitmapped image in pels   */
    USHORT cy;               /* height of bitmapped image in pels  */
    USHORT cPlanes;          /* format of color plane              */
    USHORT cBitCount;        /* bitcount format                    */
    RGB    argbColor[1];     /* array of RGB structures            */
} BITMAPINFO;
```

The **BITMAPINFOHEADER** structure has the following form:

```
typedef struct _BITMAPINFOHEADER { /* bmp                          */
    ULONG  cbFix;          /* length of structure in bytes         */
    USHORT cx;             /* width of bitmapped image in pels     */
    USHORT cy;             /* height of bitmapped image in pels    */
    USHORT cPlanes;        /* format of color plane                */
    USHORT cBitCount;      /* bitcount format                      */
} BITMAPINFOHEADER;
```

38.2.4 Creating and Loading Custom Bitmaps

You can create a custom bitmap by setting bits in a hard-coded array and passing the array to the **GpiCreateBitmap** function, or by running Icon Editor and loading the bitmap into your application by calling the **GpiLoadBitmap** function. To create a custom hard-coded bitmap, you must perform the following steps:

1. Define in your application's source code an array of bytes that will set pels in an image to the appropriate colors.
2. Set the fields in the **BITMAPINFOHEADER** structure to their appropriate values.
3. Set the fields in the **BITMAPINFO** structure to their appropriate values.
4. Call **GpiCreateBitmap**, pass it the addresses of the structures and the array of bytes you have defined already, and set the flOptions flag to TRUE.

If you want to use this bitmap as a fill pattern, assign it a local identifier by calling the **GpiSetBitmapId** function.

To load a custom bitmap that you created by using Icon Editor, you must perform the following steps:

1. Copy the bitmap file to the directory in which you compile your applications.
2. Create a **BITMAP** entry in your application's resource file, assigning a unique integer identifier to the bitmap.
3. In your application's source code, call the **GpiLoadBitmap** function, passing it the integer identifier that you assigned to the bitmap in the resource file.

You can use **GpiLoadBitmap** to load any bitmap from a file that conforms to the MS OS/2 bitmap file format. This means that you could load a bitmap created by another application, if that application created the correct bitmap header and stored the bitmap bits correctly. For a description of the bitmap file format, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

38.2.5 Drawing Bitmapped Images

You can draw bitmapped images when your application's drawing mode is DM_DRAW, DM_RETAIN, or DM_DRAWANDRETAIN. This means that you can draw bitmapped images on a raster printer or video display, and you can draw them into segments or metafiles associated with a raster device. Most of the bitmap drawing operations occur in your application's device space; however, one drawing operation lets you draw in your application's world space. Table 38.1 describes the functions you can use in each drawing mode:

Table 38.1 Drawing Modes and Bitmapped Output

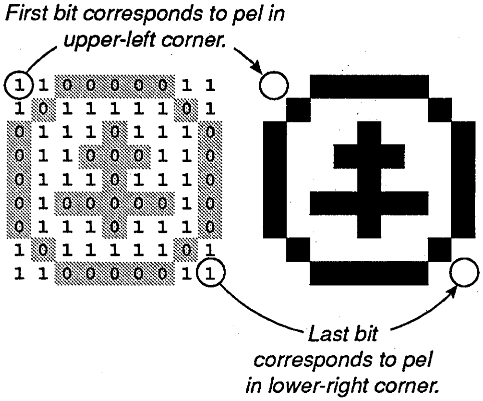
Drawing mode	Function	Output
DM_DRAW	**GpiBitBlt**	Bitmapped image on a raster display or printer.
DM_DRAW	**WinDrawBitmap**	Monochrome bitmapped image on a raster display.
DM_DRAW	**GpiImage**	Special monochrome bitmapped image on a raster display or printer.
DM_DRAW	**GpiWCBitBlt**	Bitmapped image on a raster display or printer; or bitmapped image into a metafile.
DM_RETAIN	**GpiWCBitBlt**	Bitmapped image into a metafile or segment.
DM_DRAWANDRETAIN	**GpiWCBitBlt**	Bitmapped image on a raster display or printer and into an associated metafile or segment.

The **GpiBitBlt** and **GpiWCBitBlt** functions copy a bitmapped image from a rectangle in a source presentation space into a rectangle in a target presentation space. You can use these functions to scale bitmaps (shrink or expand them) by altering the dimensions of the target and source rectangles. **GpiBitBlt** requires that you use device coordinates for the dimensions of source and target rectangles. **GpiWCBitBlt**, however, requires device coordinates for the source rectangle and world coordinates for the target rectangle. You should use **GpiWCBitBlt** to draw a bitmap with consistent dimensions on devices with different aspect ratios. (The aspect ratio is the ratio of a pel's width to its height.)

The **WinDrawBitmap** function draws a bitmapped image by copying it into a window linked to a target presentation space. Unlike **GpiBitBlt** and **GpiWCBitBlt, WinDrawBitmap** does not require you to select a bitmap into a presentation space before you draw the corresponding image. You can use this function to scale bitmaps by specifying DBM_STRETCH as the last argument and the address of a **RECTL** structure as the fourth argument. The coordinates in this structure are always device coordinates.

The **GpiImage** function draws a special bitmapped image. The bitmap bits for a **GpiImage** call are not stored like normal bitmap bits—for example, the first bit in the bitmap contains color information for the pel in the upper-left (rather than the lower-left) corner of the bitmap; the last bit in the bitmap contains color information for the pel in the lower-right (rather than the upper-right) corner of the bitmap. Figure 38.5 shows the correspondence between bitmap bits for the **GpiImage** call and pels in the special bitmapped image:

Figure 38.5
Bits and Pels in a Special Bitmapped Image



```
1 1 0 0 0 0 0 1 1
1 0 1 1 1 1 1 0 1
0 1 1 1 0 1 1 1 0
0 1 1 0 0 0 1 1 0
0 1 1 1 0 1 1 1 0
0 1 0 0 0 0 0 1 0
0 1 1 1 0 1 1 1 0
1 0 1 1 1 1 1 0 1
1 1 0 0 0 0 0 1 1
```

First bit corresponds to pel in upper-left corner.
Last bit corresponds to pel in lower-right corner.

You cannot scale bitmapped images by calling the **GpiImage** function.

You should use the **GpiBitBlt** or **GpiWCBitBlt** function to draw bitmaps that use color formats of 1, 4, 8, or 24 bits per pel. The **GpiImage** and **WinDrawBitmap** functions draw bitmaps using two colors. These two functions do not use a color table; instead, they use the foreground and background colors from the **IMAGEBUNDLE** structure. You can set these colors (and their corresponding mix modes) by calling the **GpiSetAttrs** function, or you can determine the **IMAGEBUNDLE** colors and mix modes by calling the **GpiQueryAttrs** function.

You can draw inverted bitmaps by calling the **GpiBitBlt** or **GpiWCBitBlt** function, passing it ROP_NOTSRCCOPY as the raster operation. You can also draw inverted bitmaps by calling **WinDrawBitmap**, passing it DBM_INVERT as the last argument. You can draw halftone bitmaps by calling **WinDrawBitmap**, passing it DBM_HALFTONE as the last argument.

38.2.6 Copying Images from a Display into a Bitmap

You can copy an image from a raster video display into a bitmap by calling the **GpiBitBlt** or **GpiWCBitBlt** function. Before doing so, however, you must create a memory device context by calling the **DevOpenDC** function. This device context lets you treat a bitmap in memory like a device, by copying color information from pels on the display into the bitmap.

Once you create a memory device context, associate it with a presentation space, and select your bitmap into the presentation space, you can use the presentation-space handle as the first argument to the **GpiBitBlt** or **GpiWCBitBlt** function. If you will be drawing the image (saved in the memory device context) on devices with different aspect ratios, you should use the **GpiWCBitBlt** function to preserve the original dimensions of the bitmap. For more information about saving bitmapped images, see Section 38.3.

38.2.7 Saving Bitmaps in a File

You can save a bitmap in a file on a disk by calling the **GpiQueryBitmapBits**, **DosOpen**, **DosWrite**, and **DosClose** functions. The **GpiQueryBitmapBits** function copies bitmap bits into a buffer. After you create a file by calling **DosOpen**, you can call **DosWrite** to copy the buffer containing the bitmap bits into the file. After copying the bits into the file, you close it by calling **DosClose**.

If you have stored a bitmap on disk and need to use it again in your application, you can copy the file's contents into a buffer by calling the **DosRead** function and then set the bitmap bits by calling the **GpiSetBitmapBits** function. (You must associate the bitmap with a memory device context before trying to set the bits.)

If your application creates bitmaps another application might use, it should create them by using the standard MS OS/2 bitmap file format. For a complete description of the file format, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

38.3 Using Bitmaps

You can use bitmap functions to perform the following tasks:

- Copy an image from a video display into a bitmap.
- Scale bitmapped images.
- Create custom fill patterns for area primitives and paths.
- Load a bitmap created by Icon Editor.
- Draw bitmapped images.
- Store bitmaps in a metafile or segment.

38.3.1 Copying an Image from a Video Display to a Bitmap

To copy an image from a video display to a bitmap, you must perform the following steps:

1. Associate the memory device context with a presentation space.
2. Create a bitmap in a memory device context.
3. Select the bitmap into the memory device context by calling the **GpiSetBitmap** function.
4. Determine the location (in device coordinates) of the image.
5. Call the **GpiBitBlt** function and copy the image to the bitmap.

The following code fragment demonstrates these steps:

```
PSZ pszData[4] = { "Display", NULL, NULL, NULL };
HAB hab;
HPS hpsMem, hps;
HDC hdcMem, hdc;
SIZEL sizlPage;
BITMAPINFOHEADER bmp;
BYTE abBuffer[80];
PBITMAPINFO pbmi;
HBITMAP hbm;
SHORT sWidth, sHeight;
POINTL aptl[6];
LONG alData[2];

/*
 * Create the memory device context and presentation space so that they
 * are compatible with the screen device context and presentation space.
 */

hdcMem = DevOpenDC(hab, OD_MEMORY, "*", 4L,
    (PDEVOPENDATA) pszData, hdc);
hpsMem = GpiCreatePS(hab, hdcMem, &sizlPage,
    PU_PELS | GPIA_ASSOC | GPIT_MICRO);

/* Determine the device's plane/bitcount format. */

GpiQueryDeviceBitmapFormats(hpsMem, 2L, alData);

/*
 * Load the BITMAPINFOHEADER and BITMAPINFO structures. The sWidth and
 * sHeight fields specify the width and height of the destination
 * rectangle.
 */

bmp.cbFix = (ULONG) sizeof(bmp);
bmp.cx = sWidth;
bmp.cy = sHeight;
bmp.cPlanes = alData[0];
bmp.cBitCount = alData[1];

pbmi = (PBITMAPINFO) abBuffer;
pbmi->cbFix = bmp.cbFix;
pbmi->cx = bmp.cx;
pbmi->cy = bmp.cy;
pbmi->cPlanes = bmp.cPlanes;
pbmi->cBitCount = bmp.cBitCount;

/* Create a bitmap that is compatible with the display. */

hbm = GpiCreateBitmap(hpsMem, &bmp, FALSE, NULL, pbmi);

/* Associate the bitmap and the memory presentation space. */

GpiSetBitmap(hpsMem, hbm);

/* Copy the screen to the bitmap. */

aptl[0].x = 0;       /* lower-left corner of destination rectangle  */
aptl[0].y = 0;       /* lower-left corner of destination rectangle  */
aptl[1].x = sWidth;  /* upper-right corner of destination rectangle */
aptl[1].y = sHeight; /* upper-right corner of destination rectangle */
aptl[2].x = 0;       /* lower-left corner of source rectangle       */
aptl[2].y = 0;       /* lower-left corner of source rectangle       */
GpiBitBlt(hpsMem,
    hps,
    3L,              /* number of points in aptlPoints              */
    aptl,
    ROP_SRCCOPY,
    BBO_IGNORE);
```

38.3.2 Scaling and Drawing a Bitmapped Image

You can scale a bitmap by calling the **GpiBitBlt** and **GpiWCBitBlt** functions and altering the dimensions of the target rectangle. The following code fragment shows how to shrink the screen copied in the first example to half its original size and redraw it by calling **GpiBitBlt**:

```
/* target-rectangle dimensions (in device coordinates) */

aptl[0].x = 0;
aptl[0].y = 0;
aptl[1].x = sWidth/2;
aptl[1].y = sHeight/2;

/* source-rectangle dimensions (in device coordinates) */

aptl[2].x = 0;
aptl[2].y = 0;
aptl[3].x = sWidth;
aptl[3].y = sHeight;
GpiBitBlt(hps, hpsMem, 4L, aptl, ROP_SRCCOPY, BBO_IGNORE);
```

38.3.3 Creating a Custom Fill Pattern

You can create a custom fill pattern that MS OS/2 will use to fill area primitives and paths. To create this pattern, you perform the following steps:

1. Set an array of bits for a bitmap that measures 8 bits by 8 bits (remember that MS OS/2 pads the bitmap bits on a **ULONG** [32-bit] boundary).
2. Create a bitmap in a screen presentation space by calling the **GpiCreateBitmap** function, passing it the address of the array of bits from Step 1.
3. Assign a local identifier (*lcid*) to the bitmap by calling the **GpiSetBitmapId** function.
4. Set the attribute of the pattern set in the **AREABUNDLE** structure by calling the **GpiSetPattern** function.

The following code fragment shows how to create the pattern:

```
/*
 * Define an array of bytes--this array creates
 * a cross-hatch pattern.
 */

BYTE abPattern5[] = {
    0xFF, 0xFF,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00,
    0x80, 0x00 };
```

```
LONG lcidCustom;
HPS hps;
BITMAPINFOHEADER bmp;
PBITMAPINFO pbmi;
HBITMAP hbm;

/* Create the bitmap, passing the address of the array of bytes. */

hbm = GpiCreateBitmap(hps, &bmp, CBM_INIT, (PBYTE) abPattern5,
    pbmi);

/* Assign a local identifier to the bitmap. */

GpiSetBitmapId(hps, hbm, lcidCustom);

/* Set the pattern-set attribute in the AREABUNDLE structure. */

GpiSetPatternSet(hps, lcidCustom);
```

38.3.4 Loading a Bitmap from a File

You can load a bitmap from a file if the format of the file corresponds to the standard MS OS/2 bitmap file format. (Any bitmap that you create using Icon Editor is automatically stored in this format.) To load a bitmap, you perform the following steps:

1. Copy the bitmap file into the directory that contains your application's resource file and source code.
2. Create an entry in your application's resource file, assigning a unique integer identifier to the bitmap.
3. Call the **GpiLoadBitmap** function in your application's source code, passing it the integer identifier that you assigned to the bitmap in your application's resource file.

The following code fragment, from an application's resource file, assigns the integer value 200 to a bitmap file called *custom.bmp*:

```
BITMAP  200 custom.bmp
```

The following code fragment, from the application, shows how to retrieve a bitmap handle by calling **GpiLoadBitmap**:

```
hbm = GpiLoadBitmap(hpsMem, /* presentation-space handle         */
    NULL,                   /* points to dynamic-link library    */
    200,                    /* bitmap identifier in resource file */
    99L,                    /* bitmap width                      */
    99L);                   /* bitmap height                     */
```

38.3.5 Storing a Bitmap in a Metafile

You can draw bitmaps in a metafile or segment by calling the **GpiWCBitBlt** function. MS OS/2 converts this function to a drawing order. The target-rectangle dimensions that you pass to **GpiWCBitBlt** are in world coordinates, not device coordinates. The following code fragment shows how to draw a bitmap in a metafile and then play the metafile:

```
/*
 * Scale a custom bitmap created with Icon Editor so that it fits
 * the entire screen, and draw it into a metafile.
 */

aptl[0].x = 0;
aptl[0].y = 0;
aptl[1].x = sWidth;              /* screen width           */
aptl[1].y = sHeight;             /* screen height          */
aptl[2].x = 0;
aptl[2].y = 0;
aptl[3].x = 99;                  /* width of custom bitmap  */
aptl[3].y = 99;                  /* height of custom bitmap */

GpiWCBitBlt(hpsMeta, hbm, 4L, aptl, ROP_SRCCOPY, BBO_IGNORE);
GpiAssociate(hpsMeta, NULL);
hmf = DevCloseDC(hdcMeta);

/* Set the options in the GpiPlayMetaFile array. */

alOptions[0] = 0L;               /* reserved                */
alOptions[1] = LT_DEFAULT;       /* uses default transforms */
alOptions[2] = 0L;               /* reserved                */
alOptions[3] = LC_DEFAULT;       /* uses default            */
alOptions[4] = RES_DEFAULT;      /* uses default            */
alOptions[5] = SUP_DEFAULT;      /* uses default            */
alOptions[6] = CTAB_DEFAULT;     /* uses default            */
alOptions[7] = CREA_DEFAULT;     /* uses default            */

/* Play the metafile. */

GpiPlayMetaFile(hps, hmf, 8L, alOptions, 0L, 0L, NULL);
```

38.4 Summary

The following list summarizes the MS OS/2 bitmap functions:

GpiBitBlt Performs a bit-block transfer, copying bitmap bits from a source presentation space to a target presentation space. You can associate the source and target presentation spaces with memory device contexts, raster-display device contexts, or raster-printer device contexts (prior to calling the function). You can use **GpiBitBlt** to set bits in a bitmap to correspond with an image on the screen, or to create an image on a screen (or a page of printer paper) that corresponds to bits in a bitmap. Table 38.2 shows the effect of associating the source and target presentation spaces with different device contexts:

Table 38.2 GpiBitBlt Output

Source presentation space	Target presentation space	Result
Associated with a memory device context.	Associated with a memory device context.	MS OS/2 copies bitmap bits from a source device context to a target device context.
Associated with a memory device context.	Associated with a screen or printer device context.	MS OS/2 sets pels on a screen or printer, using bitmap bits from a memory device context.
Associated with a screen device context.	Associated with a memory device context.	MS OS/2 sets bitmap bits in a memory device context, using pels on the screen.

You can also scale an image with this function by specifying different dimensions for the source and target rectangles.

GpiCreateBitmap Creates a bitmap that is compatible with a device associated with a presentation space. Before calling **GpiCreateBitmap**, you call the **GpiQueryDeviceBitmapFormats** function and determine the color format of the device. You will use this information to fill the fields in the **BITMAPINFO** and **BITMAPINFOHEADER** structures that you pass as arguments to **GpiCreateBitmap**. You can also specify zero planes with a bitcount of zero and MS OS/2 will set the appropriate values. You can set the bitmap bits when you call the function, or you can set them later when you run your application.

GpiDeleteBitmap Decrements a bitmap's use count. If the count reaches zero, MS OS/2 deletes the bitmap. You should delete a bitmap whenever your application no longer needs it, in order to free system resources.

GpiImage Draws a special bitmap called an image, which is a nonstandard monochrome (two-color) bitmap. A data structure (IMAGEBUNDLE) contains fields that correspond to the image foreground and background colors and mix modes. The foreground color of the image specifies the color of pels that are set; the background color of the image specifies the color of pels that are not set.

GpiLoadBitmap Loads a bitmap from a file that uses the standard MS OS/2 bitmap-file format. You can load files that you created by using Icon Editor or you can load files that your application creates—if it stores the files in the correct file format.

GpiQueryBitmapBits Copies an array of bitmap bits into a buffer, which you can then save in a file on disk.

GpiQueryBitmapDimension Retrieves the width and height (in 0.1-mm units) of a bitmap.

GpiQueryBitmapHandle Retrieves the handle of a bitmap with a specific local identifier (*lcid*) value. A bitmap is tagged with a local identifier when an application uses it as a fill pattern for area primitives and paths.

GpiQueryBitmapParameters Fills a copy of the **BITMAPINFOHEADER** structure with information about a particular bitmap. This information includes the bitmap's width, height, color-plane count, and bits per pel.

GpiQueryDeviceBitmapFormats Retrieves the color format that you should use for bitmaps generated on a specific device. MS OS/2 fills an array that you pass as a function argument with the various color formats that the device supports. The array contains pairs of color-plane and bitcount values. The first pair in the array contains the values most suited to the device.

GpiQueryPel Retrieves an index value from your application's logical color table for a pel at a specified position (in world coordinates).

GpiSetBitmap Associates a bitmap with a memory device context. If your application had previously associated a bitmap with the memory device context, **GpiSetBitmap** disassociates the original bitmap from the device context and returns the handle that identifies the old bitmap.

GpiSetBitmapBits Uses information stored in a buffer to set rows of bitmap bits. You should use **GpiSetBitmapBits** to restore a bitmap that you have saved on disk in a file.

GpiSetBitmapDimension Sets the width and height (in 0.1-mm units) of a bitmap.

GpiSetBitmapId Associates a local identifier (*lcid*) with a bitmap. You should use this function to identify a bitmap that your application will use as a fill pattern for area primitives and paths.

GpiSetPel Sets a pel at a location (in world coordinates), using a color from the logical color table.

Chapter
39

Regions

39.1 Introduction 529

39.2 About Regions 529
39.2.1 Clip Regions 530
39.2.2 Combining Regions 531
39.2.3 Painting Regions 532

39.3 Using Regions 533
39.3.1 Creating a Region 533
39.3.2 Combining Regions 534
39.3.3 Destroying a Region 534
39.3.4 Comparing Regions 534
39.3.5 Offsetting a Region 535
39.3.6 Painting a Region 535
39.3.7 Locating a Point with Respect to a Region 535
39.3.8 Determining Coordinates of Rectangles in a Region ... 536

39.4 Summary 537

39.1 Introduction

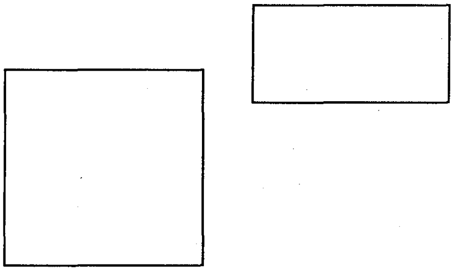
This chapter describes a graphics object called a region. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Coordinate spaces and transformations
- Color and mix modes
- Fill patterns (areas)

39.2 About Regions

A region is a rectangle, multiple disjoint rectangles, or a polygon formed by multiple intersecting rectangles, in your application's device space. If a region consists of intersecting rectangles, the intersecting sides are always perpendicular. Since regions are always created and drawn in your application's device space, region coordinates (the coordinates that define the location and dimensions of a region) are always device coordinates. Figure 39.1 shows a region that consists of two disjoint rectangles:

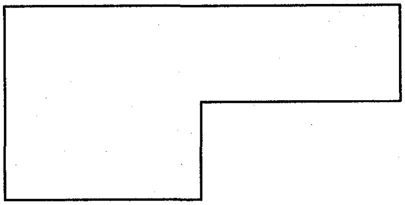
Figure 39.1
Disjoint Region



An application defined the region by passing an array containing the coordinates for the two rectangles to the **GpiCreateRegion** function. The application drew the region by calling the **GpiPaintRegion** function.

Figure 39.2 shows a region that consists of two intersecting rectangles:

Figure 39.2
Region of Two Intersecting Rectangles



This region was also defined by calling the **GpiCreateRegion** and **GpiPaintRegion** functions.

Most of the region creation, detection, offsetting, and filling functions use a special **RECTL** structure to define the device-space coordinates of rectangles that correspond to regions. The **RECTL** structure has the following form:

```
typedef struct _RECTL {        /* rcl                              */
    LONG  xLeft;               /* x-coordinate for lower-left corner  */
    LONG  yBottom;             /* y-coordinate for lower-left corner  */
    LONG  xRight;              /* x-coordinate for upper-right corner */
    LONG  yTop;                /* y-coordinate for upper-right corner */
} RECTL;
```

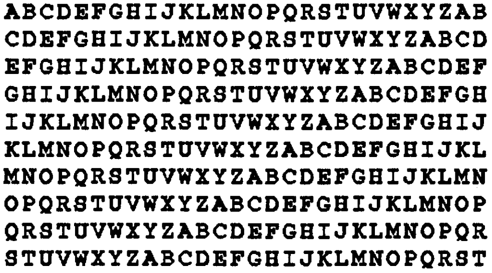
When you create a rectangle in a device space and pass its coordinates to a region function, MS OS/2 excludes the top and rightmost edges. This means that you will need to add 1 to the values in the **xRight** and **yTop** fields in order to obtain the desired dimensions. For example, if your application requires a region that measures 100-by-100 device units with a lower-left corner at (10,10), you would need to set **xRight** and **yTop** to 111 instead of 110.

You can use regions to clip output in your application's device space, to draw a single filled rectangle, or to draw a special filled polygon consisting of two intersecting rectangles. You can also use regions to repaint the background in the client area of your application's window or to repaint part of an object in a picture.

39.2.1 Clip Regions

A clip region is a special region that MS OS/2 uses to clip output in your application's device space. Figures 39.3 and 39.4 show the effect of setting a clip region. In Figure 39.3, there is no clip region and text is drawn in the application's client area:

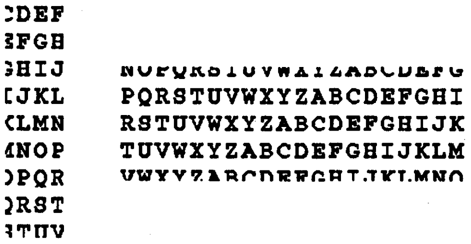
Figure 39.3
Text in Client Area



```
ABCDEFGHIJKLMNOPQRSTUVWXYZAB
CDEFGHIJKLMNOPQRSTUVWXYZABCD
EFGHIJKLMNOPQRSTUVWXYZABCDEF
GHIJKLMNOPQRSTUVWXYZABCDEFGH
IJKLMNOPQRSTUVWXYZABCDEFGHIJ
KLMNOPQRSTUVWXYZABCDEFGHIJKL
MNOPQRSTUVWXYZABCDEFGHIJKLMN
OPQRSTUVWXYZABCDEFGHIJKLMNOP
QRSTUVWXYZABCDEFGHIJKLMNOPQR
STUVWXYZABCDEFGHIJKLMNOPQRST
```

In Figure 39.4, a clip region consisting of two disjoint rectangles is set and the text from Figure 39.3 is clipped:

Figure 39.4
Disjoint Clip Region



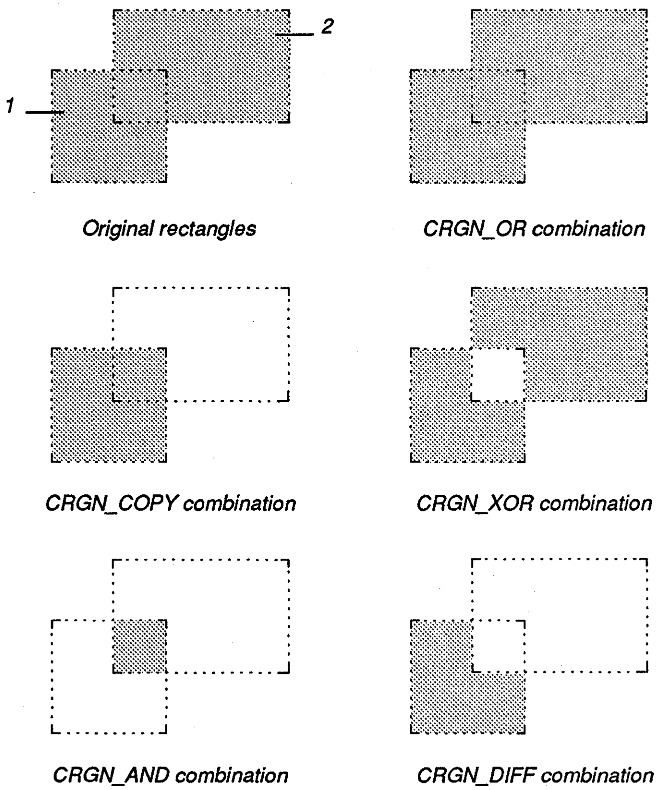
```
CDEF
EFGH
GHIJ     NOPQRSTUVWXYZABCDEFG
IJKL     PQRSTUVWXYZABCDEFGHI
KLMN     RSTUVWXYZABCDEFGHIJK
MNOP     TUVWXYZABCDEFGHIJKLM
OPQR     VWXYZABCDEFGHIJKLMNO
QRST
STUV
```

The application created the region from two disjoint rectangles by calling the **GpiCreateRegion** function. It then sets the clip region by calling the **GpiSetClipRegion** function. For more information about using regions for clipping, see Chapter 40, "Clipping."

39.2.2 Combining Regions

You can use the **GpiCombineRegion** function to combine two rectangles in one of five ways to form a region. Figure 39.5 shows two rectangles, labeled 1 and 2, before they are combined by **GpiCombineRegion** and then after they are combined, using each of the five combining methods:

Figure 39.5
Combining Regions



Original rectangles

CRGN_OR combination

CRGN_COPY combination

CRGN_XOR combination

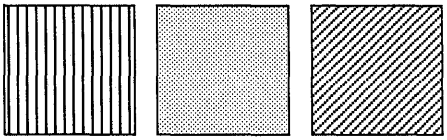
CRGN_AND combination

CRGN_DIFF combination

39.2.3 Painting Regions

You can use regions to draw a single filled rectangle or a filled polygon (that consists of multiple intersecting rectangles) by calling the **GpiPaintRegion** function after you create a region. This function uses the current fill pattern to fill the interior of the region. The default fill pattern is solid black. If this pattern is not appropriate for your application, you can select one of fifteen other predefined patterns, or you can create a custom pattern by using a bitmap. Figure 39.6 shows a region drawn three times, each time filled with a different predefined pattern:

Figure 39.6
Three Square Regions and Three Fill Patterns



MS OS/2 stores the fill pattern and the fill-pattern colors in the **AREABUNDLE** structure. You can determine which predefined pattern is currently selected by calling the **GpiQueryPattern** or **GpiQueryAttrs** function. You can select a different predefined pattern by calling the **GpiSetPattern** or **GpiSetAttrs** function. For more information about selecting a new pattern or creating a custom pattern, see Chapter 36, "Area Primitives."

39.3 Using Regions

You can use region functions to perform the following tasks:

- Create a region.
- Combine regions.
- Destroy a region.
- Compare regions.
- Move a region.
- Fill a region.
- Locate a point with respect to a region.
- Determine coordinates of region rectangles.
- Locate a rectangle with respect to another region.

39.3.1 Creating a Region

To create a region, you must perform the following steps:

1. Create an array of **RECTL** structures containing the dimensions of the rectangles that will compose the region.
2. Call the **GpiCreateRegion** function to create the region (this function returns a handle that identifies the region).

The following code fragment shows how to create a region:

```
HPS hps;                     /* presentation-space handle */
HRGN hrgn;                   /* region handle             */
RECTL rcl[] = { 25, 50,      /* rectangle 1               */
    75, 100,
    50, 75,                  /* rectangle 2               */
    100, 150,
    75, 125,                 /* rectangle 3               */
    200, 175,
    150, 75,                 /* rectangle 4               */
    250, 150};
    .
    .
    .
hrgn = GpiCreateRegion(hps,  /* creates region                   */
    4L,                      /* number of rectangles in region */
    rcl);                    /* array of rectangle structures   */
```

39.3.2 Combining Regions

To combine two regions, you must perform the following steps:

1 Create a region that MS OS/2 can use as the final destination region (one that contains the two combined regions).

2 Determine which of the five available combining methods is best suited to your application.

3 Call the **GpiCombineRegion** function.

The following code fragment shows how to combine two regions by using the OR operation:

```
HPS hps;                   /* presentation-space handle            */
HRGN hrgn1, hrgn2, hrgn3;  /* region handles                       */
RECTL rcl1[] = { 50, 100,  /* rectangle forming first region       */
    200, 175};

RECTL rcl2[] = { 125, 150, /* rectangle forming second region      */
    225, 200 };

RECTL rcl3[] = { 0, 0,     /* rectangle forming destination region */
    0, 0 };
    .
    .
    .
hrgn = GpiCreateRegion(hps, 1L,  /* creates first region            */
    rcl1);
hrgn2 = GpiCreateRegion(hps, 1L, /* creates second region           */
    rcl2);
hrgn3 = GpiCreateRegion(hps, 1L, /* creates destination region      */
    rcl3);
GpiCombineRegion(hps,            /* creates union of hrgn and hrgn2 */
    hrgn3, hrgn1, hrgn2, CRGN_OR);
```

39.3.3 Destroying a Region

To destroy a region, you must call the **GpiDestroyRegion** function and pass it a handle that identifies the region your application is no longer using, as shown in the following code fragment:

```
HPS hps;       /* presentation-space handle */
HRGN hrgn;     /* region handle             */

GpiDestroyRegion(hps, hrgn);   /* destroys region identified by hrgn */
```

39.3.4 Comparing Regions

You can determine whether two regions are defined by the same rectangle by calling the **GpiEqualRegion** function, as shown in the following code fragment:

```
HPS hps;         /* presentation-space handle       */
HRGN hrgn1;      /* handle of first region          */
HRGN hrgn2;      /* handle of second region         */
LONG lEqual;     /* return value for GpiEqualRegion */

lEqual = GpiEqualRegion(hps,
    hrgn1, hrgn2);              /* compares regions 1 and 2 */
if (lEqual == EQRGN_EQUAL) {
    .
    . /* regions are equal */
    .
}
```

```
if (lEqual == EQRGN_NOTEQUAL) {
     .
     . /* regions are not equal */
     .
}
if (lEqual == EQRGN_ERROR) {
     .
     . /* error occurred */
     .
}
```

39.3.5 Offsetting a Region

The **GpiOffsetRegion** function moves a region by a specified offset in world space. When you call this function, you pass the address of a **POINTL** structure that contains an *x*- and a *y*-translation factor. The following code fragment shows how to offset a region:

```
HRGN hrgn;          /* region handle               */
HPS hps;            /* presentation-space handle   */
POINTL ptlNewPos;   /* structure for offset value  */

/*
 * Set the offset here, storing the x- and
 * y-components in ptlNewPos.
 */

GpiOffsetRegion(hps, hrgn, &ptlNewPos);    /* offsets region */
```

39.3.6 Painting a Region

The **GpiPaintRegion** function fills a region with the current fill pattern, using the colors and mix modes that appear in the current **AREABUNDLE** structure. The following code fragment shows how to change the fill-pattern color to green and then paint the region:

```
HRGN hrgn;       /* region handle             */
HPS hps;         /* presentation-space handle */

GpiSetColor(hps, CLR_GREEN);
GpiPaintRegion(hps, hrgn); /* paints region at new location */
```

39.3.7 Locating a Point with Respect to a Region

The **GpiPtInRegion** function determines whether a point lies within the borders of a region. This function is especially useful in applications that must determine whether the mouse pointer lies over a region.

You must perform the following steps to determine the location of the mouse pointer with respect to a region—for example, when the user presses the leftmost mouse button:

1 Retrieve the mouse-pointer coordinates and store them in a **POINTL** structure.

2 Call the **GpiPtInRegion** function, passing it a handle that identifies the appropriate region and the address of the **POINTL** structure from Step 1.

3 Examine the value that **GpiPtInRegion** returns in order to determine whether the point lies within the region.

The following code fragment shows how to locate the mouse pointer with respect to the region:

```
HPS hps;           /* presentation-space handle      */
POINTL ptlCurPos;  /* point structure                */
HRGN hrgn;         /* region handle                  */
LONG lPosition;    /* return value for GpiPtInRegion */

/* Determine the mouse coordinates and store them in ptlCurPos. */

lPosition = GpiPtInRegion(hps, hrgn, &ptlCurPos);

    if (ptlCurPos == PRGN_INSIDE) {
        .
        . /* point lies within region */
        .
    }

    if (ptlCurPos == PRGN_OUTSIDE) {
        .
        . /* point lies outside region */
        .
    }
```

39.3.8 Determining Coordinates of Rectangles in a Region

If a region consists of more than one rectangle, you can call the **GpiQueryRegionRects** function to retrieve the coordinates of the lower-left and upper-right corners of each rectangle.

To determine the coordinates for the rectangles that form a region, you must perform the following steps:

1 Create a copy of the **RGNRECT** structure that MS OS/2 uses when it retrieves the rectangle coordinates.

2 Create an array of **RECTL** structures that MS OS/2 can load with rectangle coordinates.

3 Determine the approximate number of rectangles that compose the region and set the **crc** field in the **RGNRECT** structure to this number.

4 Determine the direction in which MS OS/2 traverses the region to retrieve the rectangle coordinates and set the **usDirection** field in the **RGNRECT** structure to the associated value.

5 Call the **GpiQueryRegionRects** function to retrieve the coordinates.

The following code fragment shows how to determine the coordinates of the rectangles in a region:

```
HPS hps;         /* presentation-space handle      */
HRGN hrgn;       /* region handle                  */
RGNRECT rgnrc;   /* structure for region rectangles */
RECTL rcl[7];    /* array of RECTL structures      */

rgnrc.crc = 7;                           /* rectangles to query    */
rgnrc.usDirection = RECTDIR_RTLF_BOTTOP; /* right/left, bottom/top */
GpiQueryRegionRects(hps, hrgn,
    OL,                       /* returns all rectangle coordinates */
    &rgnrc, rcl);
```

39.4 Summary

The following list summarizes the MS OS/2 region functions:

GpiCombineRegion Combines two regions in one of five ways. When you call this function, you specify one of five constants that identify methods of combining the regions.

GpiCreateRegion Creates a region from one or more rectangles. If your application creates the region from multiple rectangles, MS OS/2 uses an OR operation to combine the multiple rectangles.

GpiDestroyRegion Destroys a region and frees the memory associated with it.

GpiEqualRegion Determines whether your application created two regions from the same rectangle(s).

GpiOffsetRegion Moves a region by a specified offset. MS OS/2 issues an error if the specified region is a clip region.

GpiPaintRegion Fills the interior of a region. MS OS/2 fills the interior with the current fill pattern, using the colors and mix modes in the attribute bundle. MS OS/2 issues an error if the specified region is a clip region.

GpiPtInRegion Determines whether a point lies within a region. MS OS/2 issues an error if the specified region is a clip region.

GpiQueryRegionBox Retrieves the dimensions of the smallest rectangle that surrounds the entire region.

GpiQueryRegionRects Retrieves the coordinates of individual rectangles that compose a region, filling an array of **RECTL** structures with these coordinates. If the number of rectangles that compose a region is unknown, you can call this function several times, refilling the array each time with the new rectangle coordinates.

GpiRectInRegion Determines whether all or part of a rectangle lies within a region.

GpiSetRegion Creates a region from one or more rectangles. MS OS/2 does not include points on the top and rightmost edges of the rectangles in the region.

Chapter

40

Clipping

40.1 Introduction 541

40.2 About Clipping 541

40.3 Using Clipping 543

40.3.1 Excluding a Rectangular Area from a Clip Region 544

40.3.2 Adding a Rectangular Area to a Clip Region 544

40.3.3 Determining the Size of a Clipping Area 545

40.3.4 Setting a Clip Region 545

40.3.5 Setting a Clip Path 546

40.4 Summary 547

40.1 Introduction

This chapter describes a graphics process called clipping. You should also be familiar with the following topics:

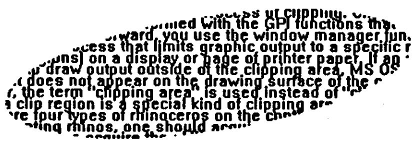
- Presentation spaces and device contexts
- Coordinate spaces and transformations
- Regions
- Paths

40.2 About Clipping

Clipping is a process that limits graphic output to a certain region (or to certain multiple regions) on a display or a page of printer paper. If an application attempts to draw output outside of the clipping area, MS OS/2 "clips" the output so that it does not appear on the drawing surface of the output device. (In this chapter, the term "clipping area" is used instead of "clip region," because in MS OS/2, a clip region is a special kind of clipping area.)

In Figure 40.1, an application defined an elliptical clipping area before drawing text output:

Figure 40.1
Text in an Elliptical Clipping Area



Clipping areas can be polygons (closed regions with straight sides), closed regions with curved sides, or closed regions with straight *and* curved sides.

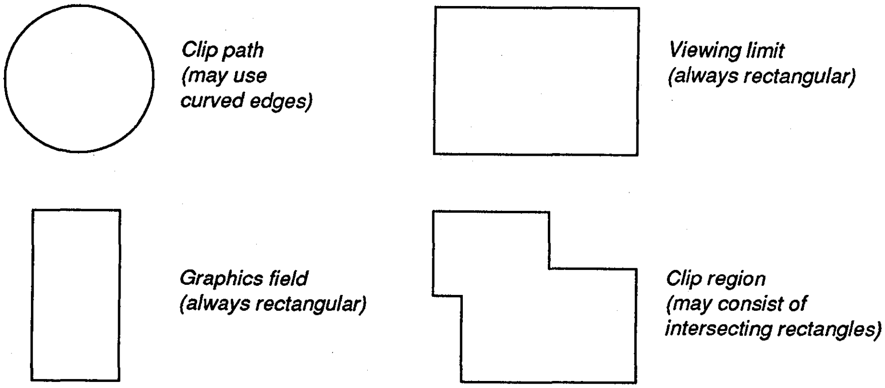
An application can define a clipping area in world space, model space, page space, or device space. (For more information about coordinate spaces, see Chapter 31, "Coordinate Spaces and Transformations.") There are four kinds of clipping areas in MS OS/2, each associated with a particular coordinate space. The following list indicates the coordinate space associated with each clipping area:

Clipping area	Associated coordinate space
Clip path	World space
Viewing limit	Model space
Graphics field	Page space
Clip region	Device space

If the clip path is a rectangle, it is drawn inclusive/inclusive, which means that MS OS/2 includes the bottom and leftmost edges of the rectangle in the clip path as well as the top and rightmost edges. The viewing limit and the graphics field are also inclusive/inclusive. The clip region is inclusive/exclusive, which means that MS OS/2 includes the bottom and leftmost edges but excludes the top and rightmost edges of the rectangle from the clip region.

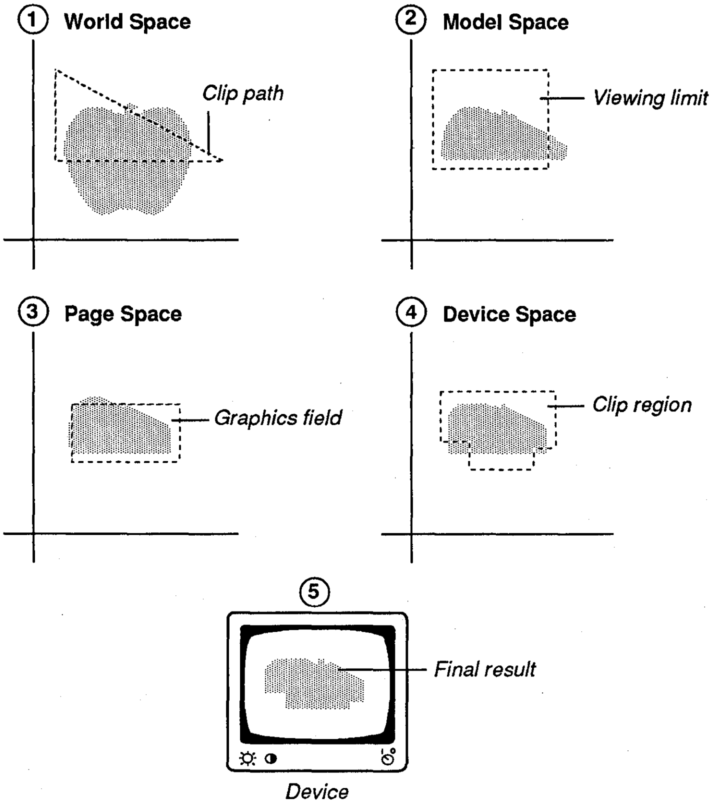
An application can use a clip path and a viewing limit for retained-drawing operations. The clip path is the only clipping area that can have curved sides, while the viewing limit and the graphics field are always rectangular. The clip region can consist of a single rectangle or any number of intersecting rectangles. Figure 40.2 shows valid shapes for each type of clipping area:

Figure 40.2
Valid Clipping Areas


Clip path (may use curved edges)
Viewing limit (always rectangular)
Graphics field (always rectangular)
Clip region (may consist of intersecting rectangles)

When an application defines clipping areas in several of the coordinate spaces, the final result is similar to combining all of the areas into a single area. This single area is defined by the intersection of the areas in each coordinate space. Figure 40.3 shows the effect of drawing a shape when a clipping area is set in each of the coordinate spaces:

Figure 40.3
Clipping in Four Coordinate Spaces



Before you attempt to transform a clipping area in the world, model, or page space, remember that the clip path is the only clipping area that your application can rotate by using one of the rotation transformations. If you try to rotate the viewing limit or the graphics field, the result will be a larger rectangle.

40.3 Using Clipping

You can use clipping functions to perform the following tasks:

- Exclude a rectangular area from a clip region.
- Intersect a rectangular area with a clip region.
- Determine the size of the smallest rectangle that will completely surround the intersection of the current clip areas.
- Determine the size of the current clip region, graphics field, or viewing limit.
- Set a clip path, clip region, graphics field, or viewing limit.

40.3.1 Excluding a Rectangular Area from a Clip Region

Some applications let you prepare output in multiple stages (for example, a word-processor application may allow you to prepare your text first and then add bitmaps that enhance and support the text). These applications can use the **GpiExcludeClipRectangle** function to exclude an area from a clip region, preventing the user from destroying output that already exists. To exclude an area from a clip region, you must perform the following steps:

1 Determine the dimensions (in device coordinates) of the smallest rectangle that completely surrounds the area containing original output (this is the area to exclude from the clip region).

2 Call the **GpiExcludeClipRectangle** function and pass it the dimensions of the area that contains the original output.

The following code fragment illustrates these steps:

```
HPS hps;         /* presentation-space handle      */
RECTL rcl;       /* rectangle structure            */
    .
    .            /* Set rectangle coordinates here. */
    .
GpiExcludeClipRectangle(hps, &rcl);
```

40.3.2 Adding a Rectangular Area to a Clip Region

Some applications may need to increase the size of a clip region. For example, a user might request that a desktop-publishing application extend a column of text on a page. To do this, the application can call the **GpiIntersectClipRectangle** function to combine a rectangular area with a clip region.

To intersect the rectangle with a clip region, you perform the following steps:

1 Determine the dimensions (in device coordinates) of the rectangular area to intersect with the clip region.

2 Call the **GpiCreateRegion** function and pass it the dimensions of the rectangle that you obtained in Step 1.

3 Call **GpiCreateRegion** again and create a third region that will become the final destination region.

4 Call the **GpiCombineRegion** function and combine the original region with the new one to form a final region.

5 Call the **GpiCreateClipRegion** function and pass it the handle returned by **GpiCombineRegion**.

The following code fragment illustrates these steps:

```
HPS hps;       /* presentation-space handle */
RECTL rcl;     /* rectangle structure       */
     .
     .
     .
/* Create the first clipping region. */

hrgn = GpiCreateRegion(hps, 1L, &rcl); /* creates rgn1 */
GpiSetClipRegion(hps, hrgn, NULL);
     .
     .
     .
/* Compute coordinates of second region here. */
     .
     .
     .
/*
 * Create second and third regions and generate a
 * new clipping region, identified by hrgn3.
 */

hrgn2 = GpiCreateRegion(hps, 1L, &rcl); /* creates rgn2 */
hrgn3 = GpiCreateRegion(hps, 1L, &rcl); /* creates rgn3 */
GpiCombineRegion(hps, hrgn3, hrgn, hrgn2, CRGN_OR);
GpiSetClipRegion(hps, hrgn3, NULL);
```

40.3.3 Determining the Size of a Clipping Area

If a computer-aided-design (CAD) application is able to set a clip path in world space, a viewing limit in model space, and a graphics field in page space, it may be necessary for you to determine the size of the clipping area formed by the intersection of the three. The **GpiQueryClipBox** function returns the dimensions (in world coordinates) of the smallest rectangle that completely surrounds the intersection of the defined clipping areas.

The following code fragment shows how to use **GpiQueryClipBox** to fill a rectangle structure with the desired coordinates:

```
HPS hps;         /* presentation-space handle */
RECTL rcl;       /* rectangle structure       */

GpiQueryClipBox(hps, &rcl);
```

40.3.4 Setting a Clip Region

Clip regions are useful when an application must clip to an area shaped like a rectangle or like a number of intersecting rectangles. To create a clip region, you must perform the following tasks:

1 Determine the shape and dimensions of the clip region.

2 Load the coordinates for the rectangle(s) that define the clip region into an array of rectangle structures.

3 Create the clip region by calling the **GpiCreateRegion** function.

4 Set the region to a clip region with **GpiSetClipRegion**.

The following code fragment shows how to create a clip region:

```
HPS hps;          /* presentation-space handle                    */
HRGN hrgn;        /* region handle                                */
RECTL arcl[4];    /* array of structures for rectangle coordinates */

/*
 * Load the array of RECTL structures with the appropriate rectangle
 * coordinates.
 */

hrgn = GpiCreateRegion(hps, 4L, arcl);
GpiSetClipRegion(hps, hrgn);
```

40.3.5 Setting a Clip Path

Drawing and computer-aided-design (CAD) applications may require clipping to curved edges. If so, they should use a clip path to define a curved clipping area in world coordinates. But clip paths, especially ones that clip to curved edges, require considerable memory and processing time. So when clipping to a straight edge, your application should use a clip region, graphics page, or viewing limit, all of which require less memory and processing time than a clip path.

You should perform the following steps when you create a clip path:

1 Determine the shape and size (in world coordinates) of the clip path.

2 Call the **GpiBeginPath** function to begin the path definition.

3 Create the path.

4 Close the path by using the **GpiEndPath** function.

5 Create a clip path from the path definition by using the **GpiSetClipPath** function.

The following code fragment shows how to create an elliptic clip path:

```
HPS hps;        /* presentation-space handle */
POINTL ptll;    /* point structure           */

/*
 * Determine the location of the clip path and load ptll with the
 * appropriate coordinates.
 */

GpiBeginPath(hps, 1L);      /* begins path           */
GpiMove(hps, &ptll);        /* sets current position */
GpiFullArc(hps,             /* defines ellipse       */
    DRO_OUTLINE, 6553600);
GpiEndPath(hps);            /* end the path          */
GpiSetClipPath(hps, 1L, SCP_ALTERNATE | SCP_AND);
```

40.4 Summary

The following list summarizes the MS OS/2 clipping functions:

GpiExcludeClipRectangle Excludes a rectangular area from a clip path, viewing limit, graphics field, or clip region.

GpiIntersectClipRectangle Combines a rectangular area with the current clip region in order to form a new clip region. If you have not set a clip region by calling the **GpiSetClipRegion** function, the rectangle you pass to **GpiIntersectClipRectangle** becomes the current clip region (the coordinates of this rectangle are in world coordinates).

GpiQueryClipBox Determines the dimensions of the smallest rectangle that completely encloses the intersection of the current clip path, viewing limit, graphics field, and clip region. This function uses only the dimensions of the clip path, viewing limit, graphics field, and clip region that your application has set; it does not use the dimensions of the default clipping areas (which are set to infinity).

GpiQueryClipRegion Retrieves a handle of the currently selected clip region. If your application has not set a clip region, this handle will be NULL.

GpiQueryGraphicsField Determines the location (in page coordinates) of the lower-left and upper-right corners of the current graphics-field rectangle. If your application has not set the dimensions of the graphics field, MS OS/2 returns the dimensions of the default graphics field. If your application uses a normal presentation space and the default transformations, the graphics-field dimensions are 268,435,455 by 268,435,455 page units.

GpiQueryViewingLimits Determines the location (in model coordinates) of the lower-left and upper-right corners of the current viewing-limit rectangle. If your application has not set the dimensions of the viewing limit, MS OS/2 returns the dimensions of the default viewing limit. If your application uses a normal presentation space and the default transformations, the viewing-limit dimensions are 268,435,455 by 268,435,455 model units.

GpiSetClipPath Creates a clip path from a path that you created in your application. The coordinates of the clip path are in world coordinates.

GpiSetClipRegion Creates a clip region from a region that you created in your application. The coordinates of the clip region are in device coordinates.

GpiSetGraphicsField Creates a graphics field in your application. The coordinates of the graphics field are in page coordinates.

GpiSetViewingLimits Creates a viewing limit in your application. The coordinates of the viewing limit are in model coordinates.

Chapter
41

Metafiles

41.1 Introduction 551

41.2 About Metafiles 551
41.2.1 Graphics Orders 551
41.2.2 Creating Metafiles 552
41.2.3 Storing Pictures in Metafiles 552
41.2.4 Editing Metafiles 553
41.2.5 Playing Metafiles 553
41.2.6 Saving Metafiles 554

41.3 Using Metafiles 554
41.3.1 Creating and Drawing into a Metafile 554
41.3.2 Drawing into a Metafile in Retain Mode 555
41.3.3 Copying a Metafile onto Disk 556
41.3.4 Playing a Metafile 557

41.4 Summary 557

41.1 Introduction

This chapter describes a special file called a metafile. You should also be familiar with the following topics:

- Color tables
- Area fill patterns
- Logical fonts
- Segments and retained drawing
- Presentation spaces and device contexts
- Coordinate spaces and transformations

41.2 About Metafiles

A metafile is a file that contains a picture and information that MS OS/2 uses when it draws the picture. The following list describes the elements found in a metafile:

- Picture
- Logical color table
- Logical font (optional)
- Fill pattern (optional)
- Viewing transformation
- Page units
- Page dimensions

You can display the contents of a metafile (its picture) in a window on a video display, or you can print it on a printer. When displaying the contents of a metafile, you can use the logical color table, logical font, fill pattern, and transformations that are in the application's presentation space, or you can use the color table, font, fill pattern, and transformations that are in the metafile. You can save metafiles onto a disk and load them from a disk into your application, or you can transfer them from application to application by using the clipboard or from user to user over a network.

41.2.1 Graphics Orders

Unlike a bitmap, which contains color information for pels in a raster image, a metafile contains graphics orders that MS OS/2 uses to construct the picture. MS OS/2 uses several types of graphics orders, which are low-level graphics commands. Two such graphics orders represent, respectively, the drawing functions and attribute functions for lines, arcs, characters, markers, text, and bitmaps; a third graphics order represents miscellaneous functions that set transformations, call segments, and so forth. There are four graphics-order sizes:

- 1-byte orders
- 2-byte orders
- Long orders (maximum length of 256 bytes)
- Very long orders (maximum length of 64K bytes)

A 1-byte order contains a hexadecimal identifier corresponding to a drawing function or attribute function. A 2-byte order contains a hexadecimal identifier in the first byte and data in the second byte. Long and very long orders contain a hexadecimal identifier corresponding to a drawing function or attribute function, a length value that specifies how many bytes are used by the graphics-order arguments, and the actual arguments. The following example shows a long graphics order that corresponds to the **GpiLine** drawing function:

```
81 8 100 0 0 0 100 0 0 0
```

The first number, 81, is the hexadecimal identifier that corresponds to the **GpiLine** function. The second number, 8, is the length value that specifies how many bytes are used by the graphics-order arguments. The next eight bytes contain the arguments for **GpiLine**. In this case, these arguments specify the line's endpoint at (100,100). For a complete list of graphics orders, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

41.2.2 Creating Metafiles

To store graphics orders and drawing information in a metafile, you need to create a metafile device context and associate it with your application's presentation space. A metafile device context is a special device context that returns a handle to a metafile when you close it. To create a metafile device context, call the **DevOpenDC** function, passing it the OD_METAFILE type as the second argument. The following code fragment creates a metafile device context that is compatible with a display:

```
DEVOPENSTRUC dop;

dop.pszLogAddress = NULL;
dop.pszDriverName = "DISPLAY";
dop.pdriv = NULL;
dop.pszDataType = NULL;

hdcMeta = DevOpenDC(hab, OD_METAFILE, "*", 4L,
    (PDEVOPENDATA) &dop, hdcComp);
```

41.2.3 Storing Pictures in Metafiles

Once you create a metafile device context, you can associate it with your application's presentation space and begin drawing the picture. If the drawing mode is DM_DRAW, MS OS/2 stores (in the metafile) the graphics orders that correspond to drawing functions and primitive attributes. If the drawing order is DM_RETAIN, MS OS/2 does not store graphics orders in the metafile. Instead, it stores them in segments. An application can then copy these orders from segments into a metafile by calling one of the following functions:

Function	Description
GpiDrawChain	Copies the segment chain into the metafile.
GpiDrawFrom	Copies a selected range of segments from the segment chain into the metafile.
GpiDrawSegment	Copies the contents of a particular segment into the metafile.

If the drawing mode is DM_DRAWANDRETAIN, MS OS/2 stores the graphics orders in the metafile and in any open segments. You can set the drawing mode by calling the **GpiSetDrawingMode** function.

After you finish drawing into a metafile, you can obtain a handle for it by calling the **DevCloseDC** function and passing the handle that identifies the metafile device context:

```
GpiAssociate(hps, NULL);
hmf = DevCloseDC(hdcMeta);
```

The **GpiAssociate** call disassociates the metafile device context from the presentation space identified by the hps argument. You disassociate the metafile from the presentation space before you call **DevCloseDC**. Once you obtain a handle to the metafile, you can save the metafile on disk or display its contents on an output device.

41.2.4 Editing Metafiles

To edit graphics orders in a metafile, you must first copy them into an array of bytes by calling the **GpiQueryMetaFileBits** function. The number of drawing orders copied depends on the size of the array that you supply. You can determine the size of the metafile (in bytes) by calling the **GpiQueryMetaFileLength** function. Once you finish editing the graphics orders, you can copy them back into the metafile by calling the **GpiSetMetaFileBits** function.

41.2.5 Playing Metafiles

You can display or print the picture in a metafile by playing its contents with the **GpiPlayMetaFile** function. When you call this function, you must specify whether you want MS OS/2 to use the logical color table, logical font, fill pattern, viewing transformation, and device transformation that are in your application's presentation space or whether you want MS OS/2 to use the definitions for these objects in the metafile. The following code fragment shows a call to the **GpiPlayMetaFile** function that uses the definitions in the application's presentation space and ignores the definitions in the metafile:

```
alOpt[0] = OL;           /* reserved                             */
alOpt[1] = LT_DEFAULT;   /* viewing transformation in PS         */
alOpt[2] = OL;           /* reserved                             */
alOpt[3] = LC_DEFAULT;   /* font and fill pattern in PS          */
alOpt[4] = RES_DEFAULT;  /* page units and dimensions in PS      */
alOpt[5] = SUP_DEFAULT;  /* draws metafile into PS               */
alOpt[6] = CTAB_DEFAULT; /* logical color table in PS            */
alOpt[7] = CREA_DEFAULT; /* sets realizable option in color table */
GpiPlayMetaFile(hps, hmf, 8L, alOpt, OL, OL, NULL);
```

If you must use the fonts, fill patterns, color tables, and transformations from the metafile, you should call the **GpiPlayMetaFile** function and load the options array with the following flags:

```
alOpt[0] = OL;              /* reserved                              */
alOpt[1] = LT_ORIGINALVIEW; /* viewing transformation in metafile    */
alOpt[2] = OL;              /* reserved                              */
alOpt[3] = LC_LOADDISC;     /* font and fill pattern in metafile     */
alOpt[4] = RES_RESET;       /* page units/dimensions in metafile     */
alOpt[5] = SUP_DEFAULT;     /* draws metafile into PS                */
alOpt[6] = CTAB_REPLACE;    /* color table in metafile               */
alOpt[7] = CREA_DEFAULT;    /* set realizable option in color table */
GpiPlayMetaFile(hps, hmf, 8L, alOpt, OL, OL, NULL);
```

When you use the definitions in the metafile, MS OS/2 resets the presentation space using the values found in the metafile.

41.2.6 Saving Metafiles

You can save a metafile onto a disk by calling the **GpiSaveMetaFile** function. This function saves the metafile in the same directory the application is in. You cannot use this function to save a metafile into a separate directory on another drive (or even on the same drive). Once you save a metafile onto a disk, its handle is no longer valid—you cannot draw the contents of the metafile again until you load it from disk. To load a metafile that you've saved on disk, you should call the **GpiLoadMetaFile** function and pass it the name of the file. This function returns a handle to the metafile after loading it.

41.3 Using Metafiles

You can use metafile functions to perform the following tasks:

- Create a metafile.
- Draw into a metafile.
- Load a metafile onto disk.
- Load a metafile from disk into an application.
- Play a metafile.
- Edit a metafile.

41.3.1 Creating and Drawing into a Metafile

To create a metafile, you must perform the following tasks:

1. Create a metafile device context by calling the **DevOpenDC** function.
2. Create a presentation space by calling the **GpiCreatePS** function, and associate the presentation space with the metafile device context.
3. Draw into the metafile by calling various **Gpi** drawing functions.
4. Disassociate the metafile device context from the presentation space by calling the **GpiAssociate** function.
5. Close the metafile device context by calling the **DevCloseDC** function.

The following code fragment shows how to create a simple metafile that draws text within the borders of a three-color box:

```
DEVOPENSTRUC dop;

dop.pszLogAddress = NULL;
dop.pszDriverName = "DISPLAY";
dop.pdriv = NULL;
dop.pszDataType = NULL;

hdcMeta = DevOpenDC(hab,
    OD_METAFILE,                 /* metafile device context   */
    "*",                         /* ignores os2.ini           */
    4L,                          /* uses first four fields    */
    (PDEVOPENDATA) &dop,         /* device information        */
    hdc);                        /* compatible device context */
hpsMeta = GpiCreatePS(hab, hdcMeta,
    &sizlPage, PU_PELS | GPIA_ASSOC);

/* Draw a box in a metafile. */

GpiSetColor(hpsMeta, CLR_CYAN);
ptl1.x = 150;
ptl1.y = 200;
GpiMove(hpsMeta, &ptl1);
ptl2.x = 300;
ptl2.y = 275;
GpiBox(hpsMeta, DRO_FILL, &ptl2, OL, OL);

GpiSetColor(hpsMeta, CLR_GREEN);
ptl1.x = 300;
ptl1.y = 200;
GpiMove(hpsMeta, &ptl1);
ptl2.x = 390;
ptl2.y = 275;
GpiBox(hpsMeta, DRO_FILL, &ptl2, OL, OL);

GpiSetColor(hpsMeta, CLR_YELLOW);
ptl1.x = 390;
ptl1.y = 200;
GpiMove(hpsMeta, &ptl1);
ptl2.x = 530;
ptl2.y = 275;
GpiBox(hpsMeta, DRO_FILL, &ptl2, OL, OL);

ptl1.x = 175;
ptl1.y = 230;
GpiMove(hpsMeta, &ptl1);
GpiSetColor(hpsMeta, CLR_PINK);
GpiCharString(hpsMeta, 41L,
    "METAFILE COPY METAFILE COPY METAFILE COPY");
GpiAssociate(hpsMeta, NULL);
hmf = DevCloseDC(hdcMeta);
```

41.3.2 Drawing into a Metafile in Retain Mode

To draw into a metafile, you set the drawing mode to the appropriate value for your application and then perform the drawing operations. If your application is a drafting, drawing, or computer-aided-design (CAD) application, you should set the drawing mode to DM_RETAIN, perform your drawing operations in retained segments, and then copy the segments to the metafile. The following code fragment shows how to copy the contents of a segment into a metafile:

```
/*
 * Open a segment, assign it an identifier of 10,
 * and draw some text into it.
 */

GpiSetDrawingMode(hps, DM_RETAIN);
GpiOpenSegment(hps, 10L);
ptl1.x = 175; ptl1.y = 230;
GpiMove(hps, &ptl1);
GpiSetColor(hps, CLR_PINK);
GpiCharString(hps, 41L,
    "METAFILE COPY METAFILE COPY METAFILE COPY");
GpiCloseSegment(hps);

GpiAssociate(hps, NULL);          /* disassociates PS and screen DC */
GpiAssociate(hps, hdcMeta);       /* associates PS and meta DC      */
GpiDrawSegment(hps, 10L);         /* draws segment into metafile    */
GpiAssociate(hps, NULL);          /* disassociates PS and meta DC   */
hmf = DevCloseDC(hdcMeta);        /* closes metafile                */

GpiAssociate(hps, hdc);           /* associates PS and screen DC    */
GpiSetDrawingMode(hps, DM_DRAW); /* sets drawing mode to DM_DRAW   */

/*
 * Load the array of options for GpiPlayMetaFile
 * with default values.
 */

alOpt[0] = 0L;                    /* reserved                       */
alOpt[1] = LT_DEFAULT;            /* default transformations        */
alOpt[2] = 0L;                    /* reserved                       */
alOpt[3] = LC_DEFAULT;            /* uses default lcids             */
alOpt[4] = RES_DEFAULT;           /* uses default                   */
alOpt[5] = SUP_DEFAULT;           /* uses default                   */
alOpt[6] = CTAB_DEFAULT;          /* uses default                   */
alOpt[7] = CREA_DEFAULT;          /* uses default                   */
GpiPlayMetaFile(hps,              /* plays metafile onto screen     */
    hmf, 8L, alOpt, 0L, 0L, NULL);
```

If you merely want to create a simple drawing in a metafile for repeated display, you can set the drawing mode to DM_DRAW and draw directly into the metafile. The code in the first code fragment shows how to do this.

41.3.3 Copying a Metafile onto Disk

You can copy a metafile onto disk by calling the **GpiSaveMetaFile** function, and you can load the file back into your application by calling the **GpiLoadMetaFile** function. The following code fragment shows how to copy a metafile into a file named *meta.met*, then load the same file back into the application and play it:

```
GpiSaveMetaFile(hmf, "meta.met");        /* saves metafile on disk  */

hmf2 = GpiLoadMetaFile(hab, "meta.met"); /* loads metafile          */

alOpt[0] = 0L;                           /* reserved                */
alOpt[1] = LT_DEFAULT;                   /* uses default transforms */
alOpt[2] = 0L;                           /* reserved                */
alOpt[3] = LC_DEFAULT;                   /* uses default            */
alOpt[4] = RES_DEFAULT;                  /* uses default            */
alOpt[5] = SUP_DEFAULT;                  /* uses default            */
alOpt[6] = CTAB_DEFAULT;                 /* uses default            */
alOpt[7] = CREA_DEFAULT;                 /* uses default            */
GpiPlayMetaFile(hps,                     /* plays metafile          */
    hmf2, 8L, alOpt, 0L, 0L, NULL);
```

41.3.4 Playing a Metafile

You can play the contents of a metafile by calling the **GpiPlayMetaFile** function. The following code fragment shows how to play the metafile using the font, color table, and fill-pattern descriptions in your application's presentation space:

```
alOpt[0] = 0L;                /* reserved                        */
alOpt[1] = LT_DEFAULT;        /* viewing transformation in PS    */
alOpt[2] = 0L;                /* reserved                        */
alOpt[3] = LC_DEFAULT;        /* font and fill pattern in PS     */
alOpt[4] = RES_DEFAULT;       /* page units and dimensions in PS */
alOpt[5] = SUP_DEFAULT;       /* draws metafile into PS          */
alOpt[6] = CTAB_DEFAULT;      /* color table in PS               */
alOpt[7] = CREA_DEFAULT;      /* sets realizable option          */
GpiPlayMetaFile(hps, hmf, 8L, alOpt, 0L, 0L, NULL);
```

The next code fragment shows how to play a metafile using the font, color table, and fill-pattern descriptions in the metafile:

```
alOpt[0] = 0L;                /* reserved                          */
alOpt[1] = LT_DEFAULT;        /* viewing transformation in PS      */
alOpt[2] = 0L;                /* reserved                          */
alOpt[3] = LC_LOADDISC;       /* font and fill pattern in metafile */
alOpt[4] = RES_DEFAULT;       /* page units and dimensions in PS   */
alOpt[5] = SUP_DEFAULT;       /* draws metafile into PS            */
alOpt[6] = CTAB_REPLACE;      /* color table in metafile           */
alOpt[7] = CREA_DEFAULT;      /* sets realizable option            */
GpiPlayMetaFile(hps, hmf, 8L, alOpt, 0L, 0L, NULL);
```

The last code fragment shows how to play a metafile using the viewing transformation, page units, and presentation-page dimensions specified in the metafile:

```
alOpt[0] = 0L;                /* reserved                          */
alOpt[1] = LT_ORIGINALVIEW;   /* viewing transformation in metafile */
alOpt[2] = 0L;                /* reserved                          */
alOpt[3] = LC_DEFAULT;        /* font and fill pattern in PS       */
alOpt[4] = RES_RESET;         /* page units/dimensions in metafile  */
alOpt[5] = SUP_DEFAULT;       /* draws metafile into PS            */
alOpt[6] = CTAB_DEFAULT;      /* uses color table in PS            */
alOpt[7] = CREA_DEFAULT;      /* sets realizable option            */
GpiPlayMetaFile(hps, hmf, 8L, alOpt, 0L, 0L, NULL);
```

41.4 Summary

The following list summarizes the MS OS/2 metafile functions:

DevCloseDC Closes a metafile device context and returns a handle that identifies a metafile.

DevOpenDC Creates a metafile device context when you pass it the OD_METAFILE constant as the second argument.

GpiCopyMetaFile Creates a copy of a metafile.

GpiDeleteMetaFile Deletes a metafile.

GpiLoadMetaFile Loads data from disk storage into a metafile and returns a handle that identifies the metafile.

GpiPlayMetaFile Plays the contents of a metafile into a presentation space. The fourth argument to this function is an array of options that specify how MS OS/2 should alter your application's presentation space before playing the metafile. The following list describes each option and its effect on the presentation space:

Option	Description
LT_NOMODIFY	MS OS/2 uses the presentation space's viewing transformation and ignores any viewing transformations that are set in the metafile.
LT_ORIGINALVIEW	MS OS/2 replaces the presentation space's viewing transformation with that of the metafile.
LC_NOLOAD	MS OS/2 uses the presentation space's logical font, logical color table, and custom fill pattern; it will ignore any logical font, logical color table, or custom fill pattern in the metafile.
LC_LOADDISC	MS OS/2 loads and uses any logical fonts, logical color tables, or custom bitmaps (for fill patterns) from the metafile; the new objects (fonts, color tables, and bitmaps) replace the existing objects in the presentation space.
RES_NORESET	MS OS/2 uses the presentation space's page units and page dimensions.
RES_RESET	MS OS/2 replaces the page dimensions and page units of the presentation space with those of the metafile.
SUP_NOSUPPRESS	MS OS/2 draws the metafile on the device associated with the presentation space.
SUP_SUPPRESS	MS OS/2 does not draw the metafile on the device associated with the presentation space. This option is useful if you need to alter the presentation space before drawing it on an output device.
CTAB_NOMODIFY	MS OS/2 uses the color table in the presentation space.
CTAB_REPLACE	MS OS/2 replaces the color table in the presentation space with the color table in the metafile.
CREA_REALIZE	MS OS/2 sets the realizable option when it loads the color table.
CREA_NOREALIZE	MS OS/2 does not set the realizable option when it loads the color table.

GpiQueryMetaFileBits Copies the contents of a metafile (drawing orders, color table description, page units, etc.) into an array of bytes. The number of bytes copied depends on the size of the array.

GpiQueryMetaFileLength Retrieves the size of a metafile in bytes. You use this function to determine the size of the array you'll need when you call the **GpiQueryMetaFileBits** function.

GpiSaveMetaFile Copies a metafile onto a disk and then removes it from your application's memory.

GpiSetMetaFileBits Copies drawing orders from an array of bytes into a metafile. You use this function to copy edited drawing orders back into a metafile.

Chapter

42

Segments and Retained Graphics

42.1 Introduction 563

42.2 About Segments and Retained Graphics 563
42.2.1 Segments, Elements, and Graphics Orders 563
42.2.2 Segments and Subpictures 564
42.2.3 Types of Segments 564
42.2.4 Segment Attributes 565
42.2.5 Storing Graphics Orders in Segments 567
42.2.6 Creating Segments 567
42.2.7 Destroying Segments 567
42.2.8 Correlation 568
42.2.9 Editing Segments 569

42.3 Using Segments and Retained Drawings 570
42.3.1 Creating a Chained Segment 571
42.3.2 Creating a Called Segment 571
42.3.3 Drawing a Segment Chain 572
42.3.4 Performing a Correlation Operation 572
42.3.5 Editing the Contents of a Segment 572

42.4 Summary 573

42.1 Introduction

This chapter describes retained graphics and graphics segments. You should also be familiar with the following topics:

- Presentation spaces and device contexts
- Coordinate spaces and transformations
- Line drawing
- Color and mix modes

42.2 About Segments and Retained Graphics

There are two kinds of graphics output in MS OS/2: retained and nonretained. Most developers are familiar with the latter. When an application draws nonretained output by calling a graphics function, the output appears immediately on a video display. If part of the picture in the display is erased, or if part of the picture is repeated in another location on the display, the application calls the same graphics functions a second time. The fundamental drawback of nonretained graphics in drawing or computer-aided design (CAD) applications is obvious: These applications frequently draw pictures and parts of pictures repeatedly, so it is important that they be able to store the primitives used to draw the pictures. Using retained graphics, however, the developer can store graphics primitives and redraw their associated output as necessary. Rather than calling the individual graphics functions each time it is necessary to redraw the picture, the application can make a single call to display the retained graphics. In MS OS/2, applications store retained graphics in segments.

42.2.1 Segments, Elements, and Graphics Orders

A segment is a collection of elements, each element containing one or more graphics orders. A graphics order is a low-level graphics command that corresponds to a graphics function or an attribute function. There are four graphics-order sizes in MS OS/2:

- 1-byte orders
- 2-byte orders
- Long orders (maximum length of 256 bytes)
- Very long orders (maximum length of 64K bytes)

A 1-byte order contains a hexadecimal identifier corresponding to a graphics function or an attribute function. A 2-byte order contains a hexadecimal identifier in the first byte and data in the second byte. The long and very long orders each contain a hexadecimal identifier that corresponds to a graphics-drawing or attribute function, a length value that specifies how many bytes are used by the graphics-order arguments, and the actual arguments. The following example shows a long graphics order that corresponds to the **GpiLine** function:

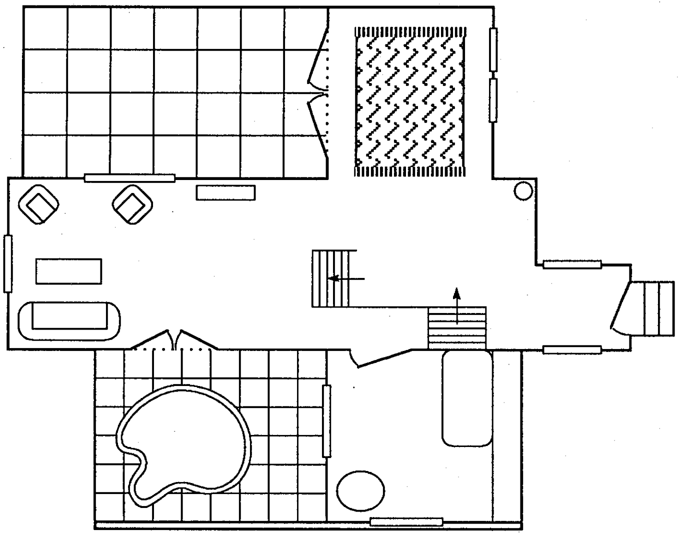
```
81 8 100 0 0 0 100 0 0 0
```

The first number, 81, is the hexadecimal identifier that corresponds to **GpiLine**. The second number, 8, is the length value that specifies how many bytes are used by the graphics-order arguments. The next eight bytes contain the arguments for **GpiLine**. In this case, these arguments specify the line's end point at (100,100). For a complete list of graphics orders, see *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

42.2.2 Segments and Subpictures

In most cases, graphics orders in a segment correspond to a subpicture, which is part of a complete, more complex picture. For example, an architectural drafting application that draws the layout of a room could store the graphics orders for a desk in one segment, the graphics orders for a chair in another segment, the graphics orders for a plant in a third segment, and so on. Each of these segments contains a subpicture, which the application combines with other subpictures to form the complete room diagram. Figure 42.1 shows a floor plan that was drawn using segments:

Figure 42.1
Combining Subpictures to Create a Floor Plan



42.2.3 Types of Segments

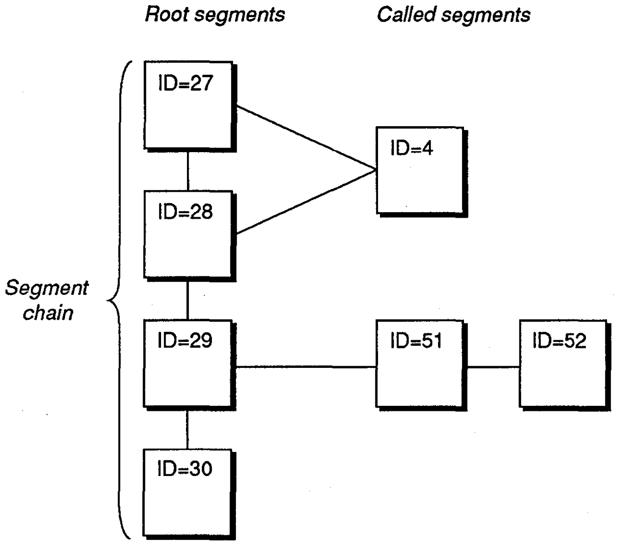
There are two types of segments: chained segments and called segments.

MS OS/2 links chained segments together in a segment chain. Chained segments are sometimes called root segments. Each presentation space can contain only one segment chain. Applications use chains to generate complete pictures from subpictures. When MS OS/2 draws subpictures in a chain, it draws the subpicture for the first segment in the chain, followed by the subpicture for the second segment in the chain, and so on. Sometimes, it is necessary to alter the order in which the subpictures are drawn. You can do this by calling the **GpiSetSegmentPriority** function. You can also draw the entire chain, part of the chain, or a single segment in the chain by calling, respectively, the **GpiDrawChain, GpiDrawFrom**, or **GpiDrawSegment** function.

A dynamic segment is a chained segment that possesses special properties. When MS OS/2 draws subpictures associated with dynamic segments, the XOR raster operation is set. An application can move the subpicture associated with a dynamic segment without destroying other subpictures in nondynamic segments. Applications draw dynamic segments by calling the **GpiDrawDynamics** function. Applications can remove the subpictures associated with dynamic segments by calling the **GpiRemoveDynamics** function.

Called segments are not linked to the chain. Instead, applications draw them by calling the **GpiCallSegmentMatrix** function from within a chained segment or another called segment. Figure 42.2 shows a typical segment chain and its associated called segments:

Figure 42.2
Chained and Called Segments



42.2.4 Segment Attributes

Segments, whether chained or called, have characteristics, called attributes, that you can set and change according to what your application needs. There are seven segment attributes, each described in the following list:

Attribute	Description
Detectable	If the detectable and visible attributes are set, your application can perform correlation operations on segments created in its presentation space.

Attribute	Description
Visible	If the visible attribute is set, the **GpiDrawChain**, **GpiDrawFrom**, and **GpiDrawSegment** functions will generate output on a device.
Chained	If the chained attribute is set, MS OS/2 adds each new segment in your application's presentation space to the segment chain.
Dynamic	If the dynamic attribute is set, MS OS/2 draws segment output using the XOR raster operation.
Fast chain	If the fast-chain attribute is set, MS OS/2 does not reset the primitive attributes to their default values before drawing the segment.
Propagate detectable	If the propagate-detectable attribute is set, MS OS/2 treats any called segments as though the detectable attribute were set for those segments.
Propagate visible	If the propagate-visible attribute is set, MS OS/2 treats any called segments as though the visible attribute were set for those segments.

When an application creates a segment in a presentation space, MS OS/2 assigns initial attributes to it. If you haven't altered the initial attributes with the **GpiSetInitialSegmentAttrs** function, five of the attributes will be set and two of the attributes will not be set. The following list describes which attributes are set and which are not:

Attribute	Default setting
Detectable	OFF
Visible	ON
Chained	ON
Dynamic	OFF
Fast chain	ON
Propagate detectable	ON
Propagate visible	ON

You can retrieve the values of the current initial attributes by calling the **GpiQueryInitialSegmentAttrs** function.

After you create a segment, you may need to alter its attributes. For example, if you created a segment using the default attributes and you want to perform a correlation operation on the subpicture in that segment, you'll need to set the detectable attribute by calling the **GpiSetSegmentAttrs** function. You can retrieve the values of the attributes for any segment by calling the **GpiQuerySegmentAttrs** function.

42.2.5 Storing Graphics Orders in Segments

There are three drawing modes that affect how MS OS/2 stores graphics orders in segments. These modes are described in the following list:

Drawing mode	Description
Draw	When the draw mode is set, it is not possible to store graphics orders in a chained segment.
Retain	When the retain mode is set, your application can store graphics orders in chained and unchained segments.
Draw-and-retain	When the draw-and-retain mode is set, your application can store graphics orders in chained and unchained segments. In this mode, output intended for a chained segment is both drawn on the device and stored in a segment.

When you create a presentation space, the drawing mode is set to draw. You can set the drawing mode to retain or draw-and-retain by calling the **GpiSetDrawingMode** function. You can determine which drawing mode is set by calling the **GpiQueryDrawingMode** function.

42.2.6 Creating Segments

In MS OS/2, applications identify segments with long integer values greater than zero. You can determine which values have already been assigned to segments by calling the **GpiQuerySegmentNames** function. This function retrieves the current segment identifiers that your application is using. Once you determine which values are assigned, you can choose a new value for the segment you are about to create. The first function you should call after you choose a segment identifier is **GpiOpenSegment**, which defines the beginning of a segment bracket. A segment bracket is a collection of graphics-drawing and attribute functions that MS OS/2 converts into graphics orders and stores in the segment. The graphics orders in a segment are organized into elements, each containing one or more graphics orders. Once you have called the necessary primitives and attribute functions, you should close the segment bracket by calling the **GpiCloseSegment** function.

42.2.7 Destroying Segments

Once you are through drawing the subpicture associated with a segment, you should delete the segment by calling the **GpiDeleteSegment** function. If you are through drawing a number of subpictures in the segment chain, you can delete an entire range of segments by calling the **GpiDeleteSegments** function. In both cases, you use the segment identifiers to identify the segment or range of segments that are no longer useful.

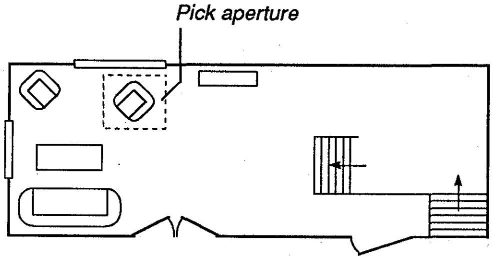
42.2.8 Correlation

Correlation is the process of determining whether graphic output appears in a particular region, called a pick aperture, in your application's page space. The pick aperture is a rectangular region that you can use to isolate part or all of a subpicture or group of subpictures. To perform correlation operations, you must set the segment attributes to detectable and visible by calling the **GpiSetSegmentAttrs** function.

If the drawing mode is retain or draw-and-retain, you can set the pick-aperture position by calling the **GpiCorrelateSegment**, **GpiCorrelateFrom**, or **GpiCorrelateChain** function. If the drawing mode is nonretained, you can set the pick-aperture position by calling the **GpiSetPickAperturePosition** function, retrieve the page-space coordinates of the center of the aperture by calling the **GpiQueryPickAperturePosition** function, increase or reduce the pick-aperture dimensions by calling the **GpiSetPickApertureSize** function, and determine the current pick-aperture dimensions by calling the **GpiQueryPickApertureSize** function. When you perform a correlation operation, MS OS/2 returns an array of long-integer pairs. The first value in the long-integer pair identifies a segment containing a subpicture that intersects the pick aperture. The second value in this pair identifies a tag. A tag identifies the specific primitives and associated graphics orders that intersect the pick aperture.

If your application will perform correlation operations, you must assign tags by calling the **GpiSetTag** function. Typically, applications assign tags only to elements that correspond to primitives. You can determine the value of the last tag assigned to an element by calling the **GpiQueryTag** function. Figure 42.3 shows the diagram of the room from Figure 42.1 as it would appear in the application's page space. In this figure, the chair intersects the pick aperture.

Figure 42.3
Correlation Operation



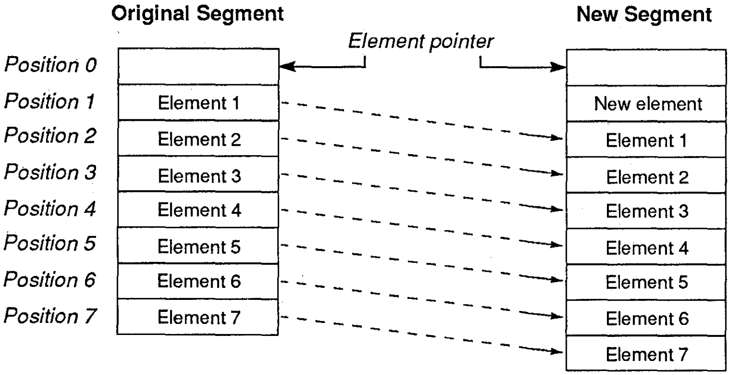
You can perform a correlation operation on the entire segment chain, part of the chain, or a single segment in the chain by calling, respectively, the **GpiCorrelateChain**, **GpiCorrelateFrom**, or **GpiCorrelateSegment** function.

42.2.9 Editing Segments

MS OS/2 provides segment-editing functions you can use to write applications that allow users to edit segments or elements in a segment. After performing a correlation operation using your application, a user may need to alter the elements that intersected the pick aperture. If your application takes advantage of the segment-editing capabilities, it should assign a label (in addition to a tag) to each element that corresponds to a graphics primitive in the segment. You assign a label, which is a long integer value, by calling the **GpiLabel** function. You can access elements in a segment with the element pointer by setting the element pointer so that it points to an element identified by a label. You can set the element pointer so that it points to an element identified by a label by calling the **GpiSetElementPointerAtLabel** function. You can also set the element pointer by adding or subtracting an offset from its current location by calling the **GpiOffsetElementPointer** function. Or you can set the element pointer by specifying an element number in the segment and calling the **GpiSetElementPointer** function. You can determine the current location of the element pointer by calling the **GpiQueryElementPointer** function.

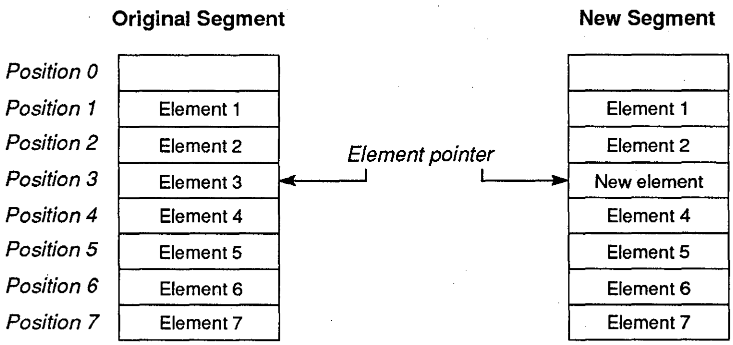
There are two edit modes in MS OS/2: insert mode and replace mode. You can set the edit modes by calling the **GpiSetEditMode** function, or you can determine which mode is currently set by calling the **GpiQueryEditMode** function. If the edit mode is set to insert, you can insert an element at the current location of the element pointer. MS OS/2 shifts the element that'was previously at that location into the next slot, and so on, until the last element is shifted into a new, final slot. Figure 42.4 shows a segment before and after a new element is inserted at position zero, the beginning of the segment:

Figure 42.4
Inserting a New Element in a Segment



If replace mode is set, you can replace the element at the current pointer location with a new element. Figure 42.5 shows a segment before and after the third element was replaced:

Figure 42.5
Replacing an Element with a New Element



In addition to inserting and replacing elements in a segment, you can delete elements from a segment. You can delete a single element at the current element-pointer location by calling the **GpiDeleteElement** function. You can delete a range of elements in a segment by calling the **GpiDeleteElementRange** function, and you can delete a series of elements between two labels by calling the **GpiDeleteElementsBetweenLabels** function.

You can copy elements from one segment to another or from one position in a segment to another position in the same segment by calling the **GpiGetData** and the **GpiPutData** functions. **GpiGetData** copies one or more graphics orders from one or more elements in a segment into a buffer of bytes. You can then copy the order from the buffer to a new location in a segment by calling **GpiPutData**. In addition to copying multiple elements from one segment to another, or from one segment to another location in the same segment, you can copy the graphics orders from a single element by calling the **GpiQueryElement** and **GpiElement** functions. **GpiQueryElement** copies the graphics orders from an element into an array of bytes. **GpiElement** copies the orders from the array back into a segment.

42.3 Using Segments and Retained Drawings

You can use segment and retained-drawing functions to perform the following tasks:

- Create a chained, chained-dynamic, or called segment.
- Draw the subpicture(s) associated with one or more segments.
- Delete a segment.
- Perform correlation operations on the subpicture associated with a segment.
- Edit the contents of a segment.
- "Drag" the subpicture associated with a dynamic segment.

42.3.1 Creating a Chained Segment

To create a chained segment, you must perform the following tasks:

1 Set the drawing mode to retain.

2 Check to see whether the chained attribute is one of the initial segment attributes by calling the **GpiQueryInitialSegmentAttrs** function.

3 If the chained attribute is not set, turn it on by calling the **GpiSetInitialSegmentAttrs** function.

4 Open the segment by calling the **GpiOpenSegment** function.

5 Perform the necessary drawing operations.

6 Close the segment by the calling the **GpiCloseSegment** function.

In the following example, the segment contains a box primitive and calls another segment with the **GpiCallSegmentMatrix** function:

```
GpiSetDrawingMode(hps, DM_RETAIN);
if (ATTR_OFF == GpiQueryInitialSegmentAttrs(hps, ATTR_CHAINED))
    GpiSetInitialSegmentAttrs(hps, ATTR_CHAINED, ATTR_ON);
GpiOpenSegment(hps, ++idSegment);
ptl.x = 150; ptl.y = 150;
GpiMove(hps, &ptl);
ptl.x = 225; ptl.y = 225;
GpiBox(hps, DRO_FILL, &ptl, OL, OL);
GpiCallSegmentMatrix(hps, --idSegment, 9L, &matlfTransform,
   TRANSFORM_REPLACE);
GpiCloseSegment(hps);
```

42.3.2 Creating a Called Segment

To create a called segment, you must perform the following tasks:

1 Set the drawing mode to retain.

2 Check to see whether the chained attribute is one of the initial segment attributes by calling the **GpiQueryInitialSegmentAttrs** function.

3 If the chained attribute is set, turn it off by calling the **GpiSetInitialSegmentAttrs** function.

4 Open the segment by calling the **GpiOpenSegment** function.

5 Perform the necessary drawing operations.

6 Close the segment by calling the **GpiCloseSegment** function.

The following code fragment shows how to draw a box in a called segment:

```
GpiSetDrawingMode(hps, DM_RETAIN);
if (ATTR_ON == GpiQueryInitialSegmentAttrs(hps, ATTR_CHAINED))
    GpiSetInitialSegmentAttrs(hps, ATTR_CHAINED, ATTR_OFF);
GpiOpenSegment(hps, idSegment);
ptl.x = 50; ptl.y = 50;
GpiMove(hps, &ptl);
ptl.x = 125; ptl.y = 125;
GpiBox(hps, DRO_FILL, &ptl, OL, OL);
GpiCloseSegment(hps);
```

42.3.3 Drawing a Segment Chain

To draw a segment chain, you call the **GpiDrawChain** function, as shown in the following code fragment:

```
if (DM_DRAW != GpiQueryDrawingMode(hps))
    GpiSetDrawingMode(hps, DM_DRAW);
GpiDrawChain(hps);
```

42.3.4 Performing a Correlation Operation

To use a correlation operation, you must perform the following tasks:

1. Size the pick aperture by calling the **GpiSetPickApertureSize** function.
2. Perform the correlation operation by calling the **GpiCorrelateChain** function, passing it the pick-aperture position as the third argument.

The following code fragment shows how to send the segment/chain identifiers from the correlation operation to your application's window with a message box:

```
HitDetect(hps, hwnd)
HPS hps;
HWND hwnd;
{
    LONG lMaxHits = 1;
    LONG lMaxDepth = 1;
    LONG alSegTag[MaxHits][MaxDepth][2]
    CHAR szChar[80];
    USHORT usCount;

    GpiCorrelateChain(hps, PICKSEL_VISIBLE, &ptlPick,
        lMaxHits, lMaxDepth, alSegTag);
    sprintf(szChar, " Segment %ld Tag %ld ", alSegTag[0], alSegTag[1]);
    WinMessageBox(HWND_DESKTOP, hwnd, szChar,
        "Segment/Tag Pairs", 0, MB_OK);

    for (usCount = 0; usCount < 2; usCount ++)
        alSegTag[usCount] = 0;
}
```

42.3.5 Editing the Contents of a Segment

To edit the contents of a segment, you must perform the following steps:

1. Set the segment edit mode to insert or replace by calling the **GpiSetEditMode** function.
2. Set the drawing mode to retain.
3. Open the segment by calling the **GpiOpenSegment** function, passing it the segment identifier from a previous correlation operation.
4. Set the element pointer so that it points to the position at which you will replace or insert an element by calling the **GpiSetElementPointer**, **GpiSetElementPointerAtLabel**, or **GpiOffsetElementPointer** function.
5. Insert the new primitives by calling any of the **Gpi** primitive functions.
6. Delete any unnecessary primitives by calling the **GpiDeleteElement** or **GpiDeleteElementRange** function.

The following code fragment shows how to insert three elements in a segment. The first element contains the graphics order that sets the color to yellow; the second element moves the current position; and the third element draws an outlined box with rounded corners. After inserting the three elements, the code deletes the elements at positions five and six in the segment (these elements were previously at positions two and three).

```
GpiSetEditMode(hps, SEGEM_INSERT);
GpiSetDrawingMode(hps, DM_RETAIN);
GpiOpenSegment(hps, 2L);
GpiSetElementPointer(hps, 1L);
GpiSetColor(hps, CLR_YELLOW);
ptl.x = 30; ptl.y = 30;
GpiMove(hps, &ptl);
ptl.x = 150; ptl.y = 150;
GpiBox(hps, DRO_OUTLINE, &ptl, 40L, 40L);
GpiDeleteElementRange(hps, 5L, 6L);
GpiCloseSegment(hps);
```

42.4 Summary

The following list summarizes the MS OS/2 segment and retained-drawing functions:

GpiBeginElement Defines the beginning of an element bracket. An element bracket is a collection of graphics orders that correspond to graphics functions and graphics-attribute functions. MS OS/2 treats the collection of graphics orders in an element bracket like a single element. The **GpiEndElement** function defines the end of an element bracket.

GpiCallSegmentMatrix Temporarily sets the model transformation, transforms the contents of a segment, and then draws the segment. You should use this function to concatenate the contents of an unchained segment bracket onto another chained or unchained segment bracket. For example, if your application repeatedly draws a single object, such as a window in a house, you could define the object (the window) in an unchained bracket and draw it repeatedly by calling **GpiCallSegmentMatrix** from within other segment brackets.

GpiCloseSegment Defines the end of a segment bracket. A segment bracket is a collection of elements that, when drawn, create a subpicture. Each element contains one or more graphics orders. The **GpiOpenSegment** function defines the beginning of a segment bracket.

GpiCorrelateChain Determines whether correlation hits occurred at the current pick-aperture location for any segment in the segment chain.

GpiCorrelateFrom Determines whether correlation hits occurred at the current pick-aperture location for any segment in a range of chained segments.

GpiCorrelateSegment Determines whether correlation hits occurred at the current pick-aperture location for a given segment.

GpiDeleteElement Deletes the element pointed to by the element pointer, then sets the element pointer so that it points to the previous element in the segment. Before deleting an element, you must set the pointer by calling the **GpiSetElementPointer** or **GpiSetElementPointerAtLabel** function. If the element pointer points to the beginning of the segment (position 0), no deletion occurs.

GpiDeleteElementRange Deletes a specified range of elements from a segment. After deleting the elements, MS OS/2 sets the element pointer so that it points to the element preceding the first of the deleted elements.

GpiDeleteElementsBetweenLabels Deletes all of the elements in a segment that appear between two labels. After deleting the elements, MS OS/2 sets the element pointer so that it points to the element preceding the first of the deleted elements.

GpiDeleteSegment Deletes a segment. If the function call occurs within a segment bracket and the segment identifier is the identifier for the currently open segment, MS OS/2 deletes the segment and ignores the remainder of the calls up to, and including, the **GpiCloseSegment** call. If the function call occurs within a segment bracket and the identifier is the identifier for a segment other than the currently open segment, MS OS/2 deletes the segment, then continues processing the remaining calls in the segment bracket. If the call occurs outside of a segment and references a segment in the segment chain, MS OS/2 removes the segment from the chain and links the two adjacent segments (if those segments exist).

GpiDeleteSegments Deletes a specified range of segments. If the segments were in the segment chain, MS OS/2 "repairs" the chain by linking the segments immediately preceding and following the deleted segments (if those segments exist).

GpiDrawChain Draws the subpictures stored in a presentation space's segment chain. MS OS/2 draws the subpictures by using four of the five drawing controls you can set by calling the **GpiSetDrawControl** function.

GpiDrawDynamics Redraws dynamic segments. If you called the **GpiRemoveDynamics** function prior to calling **GpiDrawDynamics** and you specified a range of dynamic segments, MS OS/2 draws only that range. If you set the DCTL_DYNAMIC control by calling the **GpiSetDrawControl** function, MS OS/2 calls the **GpiRemoveDynamics** function before drawing the subpictures from the dynamic segments.

GpiDrawFrom Draws the subpictures from a specified range of segments in the segment chain. MS OS/2 draws the subpictures by using four of the five drawing controls that you can set by calling the **GpiSetDrawControl** function.

GpiDrawSegment Draws a subpicture from a single segment. MS OS/2 draws the subpicture by using four of the five drawing controls that you can set by calling the **GpiSetDrawControl** function.

GpiErase Clears a window identified by a screen device context and paints the window, using the color identified by index 0 in your presentation space's color table.

GpiElement Creates an element from graphics orders that you store in a buffer and pass to the function. If the current drawing mode is DM_RETAIN or DM_DRAWANDRETAIN, MS OS/2 stores the element in a segment. If the current drawing mode is DM_DRAW or DM_DRAWANDRETAIN, MS OS/2 draws output by using the graphics orders in the buffer.

GpiEndElement Defines the end of an element bracket. An element bracket is a collection of graphics orders that correspond to graphics functions and attribute functions. MS OS/2 treats the collection of graphics orders in an element bracket as a single element. The **GpiBeginElement** function defines the beginning of an element bracket.

GpiErrorSegmentData Provides information about the last error that occurred during a retained-drawing operation. It returns a pointer to the segment identifier and a pointer to one of three constants that indicate when the error occurred: GPIE_SEGMENT, while drawing the segment; GPIE_ELEMENT, while calling the **GpiElement** function; or GPIE_DATA, while calling the **GpiPutData** function.

GpiGetData Copies graphics orders from a segment into a buffer. You can use this function to copy parts of a segment into another segment by collecting specific graphics orders from a source segment by using the **GpiGetData** function, then inserting them into a target segment by calling the **GpiPutData** function. You can also use this function to copy graphics orders into a buffer, edit the orders, then copy them back into the original segment by calling **GpiPutData.**

GpiLabel Generates a special element called a label. When a segment contains labels, you can perform the following tasks:

- Insert a new element before or after the element associated with a certain label.
- Delete an element at a label.
- Delete each of the elements that appear between two labels.

GpiOffsetElementPointer Moves the element pointer in a segment by a specified offset. If the sum of the offset and the current pointer position is less than zero, MS OS/2 sets the pointer so that it points to position 0 (preceding the first element in the segment). If the sum of the offset and the current pointer position is greater than the number of elements in the segment, MS OS/2 sets the pointer so that it points to the last element in the segment.

GpiOpenSegment Defines the beginning of a segment bracket. A segment bracket is a collection of elements that, when drawn, create a subpicture. Each element contains one or more graphics orders. The **GpiCloseSegment** function defines the end of a segment bracket.

GpiPutData Copies graphics orders from a buffer into a segment. You can use this function to copy parts of a segment into another segment by collecting specific graphics orders from a source segment using the **GpiGetData** function, then inserting them into a target segment by calling **GpiPutData**. You can also use this function to copy graphics orders into a buffer, edit the orders, then copy them back into the original segment by calling **GpiPutData.**

GpiQueryBoundaryData Retrieves the size of the smallest rectangle that completely surrounds the current segment output, if the drawing control DCTL_BOUNDARY has been set by calling the **GpiSetDrawControl** function.

GpiQueryDrawControl Determines whether one of the five drawing controls is set.

GpiQueryDrawingMode Determines which of the three drawing modes is set. The three modes are draw, retain, and draw-and-retain.

GpiQueryEditMode Determines which of the two segment-editing modes is currently set in your application's presentation space. If the edit mode is replace, you can replace the element at the current element-pointer location with a new element. If the edit mode is insert, you can insert an element at the current element-pointer location and MS OS/2 will shift each succeeding element into the next slot, creating a new, final slot.

GpiQueryElement Retrieves the graphics orders from the element at the current position of the element pointer and copies the orders into a buffer of bytes. You can use this function only if the drawing mode is retain or draw-and-retain and a segment is open.

GpiQueryElementPointer Retrieves the location of the element pointer. You can use this function only if the drawing mode is retain or draw-and-retain and a segment is open.

GpiQueryElementType Retrieves the type of the element at the current location of the element pointer. If an element is generated by a single **Gpi** function call, the type corresponds to the graphics-order code for that call. If an element is generated by the application and contains a number of **Gpi** function calls, the application can define a unique type for the element.

GpiQueryInitialSegmentAttrs Retrieves the setting of one of the seven initial segment attributes. MS OS/2 assigns these attributes to a segment when you create it in your application's presentation space.

GpiQueryPickAperturePosition Retrieves the location of the center of the pick aperture in your application's page space. The pick aperture is a rectangle that applications use when performing correlation operations.

GpiQueryPickApertureSize Retrieves the dimensions of the pick aperture in presentation-page coordinates. The pick aperture is a rectangle that applications use when performing correlation operations.

GpiQuerySegmentAttrs Retrieves the setting of one of the seven segment attributes. MS OS/2 assigns these attributes to a segment after you create it in your application's presentation space.

GpiQuerySegmentNames Retrieves a list of existing segment identifiers within a specified range. A segment identifier is a long integer value.

GpiQuerySegmentPriority Returns the identifier for a segment that precedes or follows a segment in the segment chain.

GpiQueryStopDraw Determines whether the stop-draw condition is set. You can set this condition by calling the **GpiSetStopDraw** function from one thread to stop a retained-drawing operation in another thread.

GpiQueryTag Retrieves the value of the last tag set by calling the **GpiSetTag** function. A tag is an integer identifier that applications use for correlation operations. When a correlation hit occurs, MS OS/2 returns the identifier for the segment containing the element that caused the correlation hit, as well as an identifier for the tag associated with the element. If your application performs correlation operations, you should assign a new tag to each element in a segment.

GpiRemoveDynamics Removes parts of a picture that were drawn by dynamic segments in the segment chain.

GpiResetBoundaryData Resets the boundary data to NULL. The boundary data contains dimensions of the smallest rectangle that will completely surround an application's retained-drawing output.

GpiSetDrawControl Sets one of the five drawing controls. These controls are described in the following list:

Control constant	Description
DCTL_BOUNDARY	If this control is set, MS OS/2 computes the dimensions of the smallest rectangle that would completely surround the retained-drawing output.
DCTL_CORRELATE	If this control is set, MS OS/2 performs correlation operations on any primitives or any output associated with the **GpiPutData** or **GpiElement** function.
DCTL_DISPLAY	If this control is set, MS OS/2 draws retained output on the device identified by the current device context. If this control is not set, no retained output will appear on the device.
DCTL_DYNAMIC	If this control is set, MS OS/2 calls the **GpiRemoveDynamics** function before drawing any retained output and then, after drawing the retained output, it calls the **GpiDrawDynamics** function to draw output stored in dynamic segments.
DCTL_ERASE	If this control is set, MS OS/2 calls the **GpiErase** function before drawing any retained output.

The DCTL_DISPLAY control is the only control set to DCTL_ON by default. All of the other controls are set to DCTL_OFF when you create a presentation space.

GpiSetDrawingMode Sets the drawing mode in your presentation space to one of three possible modes: draw, retain, or draw-and-retain.

GpiSetEditMode Determines which of the two segment-editing modes is currently set in your application's presentation space. If the edit mode is replace, you can replace the element at the current location of the element pointer with a new element. If the edit mode is insert, you can insert an element at the current location of the element pointer and MS OS/2 will shift each succeeding element into the next slot, creating a new, final slot.

GpiSetElementPointer Sets the element pointer so that it points at the *n*th element in a segment.

GpiSetElementPointerAtLabel Sets the element pointer to the element identified by a particular label.

GpiSetInitialSegmentAttrs Sets the default segment attributes.

GpiSetPickAperturePosition Sets the location of the center of the pick aperture in your application's page space. The pick aperture is a rectangle that applications use when performing correlation operations.

GpiSetPickApertureSize Sets the dimensions of the pick aperture in presentation-page coordinates. The pick aperture is a rectangle that applications use when performing correlation operations.

GpiSetSegmentAttrs Sets one of the seven segment attributes. MS OS/2 assigns these attributes to a segment after you create it in your application's presentation space. The segment attributes are described in the following list:

Attribute constant	Description
ATTR_CHAINED	If this attribute is set, MS OS/2 adds each new segment in your application's presentation space to the segment chain.
ATTR_DETECTABLE	If this attribute is set, your application can perform correlation operations on segments created in its presentation space.
ATTR_DYNAMIC	If this attribute is set, MS OS/2 draws segment output by using the XOR raster operation.
ATTR_FASTCHAIN	If this attribute is set, MS OS/2 resets the primitive attributes to their default values before drawing the segment chain.
ATTR_PROP_DETECTABLE	If this attribute is set, the detectable attribute will be set in each segment called by a chained segment.
ATTR_PROP_VISIBLE	If this attribute is set, the visible attribute will be set in each segment called by a chained segment.
ATTR_VISIBLE	If this attribute is set, the **GpiDrawChain**, **GpiDrawFrom**, and **GpiDrawSegment** functions will generate output on a device.

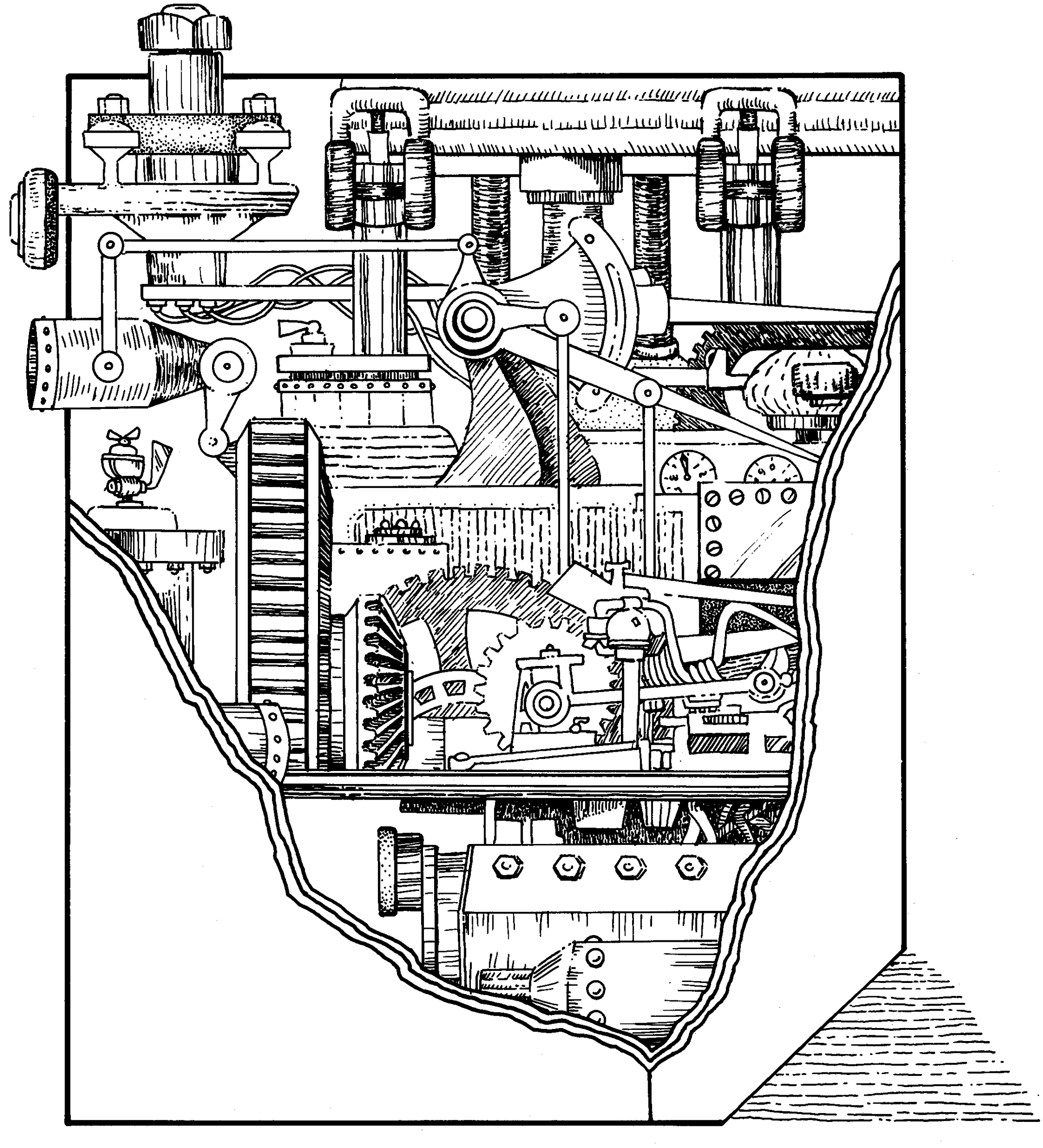
GpiSetSegmentPriority Alters the order in which MS OS/2 draws and detects segments in the segment chain.

GpiSetStopDraw Sets the stop-draw condition. You can set this condition by calling **GpiSetStopDraw** from one thread to stop a retained-drawing operation in another thread.

GpiSetTag Assigns a tag to an element in a segment. A tag is an integer identifier that applications use for correlation operations. When a correlation hit occurs, MS OS/2 returns the identifier for the segment containing the element that caused the correlation hit, as well as an identifier for the tag associated with the element. If your application performs correlation operations, you should assign a new tag to each element in a segment.

Part 4

System Services



Part 4

System Services

Chapter 43 Processes, Threads, and Sessions 585

Chapter 44 The Memory Manager 601

Chapter 45 Dynamic Linking 613

Chapter 46 The File System 621

Chapter 47 Video Input and Output 637

Chapter 48 Advanced Video Input and Output 647

Chapter 49 The Mouse 659

Chapter 50 The Keyboard 667

Chapter 51 Interprocess Communication 675

Chapter 52 Timers 693

Chapter 53 Window Timers 697

Chapter 54 Device Monitors 703

Chapter 55 Atom Tables 707

Chapter 56 System Information 715

Chapter

43

Processes, Threads, and Sessions

43.1 Introduction 587

43.2 About Processes, Threads, and Sessions 587
43.2.1 Processes and Threads 587
43.2.2 The Scheduler 588
43.2.3 Child Processes 589
43.2.4 Program Startup 590
43.2.5 Sessions 591

43.3 Using Processes 593
43.3.1 Starting a Process 593
43.3.2 Setting the Program Command Line and Environment 593
43.3.3 Running an Asynchronous Child Process 594
43.3.4 Waiting for a Child Process to End 594
43.3.5 Retrieving the Termination Status of a Child Process 595
43.3.6 Ending a Process 595
43.3.7 Terminating Another Process 595
43.3.8 Cleaning Up Before Ending a Process 596

43.4 Using Threads 597
43.4.1 Creating a Thread 597
43.4.2 Controlling the Execution of a Thread 598
43.4.3 Suspending a Thread 598
43.4.4 Changing the Priority 598

43.5 Summary 599

43.1 Introduction

This chapter describes processes, threads, sessions, and the Microsoft Operating System/2 multitasking functions used to create and manage them. You should also be familiar with the following topics:

- The file system
- Dynamic linking
- The system-configuration file (*config.sys*)

43.2 About Processes, Threads, and Sessions

Multitasking, one of the principal features of MS OS/2, is the ability of the system to manage the execution of more than one program at a time. A multitasking system such as MS OS/2 lets users simultaneously run all the programs they need to complete their work. This ability helps to optimize use of the computer, since time normally spent by a program waiting for user input is distributed to other programs that may be printing a document or recalculating a spreadsheet. Also, running more than one program at a time makes working with multiple programs easier, because the user can readily move from one program to another as the work requires.

Although MS OS/2 provides multitasking in the traditional sense of having more than one program run at a time, it also extends this concept to permit a single program to run more than one copy of itself at the same time and to provide a means of quickly moving between programs without disrupting the display or execution of the programs.

43.2.1 Processes and Threads

An MS OS/2 program that has been loaded into memory and prepared for execution is called a process. Each process has at least one thread, called the main thread or thread 1. The process consists of the program code, data, and other resources, such as files, pipes, and queues, that belong to the program. The thread consists of the current register values, the stack, and the state of execution of the program. When MS OS/2 executes a program, it confirms that the process's code and data are in memory and that the thread's registers and stack are set before it passes execution control. Each program has access to all the computer's resources, such as memory, disk drives, screen, keyboard, and the CPU itself. The system carefully manages these resources so that programs can access them without conflict.

A process can have more than one thread. Each thread runs independently, keeping its own register, stack, and execution state. Threads share the same data segment (that is, they share the program's globally defined variables). Although a thread can execute any part of the program, including a part being executed by another thread, threads are typically used to execute separate parts. This distributes the available CPU time and lets a program carry out several tasks simultaneously—for example, loading a file and prompting the user for input at the same time.

Since the system can create and execute threads quickly, the preferred multitasking method is to distribute tasks among parts of the same program instead of between programs.

A process does not have to rely on MS OS/2 to control execution of its threads. A process can use the **DosSuspendThread** and **DosResumeThread** functions to suspend and resume the execution of a given thread. When a process suspends a thread, the thread remains suspended until the process calls the **DosResumeThread** function.

A process or thread ends when it calls the **DosExit** function. MS OS/2 automatically closes any files or other resources left open by the process when the process ends, but when a thread ends, any resources it may have open remain open until another thread closes them or the process ends. A process can direct MS OS/2 to carry out other actions when the process ends by using the **DosExitList** function to create a list of termination functions. MS OS/2 calls the termination functions, in the order given, when the process is about to end.

Threads in a process must be given a stack when they are created. The stack can be in the automatic data segment of the process (that is, be defined as a global variable in the program), or it can be a segment that is explicitly allocated for use as a stack. In either case, a program that uses multiple threads may need to disable code used to check for available space in the stack if that code assumes that the program has only one stack.

Although each thread in a process has its own registers, a new thread inherits some registers, including the **es** register, from the thread that creates it. This may lead to an unexpected protection violation if the selector in the **es** register is later freed or invalidated by the first thread and the new thread unwittingly uses the register. (This can happen in high-level-language programs in which the **es** register is automatically saved and restored. Restoring an invalid selector may cause a protection violation.) A new thread can avoid a protection violation by clearing the **es** register when it first starts. Alternatively, an existing thread can clear the **es** register before creating a new thread. The following assembly-language routine, ClearES, clears the **es** register:

```
ClearES proc far
        sub ax,ax
        mov es,ax
ClearES endp
```

43.2.2 The Scheduler

The MS OS/2 system scheduler determines how to distribute execution control among the programs currently running. The scheduler uses time slicing to distribute execution control. This means the scheduler periodically gives each thread in each process a small slice of CPU time. The thread executes until its time is up; then the scheduler stops the thread and starts another. The amount of time in each time slice is defined by the **timeslice** command in the *config.sys* file. **Timeslice** sets a maximum and minimum number of milliseconds for the system's time slices.

The scheduler does not share CPU time equally among all threads. It uses a priority scheme to determine when a thread receives a time slice. The scheduler has three priority classes: time-critical, regular, and idle-time. A time-critical thread always receives a time slice before a regular thread, and a regular thread always receives a time slice before an idle-time thread.

Time-critical is a special class for threads that must react to events outside the system. Time-critical threads should execute quickly and then relinquish the CPU for other work until another time-critical event occurs. A thread in a communications program that is responsible for reading data from the communications device is a good example. The thread needs enough time to read all incoming data. Since this amount of time may be more than a regular time slice, a time-critical classification ensures that the thread gets all the time it needs.

Idle-time is a special class for threads that need very little CPU time. Idle-time threads get CPU time only when there is no other work to do.

Regular class is for all other threads.

Within each class, the scheduler maintains a priority level for a thread. The priority level can be from 0 through 31. A thread with priority level 31 always receives a time slice before a thread with priority level 30, and so on. If two or more threads have the same priority level, the scheduler distributes the CPU time equally by using a round-robin scheme; that is, the scheduler gives a time slice to first one, then another, and so on, and then goes back to the first. A process can set and retrieve the priority level by using the **DosSetPrty** and **DosGetPrty** functions.

Although you can set the priority level of a thread at any time, you should do so only for programs that use more than one thread or process. The best use of priority is to speed up threads or processes on which other threads or processes depend. For example, you might temporarily increase the priority of a thread loading a file if another thread is waiting for that file to be loaded.

Since the priority of a thread is relative to all other threads in the system, increasing the priority of the threads in your program merely to get the extra CPU time will adversely affect the overall operation of the system.

On personal computers, most programs spend a considerable amount of time interacting with the user. A program may occasionally execute without interaction for a period of time, such as when recalculating a column of numbers or formatting a paragraph of text, but for the majority of the time the program is either processing input from the user or waiting for more input. MS OS/2 can dynamically alter the priority of a process based on whether the user is using it. If the **priority** command in the *config.sys* file specifies **dynamic**, MS OS/2 grants higher priority to the foreground process than to background processes. This ensures that the foreground process—the process most likely to be in use—receives enough CPU time to provide quick response to user input. If **priority** specifies **absolute**, all processes receive CPU time based on the priority established by the **DosSetPrty** function.

43.2.3 Child Processes

Programs can load and execute other programs by using the **DosExecPgm** function. The new program, once loaded, is called a child process. The process that starts the new program is called the parent process. A child process is like any other process. It has its own program code and data and its own threads. The child process inherits the other resources of the parent process, such as files, pipes, and queues. The child process can use the inherited resources without preparing them. For example, if the parent process opens a file for reading and then starts the child process, the child process can read from the file immediately; it does not have to open the file for itself. Once the child process has been created, however, any additional resources that the parent process may create are not available to the child process. Similarly, any resources that the child process may create are not available to the parent process.

The parent process determines how the child process should run. A child process can run independently of the parent process—that is, both can run at the same time—or the parent process can wait until the child process ends before continuing execution. A process can use the **DosCwait** function to retrieve the termination status of a child process that runs independently.

43.2.4 Program Startup

Unlike MS-DOS, MS OS/2 does not prepare a program segment prefix (PSP) for protected-mode programs. Instead, it creates an environment segment that contains the definitions for environment variables and the command line used to start the program. When a program first starts, MS OS/2 gives the program access to this environment segment, as well as to information about the version of MS OS/2.

A program can retrieve definitions from the environment segment by first using the **DosGetEnv** function to retrieve the segment selector. The **DosScanEnv** and **DosSearchPath** functions also permit the program to use the information in the environment segment.

When MS OS/2 first starts a program, it sets the CPU registers to the following values:

Value	Description
cs:ip	Contains the program's initial entry point. This is the same as the entry point specified in the executable file.
ss:sp	Contains the starting address of the stack. This is the same stack address as specified in the executable file.
ds	Contains the segment selector of the automatic data segment. The automatic data segment is specified in the executable file.
es	Contains zero.
ax	Contains the segment selector of the environment segment. The environment segment contains the environment strings and command-line arguments for the program.
bx	Contains the offset to the start of the program command line from the beginning of the environment segment.
cx	Contains the size (in bytes) of the automatic data segment. If this value is zero, the segment is 65,536 bytes.
bp	Contains zero.

A program written in assembly language can use these registers to access the command line and environment string. Programs written in high-level languages can retrieve the environment-segment selector and command-line offset by using the **DosGetEnv** function.

When MS OS/2 starts a dynamic-link library, it sets the CPU registers to the following values:

Value	Description
cs:ip	Contains the entry-point address of the library initialization function.
ss:sp	Contains the address of the current system stack. The initialization function can use the stack to define local variables and call other functions, but it must restore the stack to its previous state before returning.
ds	Contains the segment selector of the library's automatic data segment (if it has one), or contains the selector for the data-segment program or system library that called the **DosLoadModule** function to load the library.
ax	Contains the module handle of the dynamic-link library.

All other register values are undefined.

The initialization function can carry out any task, but it must return by using an intersegment return instruction. All registers except **ax** and **dx** must be restored to their previous values. The function can use the **ax** register to indicate whether it was successful. The system expects the function to set the **ax** register to a nonzero value if the function was successful, or to zero if the function failed.

43.2.5 Sessions

MS OS/2 uses sessions to help the user move from one program to the next without disrupting the screen display of a program. A session consists of at least one process and either a full, character-based screen or a Presentation Manager window. When the system creates a session, the process in the session displays output in the screen or window. The user can view the output and supply input by moving to the session. The user moves to a session by pressing the ALT+ESC key combination or by selecting the title of the session from the list of program titles in the Task Manager window.

A process creates and controls sessions by using the MS OS/2 session-manager functions. Processes that use these functions have much the same control over sessions as does Task Manager, but only for the sessions they create. The functions are typically used by debugging programs to keep output of the program being debugged separate from the debugger's output.

A process creates a new session by using the **DosStartSession** function. The function uses a **STARTDATA** structure that specifies the name of the program to start in the session. It also specifies whether the session should be full-screen or in a window. When a session is created, MS OS/2 adds the program title for the session to the list of titles in Task Manager.

A session can be either a child session or an unrelated session. A child session, created by setting the **Related** field in **STARTDATA** to TRUE, is under the control of the process that created it. The process can select, set, or stop a child session by using the **DosSelectSession**, **DosSetSession**, or **DosStopSession** function, respectively. **DosStartSession** gives the child session a unique session identifier for use in these functions.

An unrelated session, created by setting the **Related** field to FALSE, is not under the process's control. Once an unrelated session starts, it is controlled entirely by the user.

A session can be either a foreground or a background session. A process can create a foreground session only if the process is in the current foreground session or if one of the sessions created by the process is the current foreground session.

A process can select a child session by using the session identifier in the **DosSelectSession** function. Selecting a child session causes that session to move to the foreground. A process can make a child session "unselectable" by using the **DosSetSession** function to change the **SelectInd** field in the **STATUSDATA** structure. This prevents the user from selecting the session from Task Manager but does not prevent a process from selecting the child session by using **DosSelectSession.**

A process can also bind a child session to its own session by using the **DosSetSession** function. Binding a session causes that session to move to the foreground whenever the user selects the parent session from Task Manager.

A process can use a session identifier with the **DosSetSession** function only if the process created the identifier. It cannot use session identifiers created by other processes.

Although a child session is related to the session that started it, the *processes* in the child and original sessions are not related. This means that even though **DosStartSession** supplies the process identifier of the process in the child session, the identifier cannot be used with MS OS/2 functions such as **DosSetPrty**.

A process can stop a child session by using the **DosStopSession** function. Stopping the session terminates the process in the session. It also stops any sessions related to the child session. If a child session is in the foreground when it is stopped, the parent session becomes the foreground session. The **DosStopSession** function breaks any bond that exists between the parent session and the specified child session.

A process running in the session specified in the **DosStopSession** function can refuse to terminate. If this happens, **DosStopSession** still returns zero (indicating success). The only way to be certain that the child session has terminated is to wait for notification through the termination queue that is specified in the **DosStartSession** function. MS OS/2 writes a data element into the specified queue when the child session terminates. The process in the parent session must call the **DosReadQueue** function to retrieve this data element, which contains the session identifier for the child session and the result code for the process in the child session. **DosReadQueue** also sets the request word to zero. Only the process that created the child session can read the data element. After reading and processing the data element, the process must free the segment that contains the data element by using the **DosFreeSeg** function.

Because the **DosStartSession** function does not supply a session identifier for an unrelated session, the process that created the unrelated session cannot select it, bind it, stop it, or make it unselectable.

43.3 Using Processes

To work successfully with multitasking, you need to understand clearly the difference between a process and a thread. A process is simply the code, data, and other resources of a program in memory, such as the open files, allocated memory, and so on. MS OS/2 considers every program that it loads to be a process. A thread, which is everything else required to execute the program, consists of a stack, the state of the CPU registers, and an entry in the execution list of the system scheduler. Every process has at least one thread, and the program executes when the system scheduler gives the thread execution control.

43.3.1 Starting a Process

You can start a process by using the **DosExecPgm** function. The process you start is a child of the starting, or parent, process and inherits many of the resources owned by the parent process, such as open files.

The following code fragment starts a program named **abc**:

```
CHAR achModuleName[128];
RESULTCODES resc;

DosExecPgm(achModuleName,       /* object-name buffer */
    sizeof(achModuleName),      /* length of buffer   */
    EXEC_SYNC,                  /* sync flag          */
    NULL,                       /* argument string    */
    NULL,                       /* environment string */
    &resc,                      /* address of result  */
    "abc.exe");                 /* name of program    */
```

This example starts **abc** so that it runs synchronously (as specified by the EXEC_SYNC constant). This means that the parent process temporarily stops while the child process executes and does not continue until the child process ends.

The MS OS/2 command processor, *cmd.exe,* is an example of a program that uses the EXEC_SYNC constant to start most child processes. That is, the processor waits for each child process to end before it prompts the user for the next command. The command processor also lets the user start asynchronous programs by using the **detach** command. When the user detaches a program, the command processor places the program in the background and continues to prompt for input.

43.3.2 Setting the Program Command Line and Environment

When you start a process, it inherits the resources of the parent. This includes open files, such as the standard-input and standard-output files. A child process also inherits the resources of the screen group, such as the mouse and video modes, and the environment variables of the parent process.

The **DosExecPgm** function determines the command line and environment that the child process receives. The fourth and fifth parameters of the function are pointers to the command line and the environment, respectively. If these pointers are NULL, the child process receives nothing for a command line and only an exact duplicate of the parent process's environment. The parent process can modify this information by creating a string (ending with two null characters) and passing the address of the string to the function, as in the following code fragment, which passes the string "test –options" to the child process as its command line:

```
CHAR achModuleName[128];
RESULTCODES resc;
CHAR chCommandLine[] = "test -options\0";

DosExecPgm(achModuleName,      /* object-name buffer */
    sizeof(achModuleName),     /* length of buffer   */
    EXEC_SYNC,                 /* sync flag          */
    chCommandLine,             /* argument string    */
    NULL,                      /* environment string */
    &resc,                     /* address of result  */
   "abc.exe");                 /* name of program    */
```

In general, any number of null-terminated strings can be passed, as long as the last string ends with two null characters.

43.3.3 Running an Asynchronous Child Process

You can use the EXEC_ASYNC constant in the **DosExecPgm** function to start a child process and let it run asynchronously (that is, without causing the parent process to pause until the child process ends). If you start a process in this way, the function copies the process identifier of the child process to the **codeTerminate** field of the **RESULTCODES** structure. You can use this process identifier to check the progress of the child process or to terminate the process.

You can also run a child process asynchronously by using **DosExecPgm** with the EXEC_ASYNCRESULT constant. This constant has the same effect as EXEC_ASYNC, except that it also directs MS OS/2 to save a copy of the child process's termination status when the child process terminates. This status specifies the reason that the child process stopped. The parent process can retrieve the termination status by using the **DosCwait** function.

43.3.4 Waiting for a Child Process to End

You can synchronize the execution of a process with the execution of one of its child processes by using the **DosCwait** function. When a process calls the **DosCwait** function, the function waits until the specified child process has finished before returning. This can be useful, for example, if the parent process needs to ensure that the child process has completed its task before the parent process continues with its own task.

In the following code fragment, the parent process waits for the child process that is specified by the process identifier in the variable pidChild:

```
RESULTCODES resc;
PID pidParent;
PID pidChild;

DosCwait(DCWA_PROCESS, DCWW_WAIT, &resc, &pidParent, pidChild);
```

You can cause a process to wait for all child processes to end by using the constant DCWA_PROCESSTREE in the **DosCwait** function.

43.3.5 Retrieving the Termination Status of a Child Process

MS OS/2 saves the termination status for a process if the process was started by using **DosExecPgm** with the EXEC_ASYNCRESULT constant. You can retrieve the termination status of the most recently terminated process by using the DCWW_NOWAIT constant with the **DosCwait** function. This constant directs the function to return immediately, without waiting for a process to end. Instead, the function retrieves the termination status from the most recent process to end.

43.3.6 Ending a Process

You can exit from a process by using the **DosExit** function. When you exit from a process, the system stops the process and frees any existing resources that it owns. If no other invocation of the process is running, the system frees the code and data segments of the process.

In the following code fragment, the **DosExit** function is used to exit from the process if the given file does not exist:

```
VOID cdecl main( )
{
    HFILE hf;
    USHORT usAction, cbWritten;

    usError = DosOpen("sample.txt", &hf, &usAction, OL, FILE_NORMAL,
        FILE_OPEN, OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, OL);

    if (usError) {
        DosWrite(2, "Cannot open file\r\n", 18, &cbWritten);
        DosExit(EXIT_PROCESS, usError);
    }
}
```

The EXIT_PROCESS constant directs the function to exit not just from the process, but from the thread that is calling the function as well. The **DosExit** function includes an error code that is returned to the parent process through the **RESULTCODES** structure specified in the **DosExecPgm** function that started the process. If you started the program from the command line, the command processor (*cmd.exe*) makes this value available through the ERRORLEVEL variable. If another process started the program, that process can call the **DosCwait** function to retrieve the error-code value.

If you want to exit only from a given thread, you can use the **DosExit** function with the EXIT_THREAD constant. This call exits from the thread without affecting other threads in the process. If the thread that you exit from also happens to be the last thread in the process, the process also exits. If the thread consists of a function, the thread exits when the function returns.

43.3.7 Terminating Another Process

One process can terminate the execution of another process by using the **DosKillProcess** function. The following code fragment terminates the specified process and all child processes belonging to that process:

```
PID pidProcess;

DosKillProcess(DKP_PROCESS, pidProcess);
```

In this example, the pidProcess variable specifies which process to terminate. Typically, you would assign the variable the process identifier for the child process. This is the process identifier that is returned by the **DosExecPgm** function when you start the child process.

43.3.8 Cleaning Up Before Ending a Process

Since in MS OS/2 any process can terminate any other process for which it has a process identifier, there is a chance that your program may lose information if a process terminates the program before the program can save its work. To prevent this loss of data, you can create a list of exit functions that clean up data and your files before MS OS/2 terminates a process. The system calls the functions on the list whenever your program is being terminated, whether by another process or by itself.

You create an exit list by using the **DosExitList** function. The function requires a function code that specifies an action to take and a pointer to the function that is to receive control upon termination. The following code fragment adds the locally defined function SaveFiles to the exit list:

```
#define INCL_SUB
#define INCL_DOSPROCESS
#include <os2.h>

VOID cdecl main() {
    DosExitList(EXLST_ADD, SaveFiles);
    .
    .
    .
}

VOID PASCAL FAR SaveFiles(usTermCode)
USHORT usTermCode;
{
    switch (usTermCode) {
        case TC_EXIT:
        case TC_KILLPROCESS:
            VioWrtTTY("Good-bye\r\n", 10, 0);
            break;

        case TC_HARDERROR:
        case TC_TRAP:
            break;

    }
    DosExitList(EXLST_EXIT, 0);
}
```

Any function that you add to the exit list must be declared with the far attribute and must have one parameter. The function can carry out any task, as shown in the preceding example, but as its last action it must call the **DosExitList** function, specifying the EXLST_EXIT constant. An exit-list function must not return and must not call the **DosExit** function to terminate.

To execute the exit-list functions, MS OS/2 reassigns thread 1 after terminating any other threads in the process. If thread 1 has already exited (for example, if it called the **DosExit** function without terminating other threads in the process), then the exit-list functions cannot be executed. In general, it is poor practice to terminate thread 1 without terminating all other threads.

You can use **DosExit** with the EXLST_REMOVE constant to remove a function from the exit list.

43.4 Using Threads

Every process has at least one thread, called the main thread or thread 1. To execute different parts of a program simultaneously, you can start several threads.

A new thread inherits all the resources currently owned by the process. This means that if you opened a file before creating the thread, that file is available to the thread. Similarly, if the new thread creates or opens a resource, such as another file, that resource is available to the other threads in the process.

43.4.1 Creating a Thread

You can use the **DosCreateThread** function to create a new thread for a process. To do so, you need the address of the code to execute, the address of the first byte in a stack, and a variable to receive the identifier of the thread. The address of the code is typically the address of a function that is defined within the program. The address of the stack can be either an address within a variable declared by using the data segment of the process or an address in a separate segment. To ensure that the new thread's stack will not be written over by other threads, you must not declare the stack within the stack segment of a process. You must also be sure that there is adequate space on the stack. The amount of space needed depends on a number of factors, including the number of function calls the thread makes and the number of parameters and local variables used by each function. If you plan to call MS OS/2 functions, a reasonable stack size is 4096 bytes.

The following code fragment creates a thread:

```
BYTE abStack[4096];
TID tidThread;

VOID main() {
    DosCreateThread(ThreadFunc, &tidThread, abStack + sizeof(abStack));
    .
    .
    .
}

VOID FAR ThreadFunc(VOID)
{
    VioWrtTTY("Message from new thread\r\n", 25, 0);
}
```

In this example, the array absStack is used for the new thread's stack. The thread identifier is copied to the tidThread variable. The thread starts execution with the first statement in the locally defined function ThreadFunc.

In MS OS/2, all stacks grow down in memory—that is, the first byte of the stack is in high memory and the last byte is in low memory—so you need to specify the last word in the stack (in this case, absStack[2048]) when you supply the starting address of the stack in the call to **DosCreateThread**.

A thread continues to run until it calls the **DosExit** function or returns control to the operating system. In the preceding example, the thread exits when the function implicitly returns control at the end of the function.

43.4.2 Controlling the Execution of a Thread

The **DosSuspendThread** and **DosResumeThread** functions let you temporarily suspend the execution of a thread if you do not need it and then resume execution when you do need it. These functions are best used when a process needs to temporarily suspend execution of a thread that is in the middle of a task. For example, consider a thread that opens and reads files from the disk. If other threads in the process do not need input from these files, the process can suspend execution of the thread so that the system scheduler does not needlessly grant execution control to the thread.

43.4.3 Suspending a Thread

You can temporarily suspend the execution of a thread for a set interval of time by using the **DosSleep** function. This function suspends execution of the thread for the specified number of milliseconds. **DosSleep** is useful if you need to delay the execution of a task. For example, you can use **DosSleep** to delay response to the user's pressing a DIRECTION key. This gives the user time to observe the results and release the key. The following code fragment uses **DosSleep** to suspend execution of a thread for 1000 milliseconds (1 second):

```
DosSleep(1000L);
```

43.4.4 Changing the Priority

You can use the **DosSetPrty** function to change the execution priority of threads in a process. The execution priority defines when or how often a thread receives an execution time slice. Threads with higher priorities receive time slices before those with lower priorities. Threads with equal priority receive time slices in a round-robin order. If you raise the priority of a thread, the thread will execute more frequently.

You can set the priority for just one thread in a process, for all threads in a process (and thus for the process itself), or for all threads in a process and in its child processes. The following code fragment uses **DosSetPrty** to lower the priority of a process that is intended to be used as a background process:

```
PIDINFO pidi;

DosGetPID(&pidi);
DosSetPrty(PRTYS_PROCESS, PRTYC_IDLETIME, 0, pidi.pid);
```

The **DosGetPID** function retrieves the process and thread identifiers and copies them to the **pid** and **tid** fields in a **PIDINFO** structure. The **DosSetPrty** function then uses the process identifier to change the priority to idle-time (idle-time processes receive the least attention by the scheduler).

You can also retrieve the priority of a process or thread by using the **DosGetPrty** function. For example, the following code fragment copies the priority of the process specified by pidiInfo.pid to the usPriority variable:

```
DosGetPrty(PRTYS_PROCESS, &usPriority, pidiInfo.pid);
```

43.5 Summary

MS OS/2 provides the following multitasking functions:

DosCreateThread Creates a new thread.

DosCwait Waits for a child process to terminate.

DosExecPgm Loads and starts a child process.

DosExit Ends a process or a specific thread.

DosExitList Specifies a function to be executed when the current process ends.

DosGetEnv Retrieves the addresses of the environment table and an offset into the environment where the command line that started the process is stored.

DosGetPID Retrieves the process, thread, and parent-process identifiers for the current process.

DosGetPPID Retrieves the parent-process identifier for the current process.

DosGetPrty Retrieves the execution priority of a process or thread.

DosKillProcess Terminates the specified process and, optionally, its child processes.

DosResumeThread Restarts a thread stopped by the **DosSuspendThread** function.

DosScanEnv Searches an environment segment for a specific environment variable.

DosSearchPath Searches the specified path for the given filenames.

DosSelectSession Switches the specified child session to the foreground.

DosSetPrty Sets the execution priority of a process or thread.

DosSetSession Sets the status of a child session.

DosSleep Suspends execution of the current thread for a specified time interval.

DosStartSession Starts another session and specifies the program to be started in that session.

DosStopSession Terminates a session started by the **DosStartSession** function.

DosSuspendThread Suspends execution of a thread until the corresponding call to the **DosResumeThread** function is executed.

Chapter

44

The Memory Manager

44.1 Introduction 603

44.2 About the Memory Manager 603

44.2.1 Memory Limits 603

44.2.2 Protection Violations 604

44.3 Using the Memory Manager 604

44.3.1 Segments 604

44.3.2 Huge Memory 605

44.3.3 Moving and Swapping 606

44.3.4 Discardable Segments 607

44.3.5 Shared Segments 608

44.3.5.1 Named Shared Segments 608

44.3.5.2 Unnamed Shared Segments 609

44.3.6 Heaps 609

44.3.7 Code-Segment Aliases 610

44.4 Summary 611

44.1 Introduction

This chapter describes the MS OS/2 memory manager. The memory manager lets programs allocate memory for their own use or to be shared with other programs. A program can allocate a segment, a huge segment, or memory blocks within a segment. You should also be familiar with the following topics:

- The system-configuration file (*config.sys*)
- Heaps
- Interprocess communication
- Module-definition files

44.2 About the Memory Manager

In MS OS/2, you allocate memory in segments of one or more bytes (up to 65,536 bytes). Each segment is identified by a unique value called a segment selector. A segment selector, combined with a segment offset, yields the address of a byte in the segment.

MS OS/2 lets you allocate any number of segments and then use these segments as additional storage for your program. When MS OS/2 first loads your program, it gives the program at least one data segment, called the automatic data segment. This segment contains the global and static data declared in the program. It may also contain the program stack. MS OS/2 also gives your program one or more code segments, which contain the program code. The code segments are protected from any changes, intentional or otherwise.

44.2.1 Memory Limits

Although MS OS/2 can access no more than 16 megabytes of physical memory, it can manage segment selectors representing up to 1 gigabyte of memory. MS OS/2 uses compaction, discarding, and swapping to manage this "virtual" memory.

When MS OS/2 runs only protected-mode programs (the **protectonly** command in the *config.sys* file is set to **yes**), all physical memory is available to protected-mode code and data segments except the memory addresses from 640K to 1000K, which are reserved for system ROM and for video buffers. When MS OS/2 runs both real-mode and protected-mode programs (**protectonly** is set to **no**), it reserves lower memory for real-mode programs. The **rmsize** command in the *config.sys* file specifies the size and the upper address of real-mode memory. The system places any protected-mode code and data segments above the last upper real-mode address. Real-mode memory is not subject to swapping or moving unless the program running in real-mode memory does so itself.

In protected mode, the system creates one local descriptor table (LDT) for each program. All threads in a program use the same LDT, so all segment selectors in the program are available to all threads. MS OS/2 reserves the global descriptor table (GDT) for the system. Also, MS OS/2 does not permit a program to examine or modify its own LDT.

The number of selectors available to a program depends on the number of code and data segments in the program. Since programs cannot use the GDT, only the approximately 8000 selectors in the LDT are available. However, half of these selectors are reserved for shared segments, so a program actually never has more than about 4000 selectors.

44.2.2 Protection Violations

In MS OS/2, memory is carefully managed by the virtual-memory and memory-protection capabilities of the Intel 80286 microprocessor. MS OS/2 grants your program access to a segment only if the segment has been explicitly allocated by your program or made available for your program's use.

Each segment has a unique selector that identifies the segment and grants a program access. If a program attempts to access a segment that is not assigned to it, uses an unknown selector, or attempts to access bytes outside of a segment, the system interrupts the program and executes the exception-handling routine for protection violations. This routine determines the cause of the exception and displays an error message to the user before terminating the program.

Protection violations are not recoverable. This means that you should be careful about using address offsets and selectors, ensuring that they are valid. Since protection violations commonly occur during program debugging, each message displayed for a protection violation shows the contents of the system registers when the violation occurred. If the violation occurs as a result of passing a system function an invalid pointer, the message also shows the parameters used in the call.

44.3 Using the Memory Manager

This section describes how you can use the MS OS/2 memory-manager functions and configuration commands to control the use of memory for your MS OS/2 programs.

44.3.1 Segments

You can allocate a segment of memory by using the **DosAllocSeg** function. You simply specify the amount of memory that you need and a variable to receive the selector created by **DosAllocSeg** to identify the new segment. The following code fragment allocates 512 bytes of memory:

```
SEL selArray;

DosAllocSeg(512, &selArray, SEG_NONSHARED);
```

You can allocate from 1 to 65,536 bytes for a segment. (To allocate 65,536 bytes, specify a size of 0 in **DosAllocSeg.**) To access a byte in the segment, you need to create a far pointer to the segment. You can do this by using the **MAKEP** macro, which combines the selector from **DosAllocSeg** and an address offset to create a pointer to the byte you want, as shown in the following code fragment:

```
PCH pch;

pch = MAKEP(selArray, 0);
```

The selector applies only to the bytes allocated for the segment. You cannot use the selector to access any other part of system memory. If you supply an incorrect offset or attempt to access a byte outside the segment, MS OS/2 displays a "protection violation" message and terminates the program.

The initial content of a segment is undefined, so it is a good idea to initialize the new memory segment immediately after creating it.

If a segment that you have allocated is too small or too large to meet your needs, you can avoid protection violations or wasted space by using the **DosReallocSeg** function to reallocate the segment. This function adjusts the size of the segment to a given size without changing the segment selector. In the following code fragment, the **DosReallocSeg** function expands the segment from 512 bytes to 1024 bytes:

```
SEL selBuf;

DosAllocSeg(512, &selBuf, SEG_NONSHARED);
     .
     .
     .
DosReallocSeg(1024, selBuf);
```

If you decrease the size of a segment, you lose any data at the end of the segment, but all other data is preserved. If you increase the segment size, new, uninitialized bytes appear at the end of the segment. You can retrieve the size in bytes of any segment by using the **DosSizeSeg** function.

When you have finished using a segment, you can free it by using the **DosFreeSeg** function. Freeing the segment returns that memory to the system's pool of available memory and invalidates the selector that identifies the segment. The data in the freed segment is lost and any attempt to access the segment by using the selector of the freed segment is a protection violation.

44.3.2 Huge Memory

Although segments are limited to 64 kilobytes, you can allocate more than 64 kilobytes of memory at a time by using the **DosAllocHuge** function. This function allocates as much memory as is requested, up to the limit available in the system. It allocates several 64K segments but ensures that the segment selectors are consecutive. (This does not mean that the segments are in consecutive memory.) You can then access the memory by computing the appropriate segment selector and offset with the help of the **DosGetHugeShift** function.

With **DosAllocHuge**, you specify the number of 64K segments you want, instead of the number of bytes. If the block of memory you need is not a multiple of 64 kilobytes, you can also specify 1 to 65,535 bytes for the last segment. The following code fragment allocates 328,192 bytes:

```
SEL selHuge;

DosAllocHuge(5,         /* allocates five 64K segments            */
    512,                /* plus 512 additional bytes              */
    &selHuge,           /* first segment selector                 */
    6,                  /* reserves six selectors for reallocation */
    SEG_NONSHARED);     /* allocation flags                       */
```

DosAllocHuge supplies only one segment selector for a huge-memory block. This is the selector for the first segment. To access bytes in the other segments, you must calculate the segment selector by adding the segment-selector offset one or more times to the selector of the first segment.

The selector offset is computed by using the huge-segment shift count retrieved by the **DosGetHugeShift** function. The shift count is an exponent of 2, so a segment-selector offset equals the value 1 shifted left by the shift count. The following code fragment shows how to compute the selector offset:

```
USHORT usShiftCount;
USHORT offSelector;

DosGetHugeShift(&usShiftCount);
offSelector = 2 << usShiftCount;
```

To create a pointer to bytes in huge memory, use the **MAKEP** macro as shown in the following code fragment:

```
PCH pch;

pch = MAKEP(selHuge+offSelector, 127); /* byte 65663 in huge memory */
```

If you write MS OS/2 programs in assembly language, the global symbols DOSHUGESHIFT and DOSHUGEINC are available for computing huge-memory addresses. DOSHUGESHIFT is equal to the shift count retrieved by **DosGetHugeShift**, and DOSHUGEINC is equal to the selector offset.

You can change the size of a huge-memory block by using the **DosReallocHuge** function, and you can free a huge-memory block by using the **DosFreeSeg** function. Freeing the selector of the first segment in a huge-memory block frees all segments.

You can use the **DosMemAvail** function to determine the size of the largest available block of free memory. This is a reasonable size to start with if you want to allocate the largest possible amount of memory with the **DosAllocHuge** function. **DosMemAvail** retrieves the available memory in bytes, so you will need to compute the number of available segments from this value before using **DosAllocHuge**. If there are many programs running that allocate memory, or if your system is swapping, the value retrieved by **DosMemAvail** may not always be accurate.

44.3.3 Moving and Swapping

The system moves data and code segments and swaps data segments whenever necessary. Moving segments gives the system the opportunity to collect all free space into one contiguous block so that it can offer much larger blocks of memory than would otherwise be possible.

Swapping of segments occurs whenever a program requests more memory than is free, thus allowing the system to handle more segments than can fit in physical memory. If there is enough space in the system swap file, *swapper.dat*, the system copies one or more data segments to the file. It copies these segments back into memory when they are needed again, possibly swapping other data segments out in the process.

The system can also discard segments if it needs additional space. In a sense, discarding is similar to swapping, except that the discarded segment is not copied to the disk. The system discards code segments if the space is needed and if the segments were loaded originally from an executable file on a hard disk. In MS OS/2, all code segments are pure—that is, not subject to change during execution—so the system can always retrieve a fresh copy of a discarded code segment from the original executable file. (For programs loaded from floppy disks, the original disk may not be present when needed, so no code segments are discarded.) For more information on discardable segments, see Section 44.3.4.

Although a program cannot control moving and swapping, the user can specify whether the system may move and swap segments by including the **memman** command in the *config.sys* file. For special-purpose programs, you can request the user to disable swapping and/or moving. Typically, disabling these features enhances the performance of the system when it is needed for dedicated, time-critical tasks.

If the **memman** command specifies **move**, MS OS/2 consolidates free space in memory by compacting the existing code and data segments. MS OS/2 compacts memory by moving the code and data segments together so that no free memory appears between them. If the **memman** command specifies **swap**, MS OS/2 writes selected data segments to the *swapper.dat* file whenever insufficient physical memory exists for a given allocation request. MS OS/2 selects the data segments to swap based on when they were last used. If a program later needs the segments, MS OS/2 reads them from the *swapper.dat* file into memory, possibly swapping other segments to make room. If the **memman** command specifies **nomove** or **noswap**, MS OS/2 does not move segments or does not swap segments, respectively.

Through swapping, programs can allocate more memory than actually exists in the computer. The exact amount of memory available to a program depends on the size of the *swapper.dat* file as well as on the number of other programs already running. The size of the *swapper.dat* file can be specified by including the **swappath** command in the *config.sys* file. Actually, **swappath** specifies how much free space to reserve on the disk. MS OS/2 adjusts the size of *swapper.dat* as needed, always leaving other files on the drive and the requested free space untouched.

Swapping and moving do not affect the segment selectors. A selector remains valid even though the location of the segment may change.

44.3.4 Discardable Segments

A discardable segment provides a convenient way to store data that you need infrequently and can regenerate easily. When you use a discardable segment, the system is free to automatically discard your segment if it needs the space. Since the segment selector of a discarded segment remains valid, you can restore the segment by simply reallocating it to the original size and again filling it with the data.

You can create discardable segments by specifying the SEG_DISCARDABLE option when you allocate segments by using the **DosAllocSeg** or **DosAllocHuge** function, as shown in the following code fragment:

```
SEL selTempData;

DosAllocSeg(256, &selTempData, SEG_DISCARDABLE);
```

MS OS/2 locks a discardable segment when it allocates it. While the segment is locked, you can examine and modify its contents without risk of the system's discarding the segment. After examining or modifying the segment, you can unlock it by using the **DosUnlockSeg** function. Unlocking the segment does not discard the segment, although it does permit the system to discard the segment at any subsequent time. To access the discardable segment again, you must first relock it by using the **DosLockSeg** function.

The system discards the segment when it needs to, but only if the segment is not locked. After the segment is discarded, any data in the segment is lost, but the segment still exists. If the program that owns the segment attempts to access the data without reallocating the segment, a protection violation results.

Before you attempt to read from or write to a segment, you must check to see if the segment has been discarded. If it has, you can reallocate the segment (to restore it to its original size) and then fill it again. You can determine whether a segment has been discarded by checking the return value of the **DosLockSeg** function. If the value is nonzero, the segment has been discarded.

44.3.5 Shared Segments

A shared segment is a memory segment that two or more programs can read from and write to. Shared segments are typically used in programs as an efficient way to share data. MS OS/2 prepares a shared segment in such a way that any program that can retrieve its segment selector can access the data in the segment. The shared segment is still protected against access by programs that do not have the segment selector.

There are two basic methods of using shared segments. One method allows two or more programs to share the same segment at the same time. Both programs read from and write to the segment, usually controlling access to the segment by using a semaphore. The other method allows one program to prepare data in a segment and then pass that segment on to another program for further processing. The first program usually releases the segment after passing it along, so that only one program accesses it at a time.

44.3.5.1 Named Shared Segments

A named shared segment has the special property of having a unique name. Any program that has the segment name can use **DosGetShrSeg** to retrieve the segment selector and access the data in the segment. Named shared segments are typically used to share a segment between two or more programs at the same time.

The name of a named shared segment has the following form:

\sharemem\*name*

The *name* parameter must conform to the rules for an MS OS/2 filename. However, no file is actually created for the segment.

You can allocate a named shared segment by using the **DosAllocShrSeg** function. The segment can have from 1 to 65,536 bytes. **DosAllocShrSeg** does not initialize a shared segment, so the content of a segment when first allocated is unknown.

You can free a named shared segment by using the **DosFreeSeg** function. The segment is not removed from memory until the last program with access to the segment frees it. You can also change the size of the segment by using the **DosReallocSeg** function.

44.3.5.2 Unnamed Shared Segments

You can allocate an unnamed shared segment by using the **DosAllocSeg** or **DosAllocHuge** function and specifying either the SEG_GETTABLE or SEG_GIVEABLE option when you allocate the segment. Sharing segments in this way is more difficult than sharing named segments, since the program creating the segment must somehow pass the segment selector to the other programs. This is typically done by using some form of interprocess communication, such as a queue or a named pipe.

If you allocate a segment with the SEG_GETTABLE option, you can pass the segment selector to another program and that program can gain access to the segment by using the **DosGetSeg** function to validate the passed selector. If you allocate a segment with the SEG_GIVEABLE option, you must create a new segment selector by using the **DosGiveSeg** function and the process identifier of the program that is to receive the new selector. Once you pass the new selector to the other program, it can use the selector immediately—that is, it does not need to use the **DosGetSeg** function.

44.3.6 Heaps

A heap is an area of memory, usually within a single segment, from which you can allocate blocks of memory. A heap is useful in programs that need to allocate many small blocks of memory. Since MS OS/2 programs have only a finite number of selectors available for use, using a heap to allocate small blocks of memory is more efficient than allocating a segment for each block. A heap lets you reserve segment selectors for large blocks of memory while satisfying your program's need for small blocks of memory.

MS OS/2 provides two methods of creating and using a heap. One method, described in Chapter 25, "Heaps," lets you create a heap that supports movable blocks of memory and fast allocation using free lists. The alternative method, described in this section, provides a less powerful but simpler way of allocating blocks of memory in a heap.

To use the alternative method for your program, you first create a heap by using the **DosAllocSeg** and **DosSubSet** functions. **DosAllocSeg** allocates the segment to contain the heap, and **DosSubSet** sets up the segment for use as a heap. The segment size can be up to 65,536 bytes. The following code fragment creates a heap having 1024 bytes:

```
SEL selHeap;

DosAllocSeg(1024, &selHeap, SEG_NONSHARED);
DosSubSet(selHeap,              /* selector for heap segment */
        1,                      /* initialize heap           */
        1024);                  /* heap size                 */
```

You can create any number of heaps for your program. Each heap is uniquely identified by the selector of the segment containing the heap.

Once you have the heap, you use the **DosSubAlloc** function to allocate blocks of memory and the **DosSubFree** function to free the memory. Each block of memory has a unique offset from the beginning of the segment. You can combine this offset with the segment selector to create a pointer to the block, as shown in the following code fragment:

```
USHORT offBlock;
PBYTE pb;

DosSubAlloc(selHeap, &offBlock, 256);   /* 256 bytes in block */
pb = MAKEP(selHeap, offBlock);
     .
     .
     .
DosSubFree(selHeap, offBlock, 256);     /* free block         */
```

The **DosSubAlloc** function always rounds the given size to the next multiple of 4, so a memory block is always at least 4 bytes. Also, MS OS/2 reserves 12 bytes in the heap for its own use. Thus, even though the segment containing the heap may have 256 bytes, only 244 are available for allocation. It is your responsibility to ensure that you do not allocate more bytes than are available in the heap.

You must be especially careful when using pointers to the memory blocks. MS OS/2 does not provide the protection against invalid offsets in a memory block that it does for a segment. Using a pointer that has an invalid offset may destroy data in another block or even destroy the heap.

If you need additional space or less space in a heap, you can always change the size of the segment by using the **DosReallocSeg** function. If you do reallocate the segment, you need to use the **DosSubSet** function again to adjust the heap to the new size. The following code fragment adjusts an existing heap to 2048 bytes:

```
DosReallocSeg(2048, selHeap);
DosSubSet(selHeap, 0, 2048);
```

Whenever you use a heap, the memory blocks you allocate are completely contained in the segment and any action the system carries out on the segment is applied to all blocks in the segment. For example, the **DosFreeSeg** function frees the segment and all blocks of memory in the segment. This is a quick way to remove the heap when you no longer need it.

The **HEAPSIZE** statement in a program's module-definition file does not apply to heaps created by using the **DosSubSet** function. **HEAPSIZE** specifies how much memory to reserve in the program's automatic data segment for a heap created by using the **WinCreateHeap** function. For more information about **WinCreateHeap**, see Chapter 25, "Heaps."

44.3.7 Code-Segment Aliases

A code-segment alias is a special segment selector that lets a program pass execution control to code in a data segment. Normally, MS OS/2 does not permit control of execution to be passed to addresses in data segments, but a code-segment alias makes the data segment appear to be a code segment. You can create a code-segment alias for a data segment by using the **DosCreateCSAlias** function.

Code-segment aliases are useful in programs that need to create and execute code while the program is running. Since MS OS/2 protects code segments against changes, the only way to create and execute code in this way is to write it to a data segment, create an alias, and use the alias to pass execution control to the code.

44.4 Summary

MS OS/2 provides the following memory-manager functions:

DosAllocHuge Allocates a huge-memory block.

DosAllocSeg Allocates a memory segment.

DosAllocShrSeg Allocates a shared memory segment.

DosCreateCSAlias Creates an aliased code-segment selector for a specified memory segment.

DosFreeSeg Frees a specified memory segment.

DosGetHugeShift Retrieves the shift count used to compute the segment-selector offset for huge memory segments.

DosGetSeg Obtains access to the specified shared memory segment.

DosGetShrSeg Retrieves the selector to a shared memory segment.

DosGiveSeg Creates a new segment selector for a shared memory segment.

DosLockSeg Locks a discardable segment in memory so that it cannot be discarded.

DosMemAvail Retrieves the size of the largest block of free memory currently available.

DosReallocHuge Reallocates a huge-memory block, changing its size.

DosReallocSeg Reallocates a memory segment, changing its size.

DosSizeSeg Retrieves the size, in bytes, of the specified segment.

DosSubAlloc Allocates memory from a segment previously allocated by using the **DosAllocSeg** or **DosAllocShrSeg** function and initialized by using the **DosSubSet** function.

DosSubFree Frees memory previously allocated by the **DosSubAlloc** function.

DosSubSet Initializes a segment for suballocation or changes the size of a previously initialized segment.

DosUnlockSeg Unlocks a discardable segment so that it can be discarded if space is needed by other segments.

Chapter

45

Dynamic Linking

45.1 Introduction 615

45.2 About Dynamic Linking 615

45.3 Building Dynamic-Link Libraries 616

45.3.1 The Source File 616

45.3.2 The Module-Definition File 617

45.3.3 Import Libraries 617

45.3.4 The Initialization Function 619

45.4 Using Dynamic Linking 620

45.5 Summary 620

45.1 Introduction

This chapter describes the portions of MS OS/2 that allow you to dynamically link functions in dynamic-link libraries to your programs. It also describes how to build dynamic-link libraries. You should also be familiar with the following topics:

- The system-configuration file (*config.sys*)
- Module-definition files

45.2 About Dynamic Linking

Dynamic linking is a way for a program to gain access at run time to functions that are not part of the program's executable code. With dynamic linking, a program loads a dynamic-link library (also called a module), retrieves the address of a function in that library, and calls the function to carry out a task.

Any application can use the functions contained in a dynamic-link library. The only requirement is that you must import the function name so that the reference can be resolved when you run the application. There are three ways to import the name of a dynamic-link function:

- Specify the import library of the dynamic-link library during linking.
- Specify the function name in the **IMPORTS** statement of the application's module-definition file.
- Use the MS OS/2 dynamic-link functions to import the function during execution of the application.

For most applications, function names are resolved by using an import library. For example, the file *os2.lib* is an import library that contains import records for all the MS OS/2 functions. Because much of the code for MS OS/2 is in dynamic-link libraries, the import library is needed to prepare MS OS/2 programs for execution.

The **IMPORTS** statement in a program's module-definition file is similar to the import library, except that you must supply the information needed by the linker to build the import records of the functions you use.

The MS OS/2 dynamic-link functions let a program load dynamic-link libraries and import functions while the program runs. Although the dynamic-link functions can be used to import any functions (including the MS OS/2 system functions), they are more often used to import functions in device drivers or in special-purpose libraries. Using an import library or the **IMPORTS** statement is more common than using the dynamic-link functions, since in those cases MS OS/2 automatically loads and links the specified function for you.

You can load any number of dynamic-link libraries and create links to any of the functions in the libraries. It is a good idea to free libraries whenever you are not using them, however, since this frees memory and avoids unnecessary swapping.

45.3 Building Dynamic-Link Libraries

This section describes how to build a dynamic-link library. A dynamic-link library usually contains executable code for common functions. With standard libraries, such as the C-language run-time library, the code for common functions is combined with the program code when the program is created, but with a dynamic-link library the code is not combined. Instead, the program contains nothing more than an import record of the function and the name of the dynamic-link library containing it. The function is not linked until the program needs it.

Using a dynamic-link library saves memory, since the library is loaded only once. All programs that use functions in the library share the library's code. To avoid conflicts, each program uses its own stack when calling a library function.

After creating the dynamic-link library, remember to copy it to one of the directories specified by the **libpath** command in your *config.sys* file.

45.3.1 The Source File

The following simple dynamic-link-library source file, *dynlink.c*, illustrates how a dynamic-link library is built:

```
#include <os2.h>
int _acrtused = 0;

EXPENTRY Sample(pchString, cbLength)
PCH pchString;
USHORT cbLength;
{
    USHORT usBytes;
    DosWrite(1, pchString, cbLength, &usBytes);
}
```

As with all MS OS/2 programs, the *os2.h* file is included in the library (since the **DosWrite** function is called). Notice that the **_acrtused** variable is declared and initialized to zero. This directs the linker not to load the *crt0* start-up module, which is not needed with dynamic-link libraries.

The function Sample is declared with the EXPENTRY attribute. This attribute, defined in the *os2.h* file, identifies the function as an export entry. Export entries use the Pascal calling conventions and require a far call to invoke them. The far call is necessary because a dynamic-link library always resides in a different segment from that of the calling program.

Dynamic-link libraries can contain either global or local variables. If your library uses global variables, the data segment of the calling program must be pushed on the stack and then reloaded with the library's own data segment. The calling program's data segment must be restored before control returns from the library. Compilers such as the Microsoft C Optimizing Compiler provide command-line options and special keywords to automatically insert the code needed to push and pop the data segment. Strings are always treated as global variables.

Dynamic-link libraries must not assume that the stack and the global data are located in the same segment. Since a calling program must use its own stack when calling a function in a dynamic-link library, the ss register will not be equal to the **ds** register if the dynamic-link library has its own data segment.

You must not include stack probes when compiling a dynamic-link library. Because the stack setup for dynamic-link libraries differs from the stack setup for MS OS/2 programs, stack checking would cause unpredictable results.

45.3.2 The Module-Definition File

Each dynamic-link library must use export definitions to make its functions directly available to other modules. You supply an export definition for a dynamic-link library in its module-definition file.

A module-definition file describes the name, attributes, exports, imports, and other characteristics of a program or library for MS OS/2. This file is optional for MS OS/2 programs but is required for dynamic-link libraries.

For example, the dynamic-link library *dynlink.dll* contains the function Sample, which can be called from other programs. To make Sample available to calling programs, the module-definition file *dynlink.def* contains the following lines:

```
LIBRARY DYNLINK
EXPORTS SAMPLE
```

The **LIBRARY** statement tells the linker that the file *dynlink* is a dynamic-link library. The **EXPORTS** statement tells the linker that the function Sample in *dynlink* is available for other programs to call. (The Microsoft C Optimizing Compiler converts the names of functions that use the Pascal calling conventions to uppercase letters.) A program that calls the dynamic-link library *dynlink* must specify an import definition for Sample (unless an import library for *dynlink* exists) by using the following **IMPORTS** statement in the program's module-definition file:

```
IMPORTS DYNLINK.SAMPLE
```

You must list every function that will be called at run time in the module-definition file of the dynamic-link library.

45.3.3 Import Libraries

An alternative way to specify import definitions is to create an import library. If you have many functions in a dynamic-link library, it is usually easier to create an import library than to use import definitions. The import library *os2.lib* is an example of an import library that is linked with all MS OS/2 applications.

To generate an import library, you must first create a module-definition file for the dynamic-link library. As in the preceding example, this file should contain **LIBRARY** and **EXPORTS** statements. Once you have created the module-definition file, use the Microsoft Import Library Manager (**implib**) to create an import library to use during linking.

For example, to create the import library *dynlink.lib* for the dynamic-link library *dynlink.dll,* run **implib** as follows:

```
implib dynlink.lib dynlink.def
```

If your program makes calls to the newly created dynamic-link library, its import-library name must be included in the library field of the linker command line. For example, if the program **dcall** calls *dynlink.dll,* link **dcall** with the library as follows:

```
link /noi dcall.obj,,,os2.lib dynlink.lib,dcall.def
```

You can also use assembly language to write a dynamic-link library. If you choose to do so, you must declare each function in your library as follows:

functionname **proc far**

You should also declare the functions to be public.

Your assembly-language dynamic-link-library functions must set up a stack frame for the parameters being passed to them. To set up the stack frame, a function must do the following:

- Set up the frame pointer (**bp**).
- Set up space for local variables.
- Save the old **ds** register and set up the data segment for your dynamic-link library.
- Save the **si**, **di**, and **ss** registers (if the values will be changed).
- Push MS OS/2 parameters on the stack.

The function must first set up a frame pointer by pushing the old **bp** register value on the stack and setting the **bp** register equal to the **sp** register, as follows:

```
push    bp
mov     bp,sp
```

After pushing the old **ds** register value on the stack and setting the **bp** register equal to the **sp** register, the function must move the stack pointer down and push local variables on the stack in they are needed. For example, the following lines save the old **bp** value, assign the **bp** register to **sp**, and move **sp** so that it points to local variables:

```
push    bp
mov     bp,sp
sub     sp,04
```

Local variables are pushed on the stack in the positions (**bp**–2), (**bp**–4), (**bp**–6), and so on. Thus, you can use these addresses to access the variables.

Before returning to the calling program, the function should restore **sp** by setting it equal to **bp**.

If you are creating a dynamic-link library that has an automatic data segment, each function must set the **ds** register to the proper value when it starts. To do this, the function should set DGROUP so that it points to the data segment of the dynamic-link library, as follows:

```
DGROUP  GROUP   data
```

Each function in the dynamic-link library that needs to use the library's automatic data segment must save the old data segment and reload the **ds** register so that it points to the data segment of the dynamic-link library. Placed at the beginning of the function, the following lines of code do this:

```
push    ds
mov     ax, seg DGROUP
mov     ds,ax
```

Before the function returns to the calling program, it must restore the old **ds** register value by popping it off the stack, as follows:

```
pop     ds
```

Functions that use the **si**, **di**, and **ss** registers should save the old register values by pushing them on the stack, after setting the frame pointer and allocating any existing local data. The following example shows how to push these registers on the stack:

```
push    bp
mov     bp,sp
sub     sp,4
push    si
push    di
push    ss
```

Before returning to the calling program, the function must pop the registers off the stack in reverse order, as follows:

```
pop     ss
pop     di
pop     si
```

Since MS OS/2 uses the Pascal calling convention to pass parameters to the MS OS/2 system functions, it is recommended that you use the same convention for the dynamic-link-library functions that you create. Although the Pascal calling convention is not required, it provides a consistent interface for programs that call functions in dynamic-link libraries. If the Pascal calling convention is used, applications must push parameters on the stack before calling the dynamic-link-library function. (Under the Pascal convention, the number of parameters required by a given function is fixed.)

When the calling program exits from a function for which parameters were pushed on the stack, the called function should always restore the **ds** and **bp** registers and then use a **ret** *nn* instruction, where *nn* is the number of bytes pushed upon entry.

Return values are returned in the **ax** (16-bit) register or in the **dx:ax** (32-bit) registers, where **dx** holds the high 16 bits and **ax** holds the low 16 bits.

45.3.4 The Initialization Function

You can optionally add an initialization function to a dynamic-link library. This function, which is called before the actual code for the dynamic-link library is called, performs user-defined operations. As with dynamic-link-library functions, you must push existing registers on the stack before using them in an initialization routine and restore them in reverse order before exiting from the routine.

Be aware that when MS OS/2 starts executing a dynamic-link library, it sets the CPU registers to known values. For a list of these initial values, see Chapter 43, "Processes, Threads, and Sessions."

Initialization functions must be written in assembly language, because the assembly-language **end** statement is required to identify the start of the function. In C, there is no equivalent **end** statement. You can, however, write your initialization code in assembly language and the library functions in C, assemble and compile these files separately, and then combine them during linking.

45.4 Using Dynamic Linking

All MS OS/2 programs use dynamic linking, since this is the only way programs can link to and call the MS OS/2 system functions.

If you use the dynamic-link functions to work with dynamic-link libraries, the first step is to load the library by using the **DosLoadModule** function. This function takes the name of the library as a parameter and returns a handle for the module. If the library is not in one of the directories specified by the **libpath** command in the *config.sys* file, you must supply a full path with the filename. If the library is not found, the function returns an error.

After the library is loaded, the program can retrieve the address of a specific function by using the **DosGetProcAddr** function. **DosGetProcAddr** takes a function name as a parameter and supplies the function's address. The function name must be spelled exactly as it appears in the library; this is typically in all uppercase letters if the Pascal calling convention is used. The function can be called like any MS OS/2 system function. The calling program must provide correct parameters in the correct order and of the correct type.

After a program has finished using a library, it can free the library by using the **DosFreeModule** function. If no other program is using the library, MS OS/2 removes the library from memory. The system keeps only one copy of a library in memory, no matter how many programs have loaded it.

You can use the **DosGetModHandle** and **DosGetModName** functions to help manage the libraries you have loaded.

45.5 Summary

MS OS/2 provides the following dynamic-link functions:

DosFreeModule Frees the specified dynamic-link module.

DosGetModHandle Retrieves the module handle for the specified dynamic-link library.

DosGetModName Retrieves the name and path of the module identified by the specified module handle.

DosGetProcAddr Retrieves the address of the specified function in the given dynamic-link module. Use this function to link to specific functions in a dynamic-link library.

DosLoadModule Loads the specified dynamic-link library.

Chapter
46

The File System

46.1 Introduction 623
46.2 About the File System 623
46.3 Using the File System 626
46.3.1 Opening Files 626
46.3.2 Creating Files 628
46.3.3 Reading from and Writing to Files 628
46.3.4 Reading and Writing Asynchronously...................... 629
46.3.5 Closing a File 629
46.3.6 Using the Standard Files 630
46.3.7 Redirecting the Standard Files 631
46.3.8 Using Wildcard Characters to Search for Files 631
46.3.9 Moving the File Pointer 632
46.3.10 Accessing Devices 633
46.3.11 Controlling Input and Output Devices..................... 634
46.4 Summary 634

46.1 Introduction

This chapter describes the file-system functions. These functions let a program open, read, write, and modify disk and device files. They also let a program access and maintain the file system—that is, the volumes, directories, and files on the computer's disk drives. You should also be familiar with the following topics:

- Files
- Devices

46.2 About the File System

In MS OS/2, the file system specifies how data is organized on the computer's mass-storage devices, such as hard (fixed) and floppy (removable) disks. Each disk drive represents a unique element of the file system through which the data on a disk can be accessed. Each drive is assigned a unique letter to distinguish it from other drives. On most personal computers, drive A is the first floppy-disk drive, drive B is the second floppy-disk drive, drive C is the first hard-disk drive, and drive D is the second hard-disk drive. A personal computer running MS OS/2 can have up to 26 drives. For exceptionally large hard disks, the disk can be divided into two or more "logical" drives. A logical drive represents a portion of the hard disk (up to 32 megabytes of storage) and, like a physical drive, is assigned a unique letter to distinguish it from other physical and logical drives.

The file system organizes disks into volumes, directories, and files. A volume is the largest file-system unit. It represents all the available storage on the disk in the drive. An optional volume identifier or name identifies the disk. Each volume has a root directory. The root directory lists the contents of the disk. Each directory entry identifies the name, location, and size of a specific file or subdirectory on the disk. A file is one or more bytes of data stored on the disk. Subdirectories provide an additional level of organization and, like the root directory, contain directory entries.

The MS OS/2 file-system functions identify files and directories by their names. These functions store or search for the file or directory in the current directory on the current drive unless the name explicitly specifies a different directory and drive. Valid MS OS/2 filenames have the following form:

[*drive*:][*directory*\]*filename*[.*extension*]

The *drive* parameter must name an existing drive and can be any letter from A through Z. The drive letter must be followed by a colon (:).

The *directory* parameter specifies the directory that contains the file's directory entry. This value must be followed by a backslash (\) to separate it from the filename. If the specified directory is not in the current directory, *directory* must include the names of all directories in the path, separated by backslashes. The root directory is specified by using a backslash at the beginning of the name. For example, if the directory *abc* is in the directory *sample* and *sample* is in the root directory, the correct *directory* specification is *\sample\abc*. A directory name consists of any combination of up to eight letters, digits, or the following special characters:

```
$ % ' - _ @ { } ~ ` ! # ( )
```

A directory name can also have an extension, which is any combination of up to three letters, digits, or special characters, preceded by a period (.).

The *filename* and *extension* parameters specify the file. A filename can be any combination of up to eight letters, digits, or the following special characters:

```
$ % ' - _ @ { } ~ ` ! # ( )
```

An extension can be any combination of up to three letters, digits, or special characters, preceded by a period.

Although these file-naming conventions apply to MS OS/2 versions 1.0 and 1.1, future releases of MS OS/2 may use other conventions. Therefore, programs written for MS OS/2 should not depend on any specific filename format.

When a program starts, it inherits the current directory and drive from the program that starts it. A program can determine which directory and drive are current by using the **DosQCurDir** and **DosQCurDisk** functions. A program can change the current directory and drive of the file system by using the **DosChDir** and **DosSelectDisk** functions.

When a program creates a new file, the system adds an entry for the file to the specified directory. Each directory can have any number of entries, up to the physical limit of the disk. A program can create new directories and delete existing directories by using the **DosMkDir** and **DosRmDir** functions. Before a directory can be deleted, it must be empty; if there are files or directories in that directory, they must be deleted or moved. A program can delete a file by using the **DosDelete** function or move a file to another directory by using the **DosMove** function.

Each directory entry includes a file attribute. The file attribute specifies whether the directory entry is for a file, a directory, or a volume identifier. It also specifies whether the file is read-only, hidden, archived, or system. A read-only file cannot be opened for writing, nor can it be deleted. A hidden file (or directory) cannot be displayed using an ordinary directory listing. A system file is excluded from normal directory searches. The archived attribute is for use by special-purpose programs that need some way to mark a file for backing up or removal. A program can retrieve and set the file attribute of a file by using the **DosQFileMode** and **DosSetFileMode** functions.

A program can retrieve information about the file system on a given drive by using the **DosQFSInfo** function. The file-system information defines the amount of storage space available on the disk. The storage space is given in number of allocation units (clusters) on the disk. Each cluster has an associated number of sectors; each sector contains a given number of bytes. A typical disk has 512 bytes for each sector and 4 sectors for each cluster. The **DosSetFSInfo** function lets a program change the volume identifier for the disk in the given drive.

A program can retrieve and set information about a specific file by using the **DosQFileInfo** and **DosSetFileInfo** functions. File information consists of the dates and times that the file was created, last accessed, and last written to; the size (in bytes) of the file; the number of sectors (or clusters) the file occupies; and the file attributes.

A program can search for all filenames that match a given pattern by using the **DosFindFirst, DosFindNext,** and **DosFindClose** functions. These functions let a program search the current directory for files whose names match a specified pattern. The pattern must be an MS OS/2 filename and can include wildcard characters. The wildcard characters are the question mark (?) and the asterisk (*). The ? matches any single character; the * matches any combination of characters. For example, the pattern *a** matches the names *abc*, *a23*, and *abca*, but the pattern *a?c* matches only the name *abc*.

A program can open an existing file or create a new file by using the **DosOpen** function. MS OS/2 identifies each open file by assigning it a file handle when the program opens or creates the file. The file handle is a unique 16-bit integer. The program can use the handle in functions that read from and write to the file, depending on how the file was opened. The program can continue to use the handle in this way until the file is closed. MS OS/2 sets the default maximum number of file handles for a program to 20. A program can change this maximum by using the **DosSetMaxFH** function.

A program can increase or decrease the size of an open file by using the **DosNewSize** function.

When a program opens a file, it must specify whether it wants to read from the file, write to it, or both read and write. Also, since more than one program may attempt to open a file, a program must specify how it wants to share the file (if at all) with other programs. A file can be shared for reading only, for writing only, for reading and writing, or not shared. A file that is not shared cannot be opened by another program (or more than once by the first program) until it has been closed by the first program.

A program reads from and writes to a file by using the **DosRead** and **DosWrite** functions. These functions read and write a specified number of bytes of data. The data is read and written exactly as given; the functions do not format the data—that is, they do not convert binary data to decimal strings or vice versa. A program can use the **DosReadAsync** and **DosWriteAsync** functions to read from and write to a file asynchronously—that is, to read and write while the rest of the program continues carrying out other operations.

Every open file has a file pointer that specifies the next byte to be read or the location to receive the next byte that is written. When a file is first opened, the system places the file pointer at the beginning of the file. As each byte is read or written, the system advances the pointer. A program can also move the pointer by using the **DosChgFilePtr** function. When the pointer reaches the end of the file and an attempt is made to read from the file, no bytes are read and no error occurs. Thus, reading zero bytes without an error means the program has reached the end of the file.

When a program writes to a disk file, MS OS/2 usually collects the data being written in an internal buffer and writes to the disk only when the amount of data equals (or is a multiple of) the sector size of the disk. If there is data in the internal buffer when the file is closed, the system automatically writes the data to the disk before closing the file. A program can also flush the buffer (that is, write the contents of the buffer to the disk) by using the **DosBufReset** function.

Although MS OS/2 lets more than one program open a file and write to it, the programs that do so must take care not to write over each other's work. A program can protect against this problem by temporarily locking a region in a file. The **DosFileLocks** function specifies a range of bytes in the file that is locked by the program and can be accessed only by that program. The program uses the same function to free the locked region.

Some file-system functions can also be used to access other input and output devices, such as serial ports, keyboard, and screen. A program can open these devices by using the **DosOpen** function and specifying one of the following device names:

clock$
com1
com2
com3
com4
con
kbd$
lpt1
lpt2
lpt3
mouse$
nul
pointer$
prn
screen$

As with a disk file, MS OS/2 supplies a unique handle that the program can use in subsequent calls to the **DosRead** and **DosWrite** functions to read from and write to the device. A program can determine the type of handle—device or file-system—by using the **DosQHandType** function. The program uses the **DosClose** function to close the handle when the device is no longer needed.

When a program opens a device, it must specify both how the device is to be accessed (read, write, or both) and what kind of sharing is to be allowed. A program can check or alter this state by using the **DosQFHandState** and **DosSetFHandState** functions. (These functions also apply to file-system handles.)

When a program starts, it inherits all open file handles from the process that starts it. If the system's command processor starts a program, file handles 0, 1, and 2 represent the standard-input, standard-output, and standard-error files. The standard-input file is the keyboard; the standard-output and standard-error files are the screen. A program can read from the standard-input file and write to the standard-output and standard-error files immediately; it does not have to open the files first.

A program can create a duplicate file handle for an open file by using the **DosDupHandle** function. A duplicate handle is identical to the original handle. Typically, duplicate handles are used to redirect the standard-input and standard-output files. For example, a program can close handle 0, open a disk file, and duplicate the disk-file handle as handle 0. Thereafter, reading from handle 0 takes data from the disk file, not from the keyboard.

46.3 Using the File System

Input and output are two of the most important tasks that any program carries out. This section explains how to read from and write to files on disks and other input and output devices, such as printers, modems, and the system console.

46.3.1 Opening Files

Before carrying out any input or output operation, you need a file handle. A file handle is a 16-bit value that identifies the file or device that you want to read from or write to. You can create a file handle by using the **DosOpen** function, which opens the specified file and returns a file handle for it. For example, in the following code fragment, **DosOpen** opens the existing file *simple.txt* for reading and copies the file handle to the hf variable:

```
HFILE hf;
USHORT usAction;

DosOpen("simple.txt",     /* name of file to open    */
    &hf,                  /* address of file handle  */
    &usAction,            /* action taken            */
    0L,                   /* size of file            */
    FILE_NORMAL,          /* file attribute          */
    FILE_OPEN,            /* open the file           */
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE,
    0L);                  /* reserved                */
```

If the **DosOpen** function successfully opens the file, it copies the file handle to the hf variable and copies a value to the usAction variable indicating what action was taken (for example, FILE_EXISTED for "existing file opened"). A size is not needed to open an existing file, so the fourth argument is zero. The fifth argument, FILE_NORMAL, specifies the normal file attribute. The sixth argument, FILE_OPEN, directs **DosOpen** to open the file if it exists or to return an error if it does not exist. The seventh argument directs **DosOpen** to open the file for reading only and to let other programs open the file even while the current program has it open. The final parameter is reserved and should always be set to zero.

The **DosOpen** function returns zero if it successfully opened the file. You can then use the file handle in subsequent functions to read data from the file or to check the status or other characteristics of the file. If **DosOpen** fails to open the file, it returns an error value.

As shown in the preceding example, when you open a file you must specify whether you want to read from the file, write to it, or both read and write. You must also specify whether you want other processes to have access to the file while you have it open. You do this by combining an OPEN_ACCESS value and an OPEN_SHARE value from the following list:

Value	Meaning
OPEN_ACCESS_READONLY	Open a file for reading.
OPEN_ACCESS_WRITEONLY	Open a file for writing.
OPEN_ACCESS_READWRITE	Open a file for reading and writing.
OPEN_SHARE_DENYREADWRITE	Open a file for exclusive use, denying read and write access by other processes.
OPEN_SHARE_DENYWRITE	Deny write access to a file by other processes.
OPEN_SHARE_DENYREAD	Deny read access to a file by other processes.
OPEN_SHARE_DENYNONE	Open a file with no sharing restrictions, granting read and write access to all processes.

In general, you can combine any access method (read, write, or read and write) with any sharing method (deny reading, deny writing, deny reading and writing, or grant any access). Some combinations have to be handled carefully, however, such as opening a file for writing without denying access to it by other processes.

46.3.2 Creating Files

You can also create new files by using the **DosOpen** function. Once you have created a new file, you can read from or write to it just as you would with an existing file.

To create a new file, specify FILE_CREATE as the sixth argument. The **DosOpen** function then creates the file if it does not already exist. In the following code fragment, the **DosOpen** function creates the file *newfile.txt*:

```
HFILE hf;
USHORT usAction;

DosOpen("newfile.txt",       /* name of file to create and open */
    &hf,                     /* address of file handle          */
    &usAction,               /* action taken                    */
    0L,                      /* size of new file                */
    FILE_NORMAL,             /* file attribute                  */
    FILE_CREATE,             /* create the file                 */
    OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYNONE,
    0L);                     /* reserved                        */
```

In this example, **DosOpen** creates the file and opens it for writing only. Note that the sharing method allows other processes to open the file for any access. The new file is empty (contains no data).

When you use **DosOpen** to create a new file, you must specify the file attribute. In the preceding example, this value is FILE_NORMAL, so the file is created as a normal file. Other possible file attributes include read-only and hidden, which correspond to FILE_READONLY and FILE_HIDDEN, respectively.

The file attribute affects how other processes access the file. For example, if the file is read-only, no process can open the file for writing. The one exception to this rule is that the process that creates the read-only file can write to it immediately after creating it. After closing the file, however, the process cannot open it for writing again.

You must also specify the original size of the new file. For example, if you specify 256, the new file is 256 bytes long. However, these 256 bytes are undefined. It is up to the program to write valid data to the file. In any case, no matter what size you specify, subsequent calls to the **DosWrite** function copy data to the beginning of the file.

46.3.3 Reading from and Writing to Files

Once you have opened a file or have a file handle, you can read from and write to the file by using the **DosRead** and **DosWrite** functions. The **DosRead** function copies a specified number of bytes (up to the end of the file) from the file to the buffer you specify. The **DosWrite** function copies bytes from a buffer to the file.

To read from a file, you must open it for reading or for reading and writing. The following code fragment shows how to open the file named *sample.txt* and read the first 512 bytes from it:

```
HFILE hf;
USHORT usAction, usError;
BYTE abBuffer[512];
USHORT cbRead;
```

```
usError = DosOpen("sample.txt", &hf, &usAction, OL, FILE_NORMAL,
    FILE_OPEN, OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, OL);
if (!usError) {
    DosRead(hf, abBuffer, 512, &cbRead);
    DosClose(hf);
}
```

If the file does not have 512 bytes, **DosRead** reads to the end of the file and copies the number of bytes read to the cbRead variable. If the file pointer is already positioned at the end of the file when **DosRead** is called, the function copies zero to the cbRead variable.

To write to a file, you must first open it for writing or for reading and writing. The following code fragment shows how to open the file *sample.txt* again and write 512 bytes to it:

```
HFILE hf;
USHORT usAction;
BYTE abBuffer[512];
USHORT cbWritten, usError;

usError = DosOpen("sample.txt", &hf, &usAction, OL, FILE_NORMAL,
    FILE_CREATE, OPEN_ACCESS_WRITEONLY | OPEN_SHARE_DENYWRITE, OL);
if (!usError) {
    DosWrite(hf, abBuffer, 512, &cbWritten);
    DosClose(hf);
}
```

The **DosWrite** function writes the contents of the buffer to the file. If it fails to write 512 bytes (for example, if the disk is full), the function copies the number of bytes written to the cbWritten variable.

46.3.4 Reading and Writing Asynchronously

The **DosRead** and **DosWrite** functions are synchronous input and output functions, since they carry out their reading and writing operations before returning control to the program. By using the asynchronous input and output functions, **DosReadAsync** and **DosWriteAsync**, your program can continue with other tasks while the function reads from or writes to a file in a separate operation. Asynchronous input and output functions minimize the effect of input and output on the speed of your applications.

46.3.5 Closing a File

You can a close a file by using the **DosClose** function. Since each program has a limited number of file handles that it can have open at any given time, it is a good practice to close a file after using it. To do so, supply the file handle in the **DosClose** function, as shown in the following code fragment:

```
HFILE hf;
USHORT usAction;
BYTE abBuffer[80];
USHORT cbRead;

usError = DosOpen("sample.txt", &hf, &usAction, OL, FILE_NORMAL,
    FILE_OPEN, OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, OL);
if (!usError) {
    DosRead(hf, abBuffer, 80, &cbRead);
    DosClose(hf);
}
```

If you have opened a file for writing, the **DosClose** function directs the system to flush the file buffer—that is, to write any existing data in the intermediate file buffer to the disk or device. The system keeps these intermediate file buffers to make file input and output more efficient. For example, it saves data from previous calls to the **DosWrite** function until a certain number of bytes are in the buffer. It then writes the contents of the buffer to the disk.

46.3.6 Using the Standard Files

Every program, when first starting, has three input and output files available for its use. These files, called the standard-input, standard-output, and standard-error files, let the program read input from the keyboard and display output on the screen without opening or preparing the keyboard or screen.

As the system starts a program, it automatically opens the three standard files and makes the handles of the files—numbered 0, 1, and 2—available to the program. You can then read from and write to the standard files as soon as your program starts.

File handle 0 is the standard-input file. This handle lets you read characters from the keyboard by using the **DosRead** function. The function reads the specified number of characters unless the user types a turnaround character—that is, a character that marks the end of a line. (The default turnaround character is a carriage-return/newline character pair.) As **DosRead** reads the characters, it copies them to the buffer you have supplied, as shown in the following code fragment:

```
BYTE abBuffer[80];
USHORT cbRead;

DosRead(0, abBuffer, 80, &cbRead);
```

In this example, **DosRead** copies the number of characters read from standard input to the cbRead variable. The function also copies the turnaround character, or characters, to the buffer. If the function reads fewer than 80 characters, the turnaround character is the last one in the buffer.

File handle 1 is the standard-output file. This handle lets you write characters on the screen by using the **DosWrite** function. The function writes the characters in the given buffer or string to the current line. If you want to start a new line, you must insert the current turnaround character in the buffer. The following code fragment displays a prompt, reads a string, and displays the string:

```
USHORT cbWritten;
USHORT cbRead;
BYTE abBuffer[80];

DosWrite(1, "Enter a name: ", 14, &cbWritten);
DosRead(0, abBuffer, 80, &cbRead);
DosWrite(1, abBuffer, cbRead, &cbWritten);
```

File handle 2 is the standard-error file. This handle also lets you write characters on the screen. Most programs use the standard-error file to display error messages, since the user can then redirect standard output to a file without also redirecting error messages to the file.

46.3.7 Redirecting the Standard Files

Although the standard-input, standard-output, and standard-error files are usually the keyboard and screen, this is not always the case. For example, if the user redirects standard output by using the greater-than (>) redirection symbol on the program command line, all data written to the standard-output file goes to the given file. The following command line redirects standard output to the file *sample.txt* and redirects error messages to the file *sample.err*:

```
type startup.cmd >sample.txt 2>sample.err
```

When a standard file is redirected, its handle is still available but corresponds to the given disk file instead of to the keyboard or screen. You can still use the **DosRead** and **DosWrite** functions to read from and write to the files.

You can redirect a standard file from within a program by closing the file and then immediately reopening another file to be used for input or output. Whenever you open a file, MS OS/2 uses the lowest available handle for the new file. For example, if you close the standard-input file (handle 0) and immediately reopen some other file, that new file receives handle 0. Any subsequent calls to the **DosRead** function that specify handle 0 are read from the new file, not from the previous standard-input file.

Redirecting in this way is especially useful if you want to start a child process and have its standard input be a disk file rather than the keyboard. The following code fragment shows how to redirect the standard-input file from within a program:

```
DosClose(0);
DosOpen("sample.c", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
```

46.3.8 Using Wildcard Characters to Search for Files

You cannot use the wildcard characters (* and ?) in filenames that you supply to the **DosOpen** function, but you can locate files with names that match a given pattern by using wildcard characters in the **DosFindFirst** and **DosFindNext** functions.

The **DosFindFirst** function locates the first filename in the current directory that matches the given pattern. The **DosFindNext** function locates the next matching filename and continues to find additional matches on each subsequent call until all matching names are found. The functions copy the file statistics on each file located, such as name, attributes, and creation date, to a structure that you supply.

The following code fragment shows how to find all filenames that have the extension *.c*:

```
HDIR hdir;
USHORT usSearch;
FILEFINDBUF findbuf;

usSearch = 1;
hdir = HDIR_SYSTEM;
DosFindFirst("*.c",
    &hdir,              /* directory handle                      */
    FILE_NORMAL,        /* file attribute to look for            */
    &findbuf,           /* result buffer                         */
    sizeof(findbuf),    /* size of result buffer                 */
    &usSearch,          /* number of matching names to look for */
    0L);                /* reserved value                        */
```

```
do {
    .
    . /* use filename in findbuf.achName */
    .
    usSearch = 1;
    DosFindNext(hdir, &findbuf, sizeof(findbuf), &usSearch);
} while (usSearch != 0);
DosFindClose(hdir);
```

This example continues to retrieve matching filenames until the **DosFindNext** function returns zero in the usSearch variable. Before each call, the usSearch variable is set to the numeral in order to direct the function to look for only one matching name at a time.

To keep track of which files have already been found, both functions use the directory handle hdir to identify the current position in the directory. The directory handle also identifies for **DosFindNext** the name of the file being sought. This handle must be set to HDIR_SYSTEM or HDIR_CREATE before the **DosFindFirst** function is called, and the value returned by **DosFindFirst** must be used in subsequent calls to **DosFindNext**.

After locating the files you need, you should use the **DosFindClose** function to close the directory handle. This ensures that when you search for the same files again, you will start at the beginning of the file.

46.3.9 Moving the File Pointer

Every disk file has a corresponding file pointer that marks the current location in the file. The current location is the byte in the file that will be read from or written to on the next call to the **DosRead** or **DosWrite** function. Usually, the file pointer is at the beginning of the file when you first open or create the file and advances one byte at a time as you read a byte from or write a byte to the file. You can, however, change the position of the file pointer at any time by using the **DosChgFilePtr** function.

The **DosChgFilePtr** function moves the file pointer a specified offset from a given position. You can move the pointer from the beginning of the file, from the end, or from the current position. The following code fragment shows how to move the pointer 256 bytes from the end of the file:

```
HFILE hf;
USHORT usAction;
ULONG ulActual;

DosOpen("sample.txt", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READONLY | OPEN_SHARE_DENYNONE, 0L);
DosChgFilePtr(hf, -256L, FILE_END, &ulActual);
```

In this example, **DosChgFilePtr** moves the file pointer to the 256th byte from the end of the file (toward the beginning). If the file is not that long, the function moves the pointer to the first byte in the file and returns the actual position (relative to the end of the file) in the ulActual variable.

You can move the file pointer only for disk files. You cannot use **DosChgFilePtr** to change the file pointer's position on the screen, nor can you use it to read ahead from the keyboard.

46.3.10 Accessing Devices

You can open a number of devices by using the **DosOpen** function. A device is a piece of hardware, other than a disk drive, that is intended to be used for input and output. For example, the keyboard and screen are devices, as are any serial or parallel ports that your computer may have.

MS OS/2 lets you open and access a device just as you would open a disk file. However, what you read from or write to a device depends on the device. (This is not true for disk files.) For example, if you open a serial port that has a printer connected to it, you will need to know the input format of the printer. Writing plain text to the printer may or may not give you the result that you want.

The device may also behave differently depending on what driver you have installed to support it. For example, if you write to the system console, each byte is interpreted as a character and is subsequently displayed on the screen. If, however, you load the ANSI display driver when you start the system, some byte sequences may represent actions to take, such as moving the cursor.

To open a device by using the **DosOpen** function, you must supply the special reserved name for that device. For example, to open the console (both keyboard and screen), you must specify the name *con*. The following is a list of the reserved device names commonly used in programs:

Device name	Description
con	System console. This device consists of both the keyboard and the screen. You can open *con* for reading (from the keyboard), writing (to the screen), or both reading and writing.
com1	Serial port 1. This device is the first serial port in your computer. You can open it for reading, writing, or both reading and writing.
prn	Default printer port. This device corresponds to one of the system parallel ports. You can open it for writing but not for reading.
lpt1	Parallel port 1. This device represents a parallel port.
nul	Null device. This device provides a method of discarding output. If you open this device for writing, any data written to the file is discarded. If you open the device for reading, any attempt to read from the file returns an end-of-file mark.
screen$	System screen. This device is the system screen. It can be written to but not read from. Writing to the screen is similar to writing to the system console. Bytes are displayed as characters unless they represent an escape sequence. Escape sequences direct the screen to carry out actions, such as moving the cursor.
kbd$	Keyboard. This device is the keyboard. It can be read from but not written to. Reading from the keyboard is similar to reading from the console.

The following code fragment opens serial port 1 for reading and writing:

```
DosOpen("com1", &hf, &usAction, 0L, FILE_NORMAL, FILE_OPEN,
    OPEN_ACCESS_READWRITE | OPEN_SHARE_DENYNONE, 0L);
```

You can open some devices only if the appropriate driver is already available. For example, you cannot open a serial port unless a communications driver, such as *com01.sys*, has been loaded by using a **device** command in the system-configuration file, *config.sys*.

Once you have opened a device, you can use the **DosRead** and **DosWrite** functions to read from and write to the device.

After using the device, you should close it by using the **DosClose** function.

46.3.11 Controlling Input and Output Devices

Many devices have more than one mode of operation. For example, a serial port typically has a variety of baud rates at which it can operate. Since these modes of operation are unique to the device (that is, they differ from device to device), MS OS/2 does not include specific functions to set or inquire about these modes. Instead, it provides the **DosDevIOCtl** function, which controls device input and output. You can use **DosDevIOCtl** to set and retrieve information about the devices in your system.

For example, you can use the ASYNC_SETBAUDRATE control function (0x0001, 0x0041) to set the baud rate of serial port 1. The following code fragment sets the baud rate to 9600:

```
USHORT usBaudRate;
usBaudRate = 9600
DosDevIOCtl(&usBaudRate, 0L, 0x0041, 0x0001, hf);
```

46.4 Summary

MS OS/2 provides the following file-system functions:

DosBufReset Clears the file buffers.

DosChDir Changes the current directory.

DosChgFilePtr Moves the file pointer a specified number of bytes in a file.

DosClose Closes a file.

DosDelete Deletes a file.

DosDevIOCtl Passes device-control functions to the specified device.

DosDupHandle Duplicates a file handle. The original handle and the duplicate handle are interchangeable.

DosFileLocks Locks or unlocks a portion of a file.

DosFindClose Closes a search directory.

DosFindFirst Searches a directory for a file whose name and attributes match the specified name and attributes.

DosFindNext Continues the search begun by using the **DosFindFirst** function.

DosMkDir Creates a directory.

DosMove Moves a file to a new directory and/or filename location.

DosNewSize Changes the size of a file.

DosOpen Opens or creates a file.

DosQCurDir Retrieves the path of the current directory on the specified drive.

DosQCurDisk Retrieves the number of the current drive.

DosQFHandState Retrieves the access and sharing status of a handle.

DosQFileInfo Retrieves information about a file and its use.

DosQFileMode Retrieves the attributes (mode) of a file.

DosQFSInfo Retrieves information about the file system on the disk in the specified drive.

DosQHandType Retrieves the type of a specified handle—that is, whether the handle identifies a file, a device, or a pipe.

DosQVerify Retrieves the current write-verification mode.

DosRead Reads data from a file or device.

DosReadAsync Reads data from a file or device while the calling process continues with other tasks.

DosRmDir Removes a directory.

DosSelectDisk Selects a default disk drive.

DosSetFHandState Sets a handle's inheritance, fail-on-error, and write-through flags.

DosSetFileInfo Modifies date-and-time information for a file.

DosSetFileMode Sets the attributes (mode) of a file.

DosSetFSInfo Sets the volume label for the disk in the specified drive.

DosSetMaxFH Sets the maximum number of file handles for the current process.

DosSetVerify Enables or disables write verification.

DosWrite Writes data to a file or device.

DosWriteAsync Writes data to a file or device while the calling process continues with other tasks.

Chapter

47

Video Input and Output

47.1 Introduction 639
47.2 About Video Input and Output 639
47.3 Using Video Input and Output 640
47.3.1 Displaying a Character 640
47.3.2 Writing a Character to a Specific Location 640
47.3.3 Writing a String of Characters to the Screen 641
47.3.4 Writing Character Cells to the Screen 641
47.3.5 Moving and Hiding the Cursor 642
47.3.6 Reading Characters from the Screen 643
47.3.7 Scrolling the Screen Contents 643
47.3.8 Using the ANSI Display Mode 643
47.4 Summary 644

47.1 Introduction

This chapter describes the video-input-and-output (video I/O) functions. These functions give programs direct access to the system display. You should also be familiar with the following topics:

- Character-based MS OS/2 programs
- Displays and display adapters
- The keyboard
- The mouse
- Advanced video input and output (AVIO)

47.2 About Video Input and Output

A program can write individual characters and strings to the screen, either at the current cursor position by using the **VioWrtTTY** function or at a specified position by using the **VioWrtNChar** or **VioWrtCharStr** function. Strings can consist of characters, attributes, or character-and-attribute pairs.

A character is actually specified by a character value. The screen uses the value to locate a character bitmap in the current screen font and then displays the bitmap at the specified location. For some displays, a program can change the current screen font by using the **VioSetFont** function. To do this, the program typically first retrieves the screen font by using the **VioGetFont** function and then modifies that font before setting it as the new font. The format of the font bitmap depends on the display.

A program can write character attributes to the screen by using the **VioWrtNAttr** or **VioWrtCharStrAtt** function. Attributes define the color and appearance of the characters at the corresponding position. The effect of a given attribute depends on the display. For a list of attribute values and meanings for some common displays, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

A program can also combine character and attribute values into a single value, called a cell or a character-and-attribute pair, and write one or more cells to the screen at the specified position by using the **VioWrtNCell** or **VioWrtCellStr** function.

A program can set the position of the cursor by using the **VioSetCurPos** function or retrieve the current position of the cursor by using the **VioGetCurPos** function. The cursor position, like any position on the screen, is specified in screen coordinates. Screen coordinates are relative to the upper-left corner of the screen. The *x*-axis values increase to the right; the *y*-axis values increase downward. The screen units are either character cells or pixels, depending on the screen mode (character cells for text mode, pixels for graphics mode). The position never exceeds the width or height of the screen. A program can also set the cursor type, which defines the width and height of the cursor as well as its color and appearance, by using the **VioSetCurType** function.

A program can also set or retrieve the screen mode by using the **VioSetMode** or **VioGetMode** function, respectively. Which modes can be set depends entirely on the display. For more information, see the *Microsoft Operating System/2 Programmer's Reference, Volume 2.*

A program can scroll the screen contents left, right, up, or down by using the **VioScrollLf, VioScrollRt, VioScrollUp,** and **VioScrollDn** functions. The scroll functions move the specified rectangle to the given location on the screen, filling any area uncovered by the rectangle with the given character and attribute.

Other screen functions carry out such special tasks as adapting the screen for country-specific or ANSI information, modifying the operation of one or more of the screen functions, and accessing the video buffers directly.

47.3 Using Video Input and Output

Since output for Presentation Manager applications is provided by using windows and the **Gpi** functions, only non-Presentation Manager programs use the MS OS/2 **Vio** functions. Typically, you use these functions in character-based programs. The following sections explain how to use some of the **Vio** functions.

47.3.1 Displaying a Character

The **KbdCharIn** function does not echo a keystroke as it reads it; that is, the function does not write the corresponding character to the screen. If you want to display the character, you can do so by using the **VioWrtTTY** function. The following code fragment writes the letter *A* to the screen:

```
CHAR ch = 'A';

VioWrtTTY(&ch, 1, 0);
```

Video functions, like keyboard functions, require a handle to identify the screen to be accessed. In the preceding example, the third argument, 0, is the default video handle and identifies the system screen. The system screen is always available to programs and does not need to be opened to be used. Video handle 0 and standard-output handle 1 are not the same. The standard-output file can be closed or redefined, but the system-screen handle cannot.

If the standard-output handle has not been redirected, you can also write a character to the screen by using the **DosWrite** function. **DosWrite** calls the **VioWrtTTY** function to write the character, so when writing to the screen, these functions are identical.

The **VioWrtTTY** function always writes the character to the current position of the cursor and then advances the cursor one position. If the cursor reaches the end of a line, it wraps to the beginning of the next line. If the cursor reaches the end of the screen, **VioWrtTTY** scrolls the screen contents up one line.

47.3.2 Writing a Character to a Specific Location

You can write a character to a specific location on the screen by using the **VioWrtNChar** function. This function lets you specify the row and column at which to place the character. The following code fragment writes the letter *A* to row 10, column 15:

```
CHAR ch = 'A';

VioWrtNChar(&ch, 1, 10, 15, 0);
```

The **VioWrtNChar** function uses the coordinate system of the screen to determine where to place the character. Such a coordinate system typically divides the screen into rows and columns. Each coordinate represents a character cell, a rectangular area of the screen large enough to display one character. When you write a character to a particular coordinate, it overwrites the character that was previously there.

In MS OS/2, the upper-left corner of the screen is at position (0,0) and, for most screen modes, the lower-right corner is at position (24,79). If you attempt to write a character outside these boundaries, the function returns an error.

You can write the same character to the screen repeatedly by specifying a value greater than 1 as the second argument. For example, the following code fragment clears a 25 × 80 screen:

```
CHAR ch = ' ';

VioWrtNChar(&ch, 2000, 0, 0, 0);
```

If the **VioWrtNChar** function reaches the end of a line, it automatically wraps to the beginning of the next line. However, it does not scroll the screen contents when it reaches the bottom of the screen, and it does not update the cursor position.

You can write a single attribute to the screen a specified number of times by using the **VioWrtNAttr** function.

47.3.3 Writing a String of Characters to the Screen

You can write a string of characters directly to the screen by using the **VioWrtCharStr** function. When you write characters to the screen in this way, you can specify the row and column at which the characters are to start. This function is similar to the **VioWrtNChar** function, except that instead of a single character, you specify an array of characters. The following code fragment writes the string "Hello, world" to the middle of the screen:

```
VioWrtCharStr("Hello, world",  12, 13, 34, 0 );
```

You can write a string of characters to the screen with a specific attribute by using the **VioWrtCharStrAtt** function.

47.3.4 Writing Character Cells to the Screen

You can write character cells to the screen by using the **VioWrtNCell** or **VioWrtCellStr** function. A character cell is a 2-byte value that specifies a character and its attribute. A character attribute defines the color, intensity, and appearance of the character to be written. The following code fragment writes a red letter *A* to the middle of a color screen:

```
BYTE abCell[2] = { 'A', 0x04 };

VioWrtNCell(abCell, 1, 13, 40, 0);
```

Character cells are useful in programs that take full advantage of the text-mode capabilities of the display adapter. However, the meaning and range of values for attributes depend on the device, so it is important that you check the display-adapter type. You can do this by using the **VioGetConfig** function, as shown in the following code fragment:

```
VIOCONFIGINFO vioin;
vioin.cb = sizeof(vioin);

VioGetConfig(0, &vioin, 0);
switch (vioin.adapter) {
    case 0: /* monochrome adapter             */
        break;
    case 1: /* color graphics adapter (CGA)    */
        break;
    case 2: /* enhanced graphics adapter (EGA) */
        break;
}
```

47.3.5 Moving and Hiding the Cursor

If you choose to use the **VioWrtTTY** or **DosWrite** function to write text to the screen, you can control the placement of that text on the screen by using the **VioSetCurPos** and **VioGetCurPos** functions to set and get the position of the cursor. The cursor is the flashing underscore or block on the screen that marks the location that will receive the next character written to the screen. The following code fragment moves the cursor to the middle of the screen and writes the string "Hello, world":

```
VioSetCurPos(13, 34, 0);
VioWrtTTY("Hello, world", 12, 0);
```

If you choose not to use the cursor, you can remove it from the screen by using the **VioSetCurType** function, which requires a **VIOCURSORINFO** structure. If you set the **attr** field in the **VIOCURSORINFO** structure to 0xFFFF, as shown in the following code fragment, the function hides the cursor:

```
VIOCURSORINFO vioci;

VioGetCurType(&vioci, 0);     /* retrieve current cursor type */
vioci.attr = 0xFFFF;          /* hide the cursor              */
VioSetCurType(&vioci, 0);     /* set new cursor type          */
```

You can restore the cursor by setting the **attr** field to its original value. In the preceding example, the **VioGetCurType** function was used to fill the **VIOCURSORINFO** structure with the current information before the structure was modified to hide the cursor. In general, whenever you use an MS OS/2 function that takes values from a structure, you should be sure that all fields contain valid values. In this case, the way to ensure valid values is to fill the structure first by using the **VioGetCurType** function.

You can also use the **VioSetCurType** function to change the shape of the cursor. For example, you can change the shape from an underscore to a block by setting the **yStart** and **cEnd** fields to appropriate values. The following code fragment creates a block cursor:

```
VioGetCurType(&vioci, 0);     /* retrieve current cursor type */
vioci.yStart = 10;            /* start of cursor              */
vioci.cEnd = 0;               /* end of cursor                */
VioSetCurType(&vioci, 0);     /* set new cursor type          */
```

47.3.6 Reading Characters from the Screen

You can read characters from the screen by using the **VioReadCellStr** or **VioReadCharStr** function. Reading characters is an easy way to determine the content of the screen. Some programs use this as a method of input. For example, a "help" program might check the location of the cursor and read the line at that point to provide context-sensitive help.

The following code fragment reads a character string at the current cursor position:

```
CHAR achBuffer[80];
USHORT cchBuffer = 80;
USHORT usRow, usCol;

VioGetCurPos(&usRow, &usCol, 0);
VioReadCharStr(achBuffer, &cchBuffer, usRow, usCol, 0 );
```

47.3.7 Scrolling the Screen Contents

You can scroll all or part of the screen contents by using the **VioScrollDn**, **VioScrollUp**, **VioScrollLf**, and **VioScrollRt** functions.

The following code fragment scrolls the screen contents up three lines, leaving three blank lines at the bottom of the screen:

```
BYTE abCell[2] = { ' ', 0x07 };

VioScrollUp(3, 0, 24, 79, 3, abCell,  0);
```

You can also use the scroll functions to clear the screen. Whenever the rectangle that you specify has the same dimensions as the screen, the entire screen is cleared. The following code fragment clears a 25 × 80 screen:

```
BYTE abCell[2] = { ' ', 0x07 };

VioScrollUp(0, 0, 24, 79, 0xFFFF, abCell, 0);
```

47.3.8 Using the ANSI Display Mode

You can set the video display to ANSI mode by using the **VioSetAnsi** function. When in ANSI mode, the video display checks for and carries out the actions specified by any ANSI escape sequences that are written to the screen by such functions as **VioWrtTTY**.

An ANSI escape sequence is a combination of characters, starting with the escape character (27), that specifies a particular action to be taken by the video display, such as moving the cursor or displaying subsequent characters in a new display mode. For a complete listing of the available ANSI escape sequences, see the *Microsoft Operating System/2 Programmer's Reference, Volume 3.*

47.4 Summary

MS OS/2 provides the following video-input-and-output functions:

VioDeRegister Restores the default **Vio** subsystem and removes any previously registered **Vio** subsystem.

VioEndPopUp Closes a pop-up screen.

VioGetAnsi Retrieves the state of the ANSI flag.

VioGetBuf Retrieves the address of the logical video buffer.

VioGetConfig Retrieves the video-display configuration.

VioGetCp Retrieves the identifier of the code page for the current screen group.

VioGetCurPos Retrieves the cursor position.

VioGetCurType Retrieves the cursor type.

VioGetFont Retrieves the specified screen font.

VioGetMode Retrieves the current screen mode.

VioGetPhysBuf Retrieves the address of the physical video buffer.

VioGetState Retrieves the current settings of the palette register, the overscan (border) color, and the blink/background intensity switch.

VioModeUndo Cancels a request to be notified of a change of video mode.

VioModeWait Waits for a change of video mode.

VioPopUp Opens a pop-up screen.

VioPrtSc Copies the contents of the screen to the printer. This function is reserved for system use.

VioPrtScToggle Enables or disables the printer-echo feature. This function is reserved for system use.

VioReadCellStr Reads one or more character-and-attribute pairs (cells) from the screen.

VioReadCharStr Reads a character string from the screen.

VioRegister Registers a **Vio** subsystem.

VioSavRedrawUndo Cancels a request to be notified of a switch of screen group.

VioSavRedrawWait Waits for a switch of screen group.

VioScrLock Locks the physical video buffer.

VioScrollDn Scrolls the contents of the screen downward.

VioScrollLf Scrolls the contents of the screen to the left.

VioScrollRt Scrolls the contents of the screen to the right.

VioScrollUp Scrolls the contents of the screen upward.

VioScrUnLock Unlocks the physical video buffer.

VioSetAnsi Enables or disables the process of ANSI escape sequences.

VioSetCp Sets the code page for the current screen group.

VioSetCurPos Sets the cursor position.

VioSetCurType Sets the cursor type.

VioSetFont Sets the font used to display characters on the screen.

VioSetMode Sets the screen mode.

VioSetState Sets the palette registers, the overscan (border) color, and the blink/background intensity switch.

VioShowBuf Updates the physical screen from the logical video buffer.

VioWrtCellStr Writes one or more character-and-attribute pairs (cells) to the screen.

VioWrtCharStr Writes a character string to the screen.

VioWrtCharStrAtt Writes a character string to the screen, using the specified attribute.

VioWrtNAttr Writes a character attribute to the screen a specified number of times.

VioWrtNCell Writes a character-and-attribute pair (cell) to the screen a specified number of times.

VioWrtNChar Writes a character to the screen a specified number of times.

VioWrtTTY Writes a character string to the screen, starting at the current cursor position.

Chapter

48

Advanced Video Input and Output

48.1 Introduction 649

48.2 About Advanced Video Input and Output 649
48.2.1 AVIO Presentation Spaces 649
48.2.2 Character Formats 650
48.2.3 Code Pages 652
48.2.4 Device Cell Size 652
48.2.5 WM_SIZE Message Processing 653

48.3 Using Advanced Video Input and Output 653

48.4 Summary 654
48.4.1 Standard Vio Functions 654
48.4.2 Base Vio Functions Not Supported 656
48.4.3 Advanced Vio Functions 656

48.1 Introduction

This chapter describes the portions of MS OS/2 that let you create advanced video-input-and-output (AVIO) programs. You should also be familiar with the following topics:

- Windows
- Presentation spaces and device contexts
- Video input and output

48.2 About Advanced Video Input and Output

Advanced video input and output is a set of MS OS/2 system functions and features that allows you to use MS OS/2 video-input-and-output (**Vio**) functions in a Presentation Manager application. An AVIO presentation space is similar to a graphics programming interface (GPI) presentation space except that the **Vio** functions can be used to display text. The **Gpi** functions for graphics are also available.

An AVIO program is any program that creates and uses an AVIO presentation space. AVIO programs are similar to Presentation Manager applications in that they create a message queue, use a message loop, and create windows. They differ from Presentation Manager applications in that they depend largely on the **Vio** functions to display output. Many AVIO programs are a hybrid of character-based programs and Presentation Manager applications.

48.2.1 AVIO Presentation Spaces

You create an AVIO presentation space by using the **VioCreatePS** function. This function returns a handle to the AVIO presentation space. As you would with a **Gpi** presentation space, you must associate the AVIO handle with a device context—in this case, the device context of the program's window. The following code fragment creates an AVIO presentation space and associates its handle with the device context of a window:

```
HVPS hvps;
HDC  hdc;
HWND hwnd;

VioCreatePS(&hvps, 25, 80, 0, FORMAT_CGA, 0);
hdc = WinOpenWindowDC(hwnd);
VioAssociate(hdc, hvps);
```

The presentation space consists of a character-cell grid that is similar to the video-input-and-output grid used with full-screen programs. The dimensions of the grid (the width and height) are measured in character cells and are set by using the **VioCreatePS** function. Each character has a corresponding character cell. The width and height of the character cell is dependent on the given font, but a program can usually switch between the available sizes by using the **VioSetDeviceCellSize** function.

The program sets the size of the AVIO presentation space when it creates it. If the presentation space is smaller than the window in either direction, the excess area is left unchanged; if the presentation space is larger, the excess data is not displayed. The size of the AVIO presentation space is typically smaller than the maximized window. Programs should restrict the size of their maximized window to the size of the AVIO presentation space.

You can set the origin (that is, data to be displayed in the top left of the window) by using the **VioSetOrg** function. If an area of the window is "exposed" by the origin being altered such that the right or bottom edge of the presentation space moves leftward or upward in the window, then the "exposed" area is cleared. This would also remove any graphics present in the area.

After directly writing to an AVIO presentation space, the program can update the screen by using the **VioShowBuf** or **VioShowPS** function. The **VioShowPS** function updates only within a specific rectangle, so it may improve performance by reducing the amount of drawing the system has to do for the update.

Although the AVIO presentation space cannot be used with **Gpi** functions, any **Gpi** presentation space associated with the same window can be used to draw graphics along with the **Vio** text.

48.2.2 Character Formats

Each character to be displayed is identified by a code point and one or more attributes. A code point is a unique number in the range 0 through 255 that identifies the character. An attribute is a value that specifies how the character is to be displayed. For example, an attribute may specify the color of the image, the color of the character cell not containing the image, whether the image has a high-intensity color, an underline, and so on.

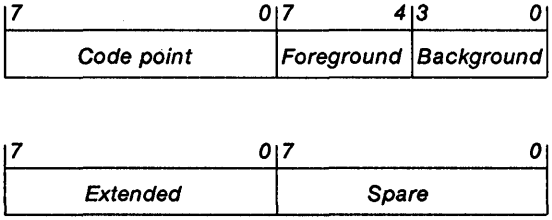
The number of attributes required to specify a character depends on the type of presentation space. In advanced video input and output, there are two types of presentation space: FORMAT_CGA and FORMAT_4BYTE. In a presentation space of the type FORMAT_CGA, each character consists of a code point, a foreground color, and a background color. The foreground and background colors are 4-bit fields that are combined to form an 8-bit attribute byte. Figure 48.1 shows the FORMAT_CGA character format:

Figure 48.1
2-Byte Character Format

7 0	7 4	3 0
Code point	*Foreground*	*Background*

In a presentation space of the type FORMAT_4BYTE, each character consists of a code point, foreground and background colors, an intensity flag, an underline flag, a reverse-video flag, a transparency flag, and a font identifier. There is also an additional 8-bit field (spare) for program-specific attribute data. Figure 48.2 shows the FORMAT_4BYTE character format:

Figure 48.2
4-Byte Character Format



If one attribute byte is present, MS OS/2 assumes suitable defaults (for example, no highlighting, opaque, default font). Presentation Manager-display device drivers do not support the blink or intensify attributes.

The color attribute defines the background and foreground color of a character. The following list shows the bits that affect these colors:

Bit	Meaning
7–4	Specifies the background color. It can be any value in the range 0 through 7.
3–0	Specifies the foreground color. It can be any value in the range 0 through 7.

The colors that appear on the screen depend on the current physical-color palette. The program must ensure that the colors displayed are the colors expected.

The extended attribute provides more information about how the character is displayed, such as which font the character is drawn from, whether the character is underlined, displayed in reverse video, or has a transparent background. The following list specifies the bits that affect the extended attributes:

Bit	Meaning
7	Underscore
6	Reverse video
5	Reserved
4	Background transparency (1 for transparent, 0 for opaque)
3	Reserved
2	Reserved
1–0	Font identifier (0, 1, 2, or 3)

The font identifier specifies the font in which to display the character. A program can reference up to four fonts—the default font and three loadable fonts. The font to be used for a particular character is controlled by the font bits, where font 0 is the default font. The numbers 1, 2, and 3 correspond to local identifiers created by using the **VioCreateLogFont** function. Unpredictable results occur if a loadable font identifier is used in a presentation space before being defined for the presentation space.

The spare attribute is for use by the program.

All valid **Vio** functions may access all AVIO presentation spaces. The program must ensure that data is in the correct format for a particular function. For example, a character cell for a FORMAT_4BYTE presentation space must be four bytes long.

48.2.3 Code Pages

MS OS/2 uses a code page to determine which character image to display on the screen for a given code point. A code page is a table of character values. Each character value represents the index to a specific character image in a given font. To retrieve a character image from a font for a given code point, MS OS/2 uses the code point to select a character value from the code page, then uses the character value to select the image from the font. Each code point corresponds to one character value in a code page.

When an AVIO presentation space is created, MS OS/2 assigns a default code page to it. MS OS/2 uses this default code page as its primary code page. Typically, the system's primary code page is set during system initialization, but any program can change the code page by using the **DosSetCp** function. The **DosGetCp** function retrieves the identifier, a unique integer, of the current primary code page.

48.2.4 Device Cell Size

MS OS/2 allows a program to use any cell size that a particular device supports. A program sets the cell size by using the **VioSetDeviceCellSize** function. The device cell size is specified in pels. A given device has a default device cell size, so if a program does not set the size, the default is used.

If a program requests a cell size that the device does not support, the system chooses a size that best fits the required size. The program can check the new size by using the **VioGetDeviceCellSize** function. In general, the actual size will be less than the requested size if there is not an exact match.

An AVIO program can also vary the device cell size by using the **VioSetMode** function to alter the number of character rows. The number of rows requested must be supported by the particular device. When the device cell size is altered, the system switches to the default (*lcid* 0) font corresponding to the cell size. The program must ensure that the loaded fonts are valid. If a mismatch exists between the cell size specified in the system and the loaded font, all the characters in that font are drawn as blanks.

To retrieve a list of available fonts that can be used in the presentation space, you use the **VioQueryFonts** function. The **VioQuerySetIds** function retrieves a list that describes the default font plus all loaded fonts for an AVIO presentation space.

You use the **DevQueryCaps** function to find out whether windows used for advanced video input and output on a particular device must be character-aligned; whether they should be character-aligned for best performance; or whether they may be either character- or pel-aligned, with no significant impact on performance.

48.2.5 WM_SIZE Message Processing

An AVIO program must pass the WM_SIZE message to the **WinDefAVioWindowProc** function. This function updates and maintains size information stored for the AVIO presentation space. Programs should call **WinDefAVioWindowProc** from the window procedure for each WM_SIZE message.

48.3 Using Advanced Video Input and Output

This section presents the source code for a simple AVIO program. The program creates the AVIO presentation space and associates it with the device context for the client window. The window procedure for the client window then uses the **VioWrtTTY** function to display a message.

```
#define INCL_WIN
#define INCL_GPI
#define INCL_VIO
#define INCL_AVIO
#include "os2.h"

HAB hab;            /* handle to the anchor block            */
HMQ hmq;            /* handle to the message queue           */
HWND hwndClient;    /* handle to the client                  */
HWND hwndFrame;     /* handle to the frame window            */
QMSG qmsg;          /* message-queue structure               */
HDC hdc;            /* handle to the device context          */
HVPS hvps;          /* handle to the AVIO presentation space */

MRESULT CALLBACK GenericWndProc(hwnd, msg, mp1, mp2)
HWND   hwnd;
USHORT msg;
MPARAM mp1;
MPARAM mp2;
{
    HPS   hps;
    RECTL rcl;
    SHORT x, y;
    SHORT cx, cy;

    switch (msg) {
        case WM_PAINT:
            hps = WinBeginPaint(hwnd, NULL, NULL);
            WinQueryWindowRect(hwnd, &rcl);
            VioGetDeviceCellSize(&cy, &cx, hvps);
            x = (rcl.xRight - rcl.xLeft) / (2 * cx);
            y = (rcl.yTop - rcl.yBottom) / (2 * cy);
            VioWrtCharStr("Hello World!", 12, y, x - 6, hvps);
            VioShowPS(25, 80, 0, hvps);
            WinEndPaint(hps);
            return (0L);

        case WM_SIZE:
            return (WinDefAVioWindowProc(hwnd, msg, mp1, mp2));
    }
    return (WinDefWindowProc(hwnd, msg, mp1, mp2));
}
```

```
ULONG flStyle = FCF_MINMAX | FCF_SYSMENU | FCF_TITLEBAR |
                FCF_SIZEBORDER | FCF_SHELLPOSITION | FCF_TASKLIST;

VOID cdecl main()
{
    hab = WinInitialize(NULL);
    hmq = WinCreateMsgQueue(hab, 0);

    if (!WinRegisterClass(hab, "MyClass",
            GenericWndProc, CS_SIZEREDRAW, 0))
        DosExit(EXIT_PROCESS, 1);

    hwndFrame = WinCreateStdWindow(HWND_DESKTOP, WS_VISIBLE, &flStyle,
        "MyClass", "My Title", 0L, NULL, 1, &hwndClient);

    hdc = WinOpenWindowDC(hwndClient);  /* opens device context      */
    VioCreatePS(&hvps, 25, 80, 0,       /* creates AVIO PS           */
        FORMAT_CGA, 0);
    VioAssociate(hdc, hvps);            /* associates DC and AVIO PS */

    while (WinGetMsg(hab, &qmsg, NULL, 0, 0))        /* message loop */
        WinDispatchMsg(hab, &qmsg);

    VioAssociate(NULL, hvps);           /* disassociates the AVIO PS */
    VioDestroyPS(hvps);                 /* destroys the AVIO PS      */
    WinDestroyWindow(hwndFrame);
    WinDestroyMsgQueue(hmq);
    WinTerminate(hab);
    DosExit(EXIT_PROCESS, 0);
}
```

48.4 Summary

This section lists the **Vio** functions that can and cannot be used with AVIO programs.

48.4.1 Standard Vio Functions

The standard **Vio** functions can be used in AVIO programs. All of the following functions must be passed a zero handle when called from a full-screen program and a nonzero **Vio** handle when called from an AVIO program:

VioEndPopUp Deallocates a pop-up display screen. An error occurs if you issue this function with a nonzero handle.

VioGetAnsi Retrieves the Ansi state.

VioGetBuf Retrieves the address and length of the AVIO presentation space. The presentation space may be used to directly manipulate displayed information.

VioGetConfig Retrieves the video configuration. Only the adapter type and display type are set.

VioGetCp Retrieves the code page currently used to display text on the screen.

VioGetCurPos Retrieves the current row and column position of the cursor.

VioGetCurType Retrieves the cursor type, which consists of the cursor start line, end line, width (assumed to be zero, which is one column width), and attribute (normal or hidden).

VioGetMode Retrieves the current mode of the video display. This mode is valid only for default video input and output. An error occurs if you issue this function with a nonzero handle.

VioPopUp Allocates a pop-up display screen. An error occurs if you issue this function with a nonzero handle.

VioPtrSc Prints the contents of the screen.

VioPtrScToggle Toggles the CTRL+PRTSC flag.

VioReadCellStr Reads a string of characters/attributes (or cells) from an AVIO presentation space, starting at the specified location.

VioReadCharStr Reads a character string from an AVIO presentation space, starting at the specified location.

VioScrollDn Scrolls the current screen down by the specified number of lines.

VioScrollLf Scrolls the current screen left by the specified number of columns.

VioScrollRt Scrolls the current screen right by the specified number of columns.

VioScrollUp Scrolls the current screen up by the specified number of lines.

VioSetAnsi Sets the ANSI state on or off.

VioSetCp Sets code page used to display text on the screen. You can use this function to set an EBCDIC code page for advanced video input and output.

VioSetCurPos Positions the cursor to the specified row and column on the display.

VioSetCurType Sets the cursor type, which consists of the cursor start line, end line, width (assumed to be zero, which is one column width), and attribute (normal or hidden).

VioSetMode Sets the mode of the video display. This mode is valid only for default video input and output. An error occurs if you issue this function with a nonzero handle. You can set text modes only, not graphics modes.

VioShowBuf Updates the display with the AVIO presentation space. You may specify a subset of the presentation space for update.

VioWrtCellStr Writes a character/attribute string to an AVIO presentation space. You must specify the starting location in the presentation space where the string is to be written.

VioWrtCharStr Writes a character string to an AVIO presentation space. You must specify the starting location in the presentation space where the string is to be written.

VioWrtCharStrAtt Writes a character string with a repeated attribute code to an AVIO presentation space. You must specify the starting location in the presentation space where the string is to be written.

VioWrtNAttr Writes an attribute code to an AVIO presentation space a specified number of times. You must specify the starting location in the presentation space where the attribute code is to be written.

VioWrtNCell Writes a cell (or character/attribute) to an AVIO presentation space a specified number of times. You must specify the starting location in the presentation space where the cell is to be written.

VioWrtNChar Writes a character to an AVIO presentation space a specified number of times. You must specify the starting location in the presentation space where the character is to be written.

VioWrtTTY Writes a character string from the current cursor position in TTY mode to an AVIO presentation space. Once it has written the string, the function positions the cursor one space after the end of the string.

48.4.2 Base Vio Functions Not Supported

The following functions must not be called by an AVIO program; otherwise, an error results:

VioDeRegister Deregisters a **Vio** subsystem.

VioGetFont Retrieves the current device font.

VioGetPhysBuf Retrieves the address of the physical video buffer.

VioGetState Retrieves the current setting of the video state.

VioModeUndo Cancels mode-change notification

VioModeWait Requests mode-change notification.

VioRegister Registers a **Vio** subsystem within a screen group.

VioSavRedrawWait Requests screen-switch notification.

VioSavRedrawUndo Cancels screen-switch notification.

VioScrLock Locks the screen.

VioScrUnLock Unlocks the screen.

VioSetFont Sets the display font.

VioSetState Sets the video state.

48.4.3 Advanced Vio Functions

The advanced **Vio** functions can be used in AVIO programs only. All of the following functions must be passed a nonzero **Vio** handle:

VioAssociate Associates an AVIO presentation space with a device context.

VioCreateLogFont Creates a logical font.

VioCreatePS Allocates an AVIO presentation space.

VioDeleteSetId Releases a font.

VioDestroyPS Destroys an AVIO presentation space.

VioGetDeviceCellSize Retrieves the current size of the device cell.

VioGetOrg Retrieves the origin of the presentation space.

VioQueryFonts Retrieves the available fonts.

VioQuerySetIds Queries which fonts are loaded into which font IDs. This call is so named for consistency with its **Gpi** equivalent.

VioSetDeviceCellSize Sets the size of the device cell.

VioSetOrg Sets the origin of the presentation space.

VioShowPS Updates the display with an AVIO presentation space for a rectangle.

Chapter

49

The Mouse

49.1 Introduction 661

49.2 About the Mouse 661

49.3 Using the Mouse 662

49.3.1 Opening the Mouse 662

49.3.2 Drawing the Mouse Pointer 663

49.3.3 Hiding the Mouse Pointer 663

49.3.4 Using the Event Queue 664

49.3.5 Specifying the Events to be Queued 665

49.4 Summary 665

49.1 Introduction

This chapter describes the mouse functions. These functions give programs direct access to the system mouse or other pointing device. Programs can read individual mouse events (mouse motions and button presses) and carry out actions based on those events. You should also be familiar with the following topics:

- The file system
- The keyboard
- Input and output control
- Shared resources

49.2 About the Mouse

In addition to using the keyboard for program input, you can use the mouse. To use the mouse, you must first open it by using the **MouOpen** function (MS OS/2 does not provide a default handle for the mouse as it does for the keyboard and screen) and then draw the mouse pointer by using the **MouDrawPtr** function. Once you have opened the mouse, you can read events from the mouse-event queue by using the **MouReadEventQue** function. An event defines the position of the mouse and the state of the mouse buttons. The system copies an event to the queue whenever the user moves the mouse or presses or releases a mouse button.

MS OS/2 specifies the position of the mouse either in screen coordinates or as the number of mickeys relative to the last position. Screen coordinates are relative to the upper-left corner of the screen. The *x*-axis values increase to the right; the *y*-axis values increase downward. The screen units are either character cells or pels, depending on the screen mode (character cells for text mode, pels for graphics mode). The position never exceeds the width or height of the screen.

A mickey is the mouse unit of motion. If the mouse position is specified in mickeys, it represents the direction and distance the mouse has moved from its last position. A program can determine how many centimeters the mouse has moved by using the **MouGetNumMickeys** function to retrieve the mickey-to-centimeter ratio. A position in mickeys is a signed value. If the *x*-coordinate is negative, the mouse moved to the left; if positive, it moved to the right. If the *y*-coordinate is negative, the mouse moved up; if positive, it moved down.

MS OS/2 reports only the mouse events that are defined in the mouse-event mask. A program can set or retrieve the event mask by using the **MouSetEventMask** or **MouGetEventMask** function, determine how many mouse events are in the queue by using the **MouGetNumQueEl** function, or flush any existing events from the queue by using the **MouFlushQue** function.

MS OS/2 cannot draw the mouse pointer unless a pointer driver has been specified in a **device** command in the *config.sys* file or a pointer-driver name is explicitly given when the mouse is opened. The pointer driver supplied with MS OS/2 is named *pointdd.sys*.

A program can change the shape of the mouse-pointer. Typically, the program first retrieves the current shape by using the **MouGetPtrShape** function and then modifies that shape before setting it by using the **MouSetPtrShape** function. The mouse-pointer shape consists of two masks: an AND mask and an XOR mask. MS OS/2 first combines the AND mask with the contents of the screen at the current position by using the bitwise AND operator. It then combines the result with the XOR mask by using the bitwise XOR operator. The format of the masks depends on the screen mode: character-and-attribute pairs for text mode and bitmaps for graphics mode.

If a program needs to temporarily hide the mouse pointer, the **MouRemovePtr** function removes the mouse pointer from all or a portion of the screen.

MS OS/2 automatically updates the mouse-pointer position as the mouse moves, but only if the amount of motion is greater than or equal to the scaling factor. The scaling factor, set by using the **MouSetScaleFact** function, defines the number of mickeys the mouse must move before the mouse pointer moves one screen unit. For example, in text mode, the horizontal and vertical scaling factors are usually the same as the width and height of a character cell. This means that the mouse moves one character cell at a time, rather than moving within a cell.

A program can move the mouse pointer by using the **MouSetPtrPos** function or retrieve the current pointer position by using the **MouGetPtrPos** function.

Other mouse functions carry out special tasks for adapting the mouse for real-mode operation and for modifying the operation of one or more of the mouse functions.

As with files in the file system, the mouse is a shared resource, available to every program in the current screen group. If two or more programs open and use the mouse, what one does affects the other. For this reason, you should close the mouse when you no longer need it.

49.3 Using the Mouse

Since mouse input for Presentation Manager applications is provided automatically through the message queue, only non-Presentation Manager programs use the MS OS/2 mouse functions. Typically, you use the mouse functions in character-based programs that need mouse input. The following sections explain how to use the mouse functions.

49.3.1 Opening the Mouse

You can open the mouse by using the **MouOpen** function. This function requires the name of the device driver and a pointer to the variable that receives the mouse handle. The following code fragment opens the mouse by using the default device driver:

```
HMOU hmou;

MouOpen(NULL, &hmou);
    .
    .
    .
MouClose(hmou);
```

In this example, NULL specifies the mouse-device driver designated in the first **device** command in the *config.sys* file. The **MouOpen** function opens the mouse and copies a handle to the variable hmou. You can use this handle in subsequent mouse functions to carry out tasks, such as reading from the event queue.

You can close the mouse by using the **MouClose** function.

49.3.2 Drawing the Mouse Pointer

Once you have a mouse handle, you need to display the mouse pointer by using the **MouDrawPtr** function. When the mouse is first opened, the mouse pointer is hidden from view, so even though the user can use the mouse and your program can process mouse input, there is nothing to tell the user where the mouse is located. The following code fragment opens the mouse and then displays the mouse pointer:

```
HMOU hmou;

MouOpen(OL, &hmou);
MouDrawPtr(hmou);
      .
      .
      .
```

49.3.3 Hiding the Mouse Pointer

If you plan to write to the screen when the mouse pointer is visible, you must be careful not to write directly over the pointer. The system displays the pointer by combining the pointer-shape masks with the contents of the screen at the current pointer position. When the user moves the mouse, the system restores the previous contents of the screen, destroying whatever is there. This means that if you write to the screen while the pointer is visible, what you write will be lost when the mouse next moves.

If you need to write to a screen position that is occupied by the mouse pointer, you can hide the pointer temporarily by using the **MouRemovePtr** function. This function specifies an exclusion rectangle. The mouse pointer, upon moving into this rectangle, disappears. The following code fragment hides the mouse pointer before writing to the screen:

```
NOPTRRECT mourt;

mourt.row = 0;
mourt.col = 0;
mourt.cRow = 24;
mourt.cCol = 79;

MouRemovePtr(&mourt, hmou);
VioWrtNChar(&ch, 1, 10, 10, 0);
MouDrawPtr(hmou);
```

The **NOPTRRECT** structure takes the coordinates of the upper-left corner of the screen and the width and height of the desired rectangle. This example specifies the entire screen as the exclusion rectangle, so the program must use the **MouDrawPtr** function to redraw the pointer after writing the character to the screen.

49.3.4 Using the Event Queue

You can use the **MouReadEventQue** function to read from the mouse-event queue. This function copies the next event (if any) in the queue to a **MOUEVENTINFO** structure. The structure has four fields: **fs**, **time**, **row**, and **col**. The **fs** field specifies the action that generated the event; for example, if the mouse moved, the field is set to 0x0001. The **row** and **col** fields specify the location of the mouse when the event occurred. The **time** field specifies the system time when the event occurred. If there is no event in the queue when you call the **MouReadEventQue** function, the function fills the structure with zeros. Since zero is a valid value for the **fs**, **row**, and **col** fields, you must check the **time** field to see if an event was received (if so, the field is nonzero).

The following sample program opens the mouse and reads events from the mouse-event queue. Each time the user presses the mouse button, the program writes the letter *A* to the location of the mouse pointer on the screen.

```
#define INCL_SUB
#include <os2.h>

NOPTRRECT mourt = { 0, 0, 24, 79 };

main( )
{
    CHAR ch;
    KBDKEYINFO kbci;
    HMOU hmou;
    MOUEVENTINFO mouev;
    USHORT fWait = MOU_NOWAIT;

    ch = ' ';
    VioWrtNChar(&ch, 2000, 0, 0, 0);
    ch = 'A';

    MouOpen(0L, &hmou);
    MouDrawPtr(hmou);

    do {
        MouReadEventQue(&mouev, &fWait, hmou);
        if (mouev.time) {
            if (mouev.fs & MOUSE_BN1_DOWN) { /* is 1st button down? */
                MouRemovePtr(&mourt, hmou);
                VioWrtNChar(&ch, 1, mouev.row, mouev.col, 0);
                MouDrawPtr(hmou);
            }
        }
        KbdCharIn(&kbci, IO_NOWAIT, 0);
    } while (kbci.chChar != 'Q');

    MouClose(hmou);
}
```

After retrieving an event from the queue (and verifying that it is a valid event), this sample program checks the **fs** field to see if it contains the constant MOUSE_BN1_DOWN. If it does, then the first mouse button (typically the leftmost button) is down. The program then hides the pointer and writes the letter *A* to the screen. After it restores the mouse pointer, the program checks the keyboard to determine whether the user has pressed the Q key. The program continues to read events from the queue until the user presses the Q key.

49.3.5 Specifying the Events to be Queued

You may have noticed that the program described in the previous section seemed to ignore some mouse events if you moved the mouse quickly. The program was not ignoring the events—it just was not receiving them. The mouse-event queue is small, and once it is full, any new event bumps the oldest event from the queue. This means that you may lose information if your program cannot read from the queue fast enough.

One way to minimize the amount of information lost is to exclude certain events from being placed in the queue. You can do this by using the **MouSetEventMask** function. This function lets you disable the mouse events that you do not wish to process. For example, in the previous program, you do not need the mouse-motion events, so disabling them would reduce the chances of losing a button-down event.

You disable an event by setting the bits in the event mask for only those events that you want to process. The following code fragment enables the first-button-down event but disables all other events:

```
USHORT fsEvents;

fsEvents = MOUSE_BN1_DOWN;

MouSetEventMask(&fsEvents, hmou);
```

If you disable an event such as a button press, the system does nothing to the queue if the user presses that button, but it internally records that button press. If some recordable event occurs while the button is still down, the system adds this button-down information to the event that it passes to the queue. In other words, even if you have disabled button-down events, the **fs** field in the **MOUEVENTINFO** structure still indicates when that button is down.

You can retrieve the number of buttons on the mouse by using the **MouGetNumButtons** function. This information can help you decide which events to enable and which to disable.

You can use the **MouGetNumQueEl** function to retrieve the number of events in the queue. This function retrieves the size of the queue, as well as a count of the events in the queue. If you decide that you do not need the events already in the queue, you can flush them from the queue by using the **MouFlushQue** function.

49.4 Summary

MS OS/2 provides the following mouse functions:

MouClose Closes a mouse device.

MouDeRegister Restores the default mouse subsystem functions for all processes in the current screen group.

MouDrawPtr Draws the mouse pointer on the screen.

MouFlushQue Clears the mouse-event queue.

MouGetDevStatus Retrieves the current status of a mouse.

MouGetEventMask Retrieves the current mouse-event mask.

MouGetNumButtons Retrieves the number of buttons on the current mouse.

MouGetNumMickeys Retrieves the number of mickeys that the mouse travels for each centimeter of motion.

MouGetNumQueEl Retrieves the number of mouse events in the queue.

MouGetPtrPos Retrieves the current mouse-pointer position.

MouGetPtrShape Retrieves the shape for the mouse pointer.

MouGetScaleFact Retrieves the scaling factors for a mouse—that is, the number of mickeys the mouse must travel horizontally or vertically in order to move the mouse pointer one screen unit.

MouInitReal Initializes the real-mode mouse driver.

MouOpen Opens a mouse device.

MouReadEventQue Reads a mouse event from the queue.

MouRegister Registers a mouse subsystem.

MouRemovePtr Removes the mouse pointer from a portion of the screen.

MouSetDevStatus Sets the current status of a mouse.

MouSetEventMask Sets the mouse-event mask.

MouSetPtrPos Sets the mouse-pointer position.

MouSetPtrShape Sets the shape for the mouse pointer.

MouSetScaleFact Sets the horizontal and vertical scaling factors for a mouse.

MouSynch Synchronizes access to a mouse so that only one process accesses the mouse at a time.

Chapter

50

The Keyboard

50.1 Introduction 669

50.2 About the Keyboard 669

50.3 Using the Keyboard 670

50.3.1 Reading Keystrokes 670

50.3.2 Reading Extended ASCII Keys 671

50.3.3 Reading a String of Characters from the Keyboard ... 672

50.3.4 Opening and Using Logical Keyboards 672

50.3.5 Flushing the Keyboard Buffer 673

50.3.6 Setting the Keyboard-Input Mode 673

50.4 Summary 674

50.1 Introduction

The keyboard functions give programs direct access to the system keyboard. Programs can read individual keystrokes from a keyboard, or they can read complete strings. Keystroke information includes the character value, the scan code for the key, and the status of the keyboard, such as the state of the shift keys. You should also be familiar with the following topics:

- The file system
- Input and output control
- Shared resources
- The mouse

50.2 About the Keyboard

MS OS/2 stores each keystroke in an input buffer for the keyboard. A program can then use the **KbdCharIn** or **KbdStringIn** function to read the next keystroke from the buffer and copy it to a specified structure or buffer. A program can also use the **KbdPeek** function to look at the next character in the buffer without removing it or use the **KbdFlushBuffer** function to flush the contents of the buffer.

The keyboard functions require a keyboard handle that identifies the keyboard to read from or modify. A program can always use keyboard handle 0 to read from the system keyboard. However, if more than one process (or thread) in the same screen group attempts to read from the system keyboard at the same time, there is no guarantee that the correct process will receive the keystrokes. To prevent these conflicts, a program can create a logical keyboard by using the **KbdOpen** function. A logical keyboard receives no keyboard input until a process obtains the keyboard focus for it by using the **KbdGetFocus** function. Only the logical keyboard that has the focus can receive input.

The keyboard status defines how the keyboard operates. A program can change this status by using the **KbdSetStatus** function. Two important features of the status are echo mode and input mode. If echo mode is on, characters are displayed on the screen as they are typed; otherwise, they are not displayed. Input mode specifies whether MS OS/2 control and editing keys are processed when characters are typed. Input mode can be ASCII (cooked) or binary (raw). In ASCII mode, all MS OS/2 control and editing keys, such as CTRL+C and F3, are processed when the program reads characters by using the **KbdStringIn** function, and all MS OS/2 control keys are processed when the program reads keystrokes by using the **KbdCharIn** function. In binary mode, only the CTRL+BREAK key combination is processed in either case.

The following key combinations are MS OS/2 control keys:

CTRL+C
CTRL+H
CTRL+J
CTRL+P
CTRL+S
CTRL+Z
CTRL+BREAK

The following are MS OS/2 editing keys:

F1
F2
F3
F4
F5
DEL
ESC
INS
BKSP

Other keyboard functions carry out such special tasks as adapting the keyboard to receive country-specific information or modifying the operation of one or more of the keyboard functions.

50.3 Using the Keyboard

Since keyboard input for Presentation Manager applications is provided automatically through the message queue, only non-Presentation Manager programs use the MS OS/2 keyboard functions. Typically, you use the keyboard functions in character-based programs that need as much information as possible about each keystroke the user makes. The following sections explain how to use the keyboard functions.

50.3.1 Reading Keystrokes

You can read keystrokes from the keyboard at any time by using the **KbdCharIn** function. Keystroke data includes not only the character value of the key pressed but also the scan code, the state of the shift keys (SHIFT, CTRL, ALT, NUMLOCK, CAPSLOCK, SCROLL LOCK, INS, SYSREQ), and the system time when the key was pressed. **KbdCharIn** is typically used to process keys that the **DosRead** function cannot read, such as DIRECTION keys, but it can also be used to read any key.

When the user presses a key, MS OS/2 copies the keystroke information, in the form of a **KBDKEYINFO** structure, to the keyboard-input buffer. The **KbdCharIn** function removes the keystroke from the input buffer as it copies the information to the specified structure. The following code fragment reads a keystroke from the keyboard:

```
KBDKEYINFO kbci;

KbdCharIn(&kbci,        /* copies keystroke info. to this structure */
    IO_WAIT,            /* waits until user presses a key           */
    0);                 /* reads from the physical keyboard         */

if (kbci.chChar == 'A')                    /* is it the letter 'A'? */
```

In this example, the function reads from the default physical keyboard (handle 0) and waits for a keystroke if none is in the input buffer. Keyboard functions, like file functions, require a handle to identify the keyboard to be accessed. The physical keyboard is always available to programs and does not have to be opened to be used. Keyboard handle 0 and standard-input handle 0 are not the same. The standard-input file can be closed or redefined, but the handle to the physical keyboard cannot.

You can use the IO_WAIT constant to direct **KbdCharIn** to wait for a keystroke if there are no keystrokes in the buffer. You can use the IO_NOWAIT constant to direct the function to return immediately, even if there is no keystroke.

You can use the **KbdPeek** function to look at the next character in the buffer without removing it.

50.3.2 Reading Extended ASCII Keys

Not all keystrokes have corresponding character values. Some keys, such as the DIRECTION keys, generate extended ASCII values. An extended ASCII value is a 2-byte value in which the first byte is either zero or 0xE0 and the second byte is the scan code of the key.

To identify a DIRECTION key, you must check both the **chChar** and **chScan** fields of the **KBDKEYINFO** structure. The following sample program searches for DIRECTION keys by reading keystrokes from the keyboard:

```
#define INCL_SUB
#include <os2.h>

VOID cdecl main()
{
    CHAR ch;
    KBDKEYINFO kbci;
    USHORT usCol = 40, usRow = 13;

    ch = ' ';
    VioWrtNChar(&ch, 2000, 0, 0, 0);
    ch = 'A';
    do {
        VioWrtNChar(&ch, 1, usRow, usCol, 0);

        KbdCharIn(&kbci, IO_WAIT, 0);

        if (kbci.chChar == 0) {  /* is it extended ASCII? */
            switch (kbci.chScan) {
                case 80: /* down  */
                    if (usRow < 24) usRow++;
                    break;
                case 72: /* up    */
                    if (usRow > 0) usRow--;
                    break;
                case 77: /* right */
                    if (usCol < 79) usCol++;
                    break;
                case 75: /* left  */
                    if (usCol > 0) usCol--;
                    break;
            }
        }
    } while (kbci.chChar != 'q');
}
```

This program updates the usRow and usCol variables by looking for the UP, DOWN, LEFT, and RIGHT DIRECTION keys from the keyboard. Before reading from the keyboard, the program writes the letter *A* to the location specified by the current usRow and usCol values. This means that the user can use the DIRECTION keys to leave a trail of *A*s on the screen. The program continues to loop until the user presses the Q key.

50.3.3 Reading a String of Characters from the Keyboard

You can read a string of characters from the keyboard by using the **KbdStringIn** function. Using this function is similar to reading input from the keyboard by using the **DosRead** function and the standard-input file.

The **KbdStringIn** function, unlike the **KbdCharIn** function, echoes characters as you type them, so you do not need to write the characters separately. The function reads the specified number of characters or reads up to the end-of-line (turnaround) character. The following code fragment reads a line of text:

```
CHAR achBuf[80];
STRINGINBUF kbsi;

kbsi.cb = 80;

KbdStringIn(achBuf, &kbsi, IO_WAIT, 0);
```

The **KbdStringIn** function records the number of characters read in the **cchIn** field in the **STRINGINBUF** structure. You can use this field to enable or disable the MS OS/2 editing keys for your program. These editing keys let the user recall and modify the previously typed line. If you set the **cchIn** field to zero before making the next call to **KbdStringIn**, you disable the editing keys. Otherwise, you enable editing up to the number of characters that you specify.

50.3.4 Opening and Using Logical Keyboards

The keyboard, like the mouse, is a shared resource to which all programs in a screen group have access. To avoid conflicts between programs sharing the keyboard, MS OS/2 lets programs open and use logical keyboards. A logical keyboard is like a file in that it has a handle and corresponds to a physical device, the physical keyboard. However, a program cannot receive input from a logical keyboard (as it can from a file) unless it has requested and received the keyboard focus for the logical keyboard. Since only one logical keyboard in a screen group can have the focus at any given time, this is an effective way to manage the keyboard access of all programs in the screen group.

A typical use of a logical keyboard is in a program that has more than one thread reading keystrokes. By having each thread create a logical keyboard and then wait for the focus before reading keystrokes, you can ensure that one thread does not read keystrokes that are intended for another thread. Consider an editing program that offers multiple windows through which you can edit a file or files. If each window has a separate logical keyboard, then keyboard input intended for one window will never be inadvertently read by another.

You can open a logical keyboard by using the **KbdOpen** function. Once opened, a logical keyboard receives keystrokes only when it has the keyboard focus. You can retrieve the focus for a logical keyboard by using the **KbdGetFocus** function. This function retrieves the focus only if no other logical keyboard has it. You can request the function to wait for the focus to be freed, if it is not immediately available.

The following code fragment opens a logical keyboard and requests the keyboard focus for it:

```
HKBD hkbd;

KbdOpen(&hkbd);

KbdGetFocus(IO_WAIT, hkbd); /* retrieve focus; wait if necessary */
  .
  . /* read from the keyboard */
  .
KbdFreeFocus(hkbd);                           /* release focus */
```

Once a logical keyboard has the focus, it keeps it until you free the focus by using the **KbdFreeFocus** function, even if another thread calls the **KbdGetFocus** function. This means that you must be careful to free the focus when you no longer need it. Even if you intend to read from the logical keyboard again, you must free the focus in the meantime, in order to permit other programs to access it. The logical keyboard remains open even if it does not have the focus, so you can request the focus again without reopening the keyboard.

If you have completely finished reading from the keyboard or are about to terminate the program, you can use the **KbdClose** function to close the logical keyboard.

Even though your program may use logical keyboards, the physical keyboard is always available. That is, even if a logical keyboard has the focus, you can still read keystrokes from the physical keyboard by using keyboard handle 0.

50.3.5 Flushing the Keyboard Buffer

You can flush unwanted keystrokes from the keyboard buffer by using the **KbdFlushBuffer** function. Flushing the buffer removes all existing keystrokes in the buffer. You would typically flush the physical keyboard's buffer if you did not want to process keystrokes that were carried over from a previous program.

50.3.6 Setting the Keyboard-Input Mode

You can set the keyboard-input mode by using the **KbdSetStatus** function. The keyboard-input mode defines whether the MS OS/2 control and editing keys are interpreted as special keys or only as keystrokes. The keyboard-input mode can be binary or ASCII. If it is binary, then only the key combination CTRL+BREAK is recognized by the system as a special key. All other keys and key combinations are read as keystrokes. If the input mode is ASCII, the system recognizes the special keys.

You can set the input mode by first retrieving the current keyboard status in a **KBDINFO** structure and then setting the binary-mode constant in the **fsMask** field. The following code fragment sets the keyboard to binary mode:

```
KBDINFO kbst;

KbdGetStatus(&kbst, 0);
kbst.fsMask =
    (kbst.fsMask & ~KEYBOARD_ASCII_MODE)      /* mask off ascii mode */
    | KEYBOARD_BINARY_MODE;                   /* OR in binary mode   */
KbdSetStatus(&kbst, 0);
```

This example makes no assumptions about the keyboard status; it clears the ASCII-mode constant and then sets the binary-mode constant.

50.4 Summary

MS OS/2 provides the following keyboard functions:

KbdCharIn Reads a character from a logical keyboard.

KbdClose Closes a logical keyboard.

KbdDeRegister Restores the default **Kbd** subsystem and releases any previously registered **Kbd** subsystem.

KbdFlushBuffer Clears the keyboard-input buffer.

KbdFreeFocus Frees the focus from a logical keyboard.

KbdGetCp Retrieves the current code-page identifier.

KbdGetFocus Retrieves the focus for a logical keyboard.

KbdGetStatus Retrieves the status of a logical keyboard.

KbdOpen Opens a logical keyboard.

KbdPeek Retrieves but does not remove character and scan-code information from the input buffer of a logical keyboard.

KbdRegister Registers a **Kbd** subsystem.

KbdSetCp Sets the code-page identifier for a logical keyboard.

KbdSetCustXt Installs a custom translation table.

KbdSetFgnd Raises the priority of the foreground keyboard's thread (used by a **Kbd** subsystem, not by an application).

KbdSetStatus Sets the status for a logical keyboard.

KbdStringIn Reads a string from a logical keyboard.

KbdSynch Synchronizes access to the keyboard device driver.

KbdXlate Translates a scan code into an ASCII value.

Chapter

51

Interprocess Communication

51.1 Introduction 677

51.2 About Interprocess Communication 677
51.2.1 Semaphores 677
51.2.1.1 System Semaphores 677
51.2.1.2 RAM Semaphores 678
51.2.1.3 Fast-Safe RAM Semaphores 678
51.2.2 Signals 679
51.2.3 Pipes 679
51.2.3.1 Named Pipes 680
51.2.3.2 Working with Named Pipes 680
51.2.3.3 Using Semaphores with Named Pipes 682
51.2.4 Queues 682

51.3 Using Interprocess Communication 683
51.3.1 Using Semaphores 683
51.3.1.1 Using a System Semaphore as a Signal 684
51.3.1.2 Protecting a Resource with a RAM Semaphore 684
51.3.1.3 Managing Fast-Safe RAM Semaphores 685
51.3.2 Using Signals 686
51.3.3 Using Pipes 687
51.3.3.1 Sending Output to a Child Process 687
51.3.3.2 Creating a Server Process 687
51.3.3.3 Creating a Client Process 688
51.3.4 Using Queues 688

51.4 Summary 689
51.4.1 Semaphore Functions 689
51.4.2 Signal Functions 690
51.4.3 Pipe Functions 690
51.4.4 Queue Functions 690

51.1 Introduction

This chapter describes system and RAM semaphores, signals, pipes, and queues. Semaphores let programs signal the completion of certain tasks and control access to resources that more than one thread or process may need to use. Signals let a user or a process control the execution of another process. Pipes let two or more related or unrelated processes communicate as if they were reading from and writing to a file. Queues let one or more processes channel data to a specific process. You should also be familiar with the following topics:

- The file system
- Shared memory
- Shared resources
- Processes, threads, and sessions
- MS OS/2 program models

51.2 About Interprocess Communication

The following sections describe the methods used for passing information between processes. This exchange of data is important in a multitasking system.

51.2.1 Semaphores

A semaphore is a special variable that a program can use to signal the beginning and ending of a given operation. Semaphores are typically used in conjunction with a limited resource to prevent more than one process or one thread within a process from accessing the resource at the same time. A process can create and use three types of semaphores: system, RAM, and fast-safe RAM.

51.2.1.1 System Semaphores

System semaphores are used between processes to control access to a shared resource. Any process can create a system semaphore by using the **DosCreateSem** function. Once the semaphore is created, any other process can use it as long as it knows the semaphore name.

A system semaphore has a unique name that the process creating the semaphore must supply. The semaphore name has the following form:

\sem\*name*

The *name* parameter must conform to the rules for MS OS/2 filenames, but no actual file is created for the semaphore.

MS OS/2 supplies a semaphore handle when a system semaphore is created. The process can use this handle in subsequent semaphore functions to set, clear, and wait for the semaphore.

If any other process wants to use the system semaphore, it can open the semaphore by using the **DosOpenSem** function and supplying the specified semaphore name. When a process is finished using a system semaphore, it should close the semaphore by using the **DosCloseSem** function.

To use a semaphore, the process sets the semaphore by using the **DosSemSet** function when it wants to access the shared resource and clears the semaphore by using the **DosSemClear** function when it is finished with the shared resource. If the semaphore is an exclusive semaphore (as specified when created), only the process that created the semaphore can set or clear it.

Once a semaphore is set, the process can wait for that semaphore to become clear, or for a specified time interval to elapse, by using the **DosSemWait** function. Ideally, one process or thread waits for the semaphore while another process or thread carries out a task and then clears the semaphore. A process or thread can also wait for a given semaphore to become clear by using the **DosSemSetWait**, **DosSemRequest**, or **DosMuxSemWait** function.

51.2.1.2 RAM Semaphores

RAM semaphores are used by the threads in a given process. A RAM semaphore is an unsigned long variable defined as a global variable for the process. To use a RAM semaphore, a process simply passes to the semaphore functions a pointer to the unsigned long variable. The process does not need to create or open the semaphore, as it does a system semaphore. However, before the process uses a RAM semaphore for the first time, it must initialize the semaphore to zero.

As with a system semaphore, a process sets and clears a RAM semaphore and can wait for that semaphore to become clear before continuing.

Since a RAM semaphore is nothing more than a global variable, you must be especially careful to prevent its value from being changed in any way other than by using the semaphore functions. Changing the value of a RAM semaphore while it is in use can invalidate the semaphore's status.

51.2.1.3 Fast-Safe RAM Semaphores

Fast-safe RAM semaphores combine the reliability of a system semaphore with the performance of a RAM semaphore. These semaphores can be used between processes or between threads in a process.

Like a RAM semaphore, a fast-safe RAM semaphore is a variable in memory. If a fast-safe RAM semaphore is used only by the threads in a given process, it can be declared as a global variable. If it is used by more than one process, it must be allocated as a shared segment so that each process has access to it.

Each fast-safe RAM semaphore has a corresponding **DOSFSRSEM** structure that contains information about the process and thread that have set the semaphore, as well as a count of the number of times the semaphore has been set and a RAM semaphore.

```
typedef struct _DOSFSRSEM {
    USHORT cb;                 /* size of structure in bytes     */
    PID    pid;                /* process ID of owning process   */
    TID    tid;                /* thread ID of owning thread     */
    USHORT cUsage;             /* use count: number of times set */
    USHORT client;             /* private ID for resource        */
    ULONG  sem;                /* RAM semaphore                  */
} DOSFSRSEM;
```

Before the semaphore is used for the first time, the **cb** field must be set to 14 and all other fields to zero.

51.2.2 Signals

A signal is special input from the user or from another process that causes a process to temporarily suspend execution while a signal-handler function is executed. (A signal handler is simply a function that receives control when the signal occurs.) A process receives a signal whenever the user presses the CTRL+C or CTRL+BREAK key combination while the process is running. A process also receives a signal when another process uses the **DosSendSignal, DosFlagProcess**, or **DosKillProcess** function.

When the user presses CTRL+C or CTRL+BREAK in a full-screen session or when a full-screen program is in the active window of the Presentation Manager session, either the current foreground process or the last process to use the **DosSetSigHandler** function receives the signal. When a process calls the **DosSendSignal, DosFlagProcess,** or **DosKillProcess** function, the process specified by these functions receives the signal.

When a process receives a signal, MS OS/2 suspends execution of the main thread (thread 1) of the process and passes control to the process's signal handler. If a process has not explicitly set a signal handler by using the **DosSetSigHandler** function, a default signal handler is used. The default signal handlers for the CTRL+C, CTRL+BREAK, and **DosKillProcess** signals terminate the process. The default signal handlers for the flag signals ignore the signal.

A process can replace the default signal handler for any signal by using the **DosSetSigHandler** function. Although a signal handler has a specific form, it can carry out any action, such as cleaning up and saving files before terminating the process. When the handler has completed its activities, it can either terminate the program or return control to the point at which the process was suspended.

51.2.3 Pipes

The MS OS/2 pipe function, **DosMakePipe**, lets a program create a pipe that can be used to transfer information between related processes. A pipe is a special internal file that a process can write to and read from. The **DosMakePipe** function creates the pipe and supplies two file handles to the pipe: one for writing to the pipe, the other for reading from the pipe. A process can write to the pipe by using the **DosWrite** function and read from the pipe by using the **DosRead** function.

A pipe is typically used to direct the output of one process to the standard input of another process. To do this, a process opens a pipe, duplicates the pipe read handle as the standard-input file for a child process, and then starts the child process. The parent process can then write to the pipe and the child process can read what the parent process has written.

A pipe continues until both handles are closed. There can be no more than 65,535 bytes of unread data in a pipe at any given time. The **DosWrite** function may wait for data to be read from the pipe before completing its operation. If the read handle is closed before the write handle is closed, writing to the pipe generates an error.

51.2.3.1 Named Pipes

A named pipe allows communication between unrelated processes. Unlike the case with pipes created by using **DosMakePipe**, any process that knows its name can open and use a named pipe. To use a named pipe, one process, called the server process, creates the pipe, and another process, called the client process, opens the pipe. The server process can then connect the pipe and the server and the client can pass data back and forth by reading from and writing to the pipe.

The server process creates a named pipe by using the **DosMakeNmPipe** function. The function returns a pipe handle that can be used with subsequent pipe operations. A named pipe can be local or remote. A local named pipe can be used between any two processes on the same computer. A remote named pipe can be used between any two processes connected to the same local area network (LAN).

Each named pipe must have a unique name that distinguishes it from other named pipes. A local-pipe name has the following form:

\pipe\*name*

A remote-pipe name has the following form:

\*server***\pipe\***name*

The *name* parameter must conform to the rules for MS OS/2 filenames, but no actual file is created for the pipe.

When a server process creates a pipe, the process specifies the direction of data through the pipe. The server uses an in-bound pipe if it intends to read data from the client process, an out-bound pipe if it intends to write data to the client, or a duplex pipe if it intends to both read from and write to the client.

Data passes through a pipe as either bytes or messages, depending on the type of the pipe. The server process also specifies the pipe type when it creates a named pipe. If a pipe has byte type, the server and client process read and write bytes. If a pipe has message type, the processes read and write messages. A message is a block of data with a system-supplied header that is read or written as a single unit. The size and format of a message are defined by the server and client processes.

The server process also specifies how many instances of the named pipe can be open. Although only one client process can be connected to the pipe at any time, several processes (up to the number of instances specified) can open the pipe at the same time. The instance count is useful if a process needs to restrict access to the named pipe. A process can also specify unlimited instances.

When the server process creates a pipe, the server can also specify whether the named pipe will be inherited by child processes and whether the process writes data to a remote pipe immediately or waits to write the data when an internal buffer is full.

51.2.3.2 Working with Named Pipes

The server process establishes a connection to a client process by using the **DosConnectNmPipe** function. The client process must open the pipe by using the **DosOpen** function before the connection can be completed. If no client process has opened the pipe when the server process calls **DosConnectNmPipe**, the function either waits until a client opens the pipe or returns the error

ERROR_PIPE_NOT_CONNECTED immediately. The action taken depends on whether the server process created the pipe to wait for data.

If a client process receives ERROR_PIPE_BUSY from calling **DosOpen**, no instances of the given pipe are available. A process can wait for one to become available by using the **DosWaitNmPipe** function. The function waits until an instance is free or until the specified interval of time elapses. When an instance becomes free, the process can open the pipe by using the **DosOpen** function again. If several processes are waiting for an instance to become available, the system attempts to give the named pipe to the process that has been waiting the longest.

The server process can disconnect a client from a pipe by using the **DosDisConnectNmPipe** function. Ideally, the client process closes the pipe by using the **DosClose** function before the server process disconnects the pipe. However, if the client process does not close the pipe, **DosDisConnectNmPipe** disconnects anyway and the client process receives errors if it attempts to access the closed pipe. Note that forcing the closure of the client's pipe may discard data in the pipe before the client reads the data.

To synchronize data through a pipe, the server process can set the pipe so that read and write operations wait if no data is available or if there is no room in the pipe. Waiting permits one process to add or remove bytes to let the other process continue. The server can also set the pipe so that reading and writing do not wait. Waiting also affects whether **DosConnectNmPipe** waits for a client process to open the pipe.

A process can read and write bytes to a named pipe by using the **DosRead** and **DosWrite** functions or by using the **DosTransactNmPipe** function. Depending on access mode, the function writes a message to the pipe, reads a message from the pipe, or both. If a named pipe contains any unread data or if the named pipe is not in message mode, the **DosTransactNmPipe** function fails. If reading from the pipe, **DosTransactNmPipe** does not return until a complete message is read. This is true even if the server set the pipe so that it does not wait when reading.

A process can read data from a named pipe without also removing the data from the pipe by using the **DosPeekNmPipe** function. The function copies the specified number of bytes from the pipe, supplies a count of the number of bytes of data left in the pipe, and supplies a count of the number of bytes left in the current message, if any.

DosPeekNmPipe also specifies the state of the pipe: connected, disconnected, listening, or closing. A pipe is connected when a client process has opened the pipe and the server has called **DosConnectNmPipe**. Only connected pipes permit processes to read from and write to them. A pipe is disconnected when the server process calls **DosDisConnectNmPipe**. A pipe is also disconnected when it is first created. A pipe is listening when the server has called **DosConnectNmPipe** but a client process has not yet opened the pipe. A pipe is closing when the client process has closed the pipe by using **DosClose** but the server process has not yet disconnected the pipe. The **DosPeekNmPipe** function never waits, regardless of whether the pipe was set to wait.

A process can open, read from, write to, and close a named pipe by using the **DosCallNmPipe** function. The function is equivalent to calling the **DosOpen**, **DosTransactNmPipe**, and **DosClose** functions. If no instances of the pipe are available, **DosCallNmPipe** waits for an instance or returns without opening the pipe if the specified interval of time elapses.

A process can retrieve information about the state of the named pipe by using the **DosQNmPHandState** function. The state is a combination of the instance count, the access mode, and the pipe type specified when the pipe was created. **DosQNmPHandState** also specifies whether the process owning the handle is a server or client and whether the pipe waits if reading and writing cannot proceed.

A process can modify the state of a named pipe by using the **DosSetNmPHandState** function. For example, it can change the reading mode for the pipe, allowing a process to read bytes from the pipe instead of messages.

A process can retrieve information about a named pipe by using the **DosQNmPipeInfo** function. Information about the pipe is returned in a **PIPEINFO** structure and includes the name of the pipe, the instance count (the maximum number of times the pipe can be opened), the size of the input and output buffers for the pipe, and the client's pipe identifier, as follows:

```
typedef struct _PIPEINFO {
    USHORT cbOut;
    USHORT cbIn;
    BYTE   cbMaxInst;
    BYTE   cbCurInst;
    BYTE   cbName;
    CHAR   szName[];
} PIPEINFO;
```

51.2.3.3 Using Semaphores with Named Pipes

A server or client process can use system semaphores in conjunction with a named pipe to control access to the pipe. System semaphores are useful for any process that reads from several named pipes. The system clears a semaphore whenever data is available in the pipe. This means that the reading process can use the **DosSemWait** or **DosMuxSemWait** function to wait for data to arrive rather than devote a thread to periodically polling the pipe.

To use a system semaphore with a pipe, the process associates the semaphore with the pipe by using the **DosSetNmPipeSem** function. One or two semaphores can be associated with a named pipe—one for the server and one for the client. If there is already a semaphore associated with one end of the pipe, the old semaphore is replaced. A process can check the state of the semaphores by using the **DosQNmPipeSemState** function. Using system semaphores to control access to named pipes works only for local pipes.

51.2.4 Queues

A queue is a special linked list of data that a process can use to receive information from other processes. Processes pass information to a queue in the form of messages. The process that owns the queue can then read the messages from the queue.

A program creates a queue by using the **DosCreateQueue** function and specifying a unique queue name. The queue name has the following form:

\queues\*name*

The *name* parameter must conform to the rules for MS OS/2 filenames, but no actual file is created for the queue.

Once the queue has been created, other processes can open it by using the **DosOpenQueue** function and supplying the specified queue name. Processes that open the queue can write messages to it by using the **DosWriteQueue** function. The format of a queue message depends entirely on the process that creates the queue. The format and content must be understood by the processes writing messages to the queue.

Only the process that created the queue can read messages from it, by using the **DosReadQueue** function. The owner process can also examine messages without removing them by using the **DosPeekQueue** function or remove all messages from the queue by using the **DosPurgeQueue** function. The system automatically supplies the process identifier of the process that adds a message to the queue, so that the owner process can determine the origin of the message.

If the queue is empty when a process attempts to read from it, the process can either wait for an element to become available or continue executing without reading from the queue. If a process manages one queue, it is usually a good idea for it to wait for an element. However, if a process manages several queues, waiting for one queue means that other queues cannot be read. To avoid this problem, a process can supply a semaphore when it calls the **DosReadQueue** or **DosPeekQueue** function. The process can then continue executing without reading from the queue, since the **DosWriteQueue** function will clear the semaphore when an element is ready. If the process uses a unique semaphore for each queue, it can use the **DosMuxSemWait** function to wait for the first queue to receive an element. The semaphore can be either a RAM semaphore or a system semaphore. If it is a RAM semaphore, it must be in shared memory. If it is a system semaphore, any process that writes to the queue must also open the semaphore by using the **DosOpenSem** function.

The order in which the owner process reads messages from the queue depends on the type of queue. A queue can have first-in/first-out (FIFO), last-in/first-out (LIFO), or priority ordering. In priority ordering, the message with the highest priority is read first. Priority values range from 0 (lowest priority) through 15 (highest priority).

The **DosReadQueue** function reads either a specified element or the element at the beginning of the queue. (The beginning of the queue is determined by the queue priority. For example, the beginning of a queue with LIFO priority is the last element in the queue.) A process can use the **DosPeekQueue** function to examine the elements in the queue to determine which one to actually read. Each call to **DosPeekQueue** returns the identifier of the next element in the queue, so the function can be called recursively to move through the queue. The identifier of the desired element can then be supplied to **DosReadQueue**, to read that element from the queue.

51.3 Using Interprocess Communication

The following sections describe some specific ways to use semaphores, signals, pipes, and queues to provide and control communication between processes.

51.3.1 Using Semaphores

Semaphores are often used to control access to shared resources, such as memory, variables, and devices, or to signal other processes upon completion of specific tasks. Semaphores are useful in all MS OS/2 programs, including Presentation Manager applications.

51.3.1.1 Using a System Semaphore as a Signal

You can use an exclusive system semaphore as a signal to trigger execution of other processes. This is useful if one process provides data to many other processes. Using a semaphore as a signal frees the other processes from the trouble of polling to determine when new data is available. You use an exclusive semaphore for this signal to prevent any other process from clearing the semaphore and sending a false signal to the other processes.

The process controlling the signal first creates an exclusive system semaphore by using the **DosCreateSem** function and then immediately sets the semaphore by using the **DosSemSet** function. When the process has new data available for the other processes, it clears the semaphore. In the following code fragment, the process uses the **DosSleep** function to wait momentarily, so that all processes waiting for the semaphore get an opportunity to respond; then it sets the semaphore and repeats the cycle:

```
HSYSSEM hssm;

DosCreateSem(CSEM_PRIVATE, &hssm, "\sem\signal");

while (TRUE) {
    DosSemSet(hssm);                        /* set the semaphore */
        .
        . /* get new data */
        .
    DosSemClear(hssm);   /* signal that data is ready            */
    DosSleep(500L);      /* give all processes chance to respond */
}
```

All other processes use the **DosOpenSem** function to open this semaphore and then use the **DosSemWait** function to wait for the signal to be sent, as shown in the following code fragment:

```
HSYSSEM hssm;

DosOpenSem(&hssm, "\sem\signal");

while (TRUE) {
    DosSemWait(hssm, SEM_INDEFINITE_WAIT);
        .
        . /* process new data */
        .

}
```

51.3.1.2 Protecting a Resource with a RAM Semaphore

You can use a RAM semaphore to control access to a shared resource in a process. You simply define the semaphore as a global variable, so that all threads have access to it, and set its initial value to zero. To gain access to the resource, you can use the **DosSemWait** function to wait for any other thread to complete its access and then use the **DosSemSet** function to set the semaphore while you work with the resource. Finally, you can use the **DosSemClear** function to clear the semaphore when you are done with the resource. The following code fragment uses these three functions to control access to a resource:

```
ULONG ulRAMSem = 0;

if (DosSemWait(&ulRAMSem, 6000L) != ERROR_SEM_TIMEOUT) {

    /* Wait 6 seconds for resource to become free. */

    DosSemSet(&ulRAMSem);

    /*
     * Set the semaphore, work with the resource,
     * then clear the semaphore.
     */

    DosSemClear(&ulRAMSem);
}
```

Although you can direct the **DosSemWait** function to wait indefinitely, it is usually a good idea to set a time limit, to prevent the thread from stopping permanently if an error in another thread is preventing the semaphore from being cleared. If the interval elapses, the function returns ERROR_SEM_TIMEOUT instead of zero.

51.3.1.3 Managing Fast-Safe RAM Semaphores

A thread sets a fast-safe RAM semaphore by using the **DosFSRamSemRequest** function. The function sets the semaphore, records the identifiers for the thread and its process, and increases the use count of the semaphore by one. The thread can also set the **client** field of the **DOSFSRSEM** structure to identify the resource being controlled by the semaphore, but only after the semaphore is set. (The values in the **client** field may be useful to a **DosExitList** function handler in determining the appropriate cleanup action.) A thread should not change any other fields in the structure.

In reality, **DosFSRamSemRequest** may wait to set the semaphore, depending on whether the semaphore is already set and on the value you specify for the *lTimeout* parameter. If the semaphore is not set, **DosFSRamSemRequest** sets it, increases the use count, and returns immediately. If the semaphore is already set, **DosFSRamSemRequest** may wait until the semaphore is cleared before returning. (The function does not return unless the specified semaphore remains clear long enough for the calling thread to obtain it.) As with other semaphores, the process can specify how much time to wait before continuing execution. When the given interval elapses, the function returns whether or not the semaphore is cleared.

A thread can clear the semaphore by using the **DosFSRamSemClear** function. This function decreases the use count by one but does not actually clear the semaphore unless the use count becomes zero. This means that a thread that sets the semaphore several times must clear it the same number of times before it is really cleared. Although any thread can wait for the semaphore to clear, only the thread that created the semaphore can clear it.

The process that set a fast-safe semaphore must clear it before terminating. One way to ensure clearing of the semaphore is to use the **DosExitList** function to identify a termination function to clean up the semaphore. The termination function first calls the **DosFSRamSemRequest** function. The **DosFSRamSemRequest** function checks the process identifier in the fast-safe RAM semaphore. If it is the identifier of the process terminating, the function changes the thread identifier to the current thread and sets the use count to 1. The termination function then completes the cleanup by calling **DosFSRamSemClear** to clear the semaphore.

51.3.2 Using Signals

When a full-screen program first starts, the system enables the SIG_CTRLC, SIG_CTRLBREAK, and SIG_KILLPROCESS signals for the process. You can disable these signals by using the **DosHoldSignal** function. The following code fragment disables all signals:

```
DosHoldSignal(HLDSIG_DISABLE);
```

When a signal is disabled, the system prevents the signal from interrupting the process that disabled it. The signal remains enabled for other processes, however, including other processes in the same session. You can restore signals by using the HLDSIG_ENABLE option in the **DosHoldSignal** function.

You can disable individual signals by using the **DosSetSigHandler** function to specify that a signal should be ignored. The following code fragment disables the SIG_CTRLC signal:

```
DosSetSigHandler(NULL, NULL, &fAction, SIGA_IGNORE, SIG_CTRLC);
```

In the preceding example, the variable fAction receives a value specifying whether the signal was previously enabled or disabled. You can use the value to restore the signal to its previous state.

You can also replace the default signal handler with your own signal handler by using the **DosSetSigHandler** function. This is useful if your application creates many temporary files. Creating your own signal handler lets you clean up the files before a signal such as SIG_CTRLC terminates the application. The following code fragment defines a signal handler and sets it by using the **DosSetSigHandler** function:

```
PFNSIGHANDLER pfnsig;

VOID PASCAL FAR MySigHandler(usSigArg, usSigNum)
USHORT usSigArg;    /* furnished by DosFlagProcess if appropriate */
USHORT usSigNum;    /* number of signals being processed          */
{
    if (usSigNum == SIG_CTRLC) {
        .
        . /* delete files */
        .
    }
    return;
}
    .
    .
    .
DosSetSigHandler(MySigHandler, &pfnsig, &fAction,
    SIGA_ACCEPT, SIG_CTRLC);
```

In the preceding example, the **DosSetSigHandler** function copies the address of the previous signal handler (if any) to the variable *pfnsig*. You can use this address to call or restore the previous signal handler. If the previous handler was the default handler, the variable is set to zero.

51.3.3 Using Pipes

Pipes are useful in programs that need to pass a continuous stream of data between processes. Unlike other methods of interprocess communication, pipes let one process read from and write to another process as if it were a file. Processes can be related, unrelated, or even on different computers. The following sections show simple examples of how pipes can be used.

51.3.3.1 Sending Output to a Child Process

You can send output to a child process by using a pipe created by the **DosMakePipe** function, as shown in the following code fragment:

```
DosMakePipe();
DosDupHandle();
DosExecPgm();
DosWrite();
```

51.3.3.2 Creating a Server Process

You can create a server process for a named pipe by using the **DosMakeNmPipe** function. You need to supply a pipe name and specify the access modes, pipe type, and sizes of the input and output buffers for the pipe.

In the following code fragment, **DosMakeNmPipe** creates a pipe named \pipe\abc and supplies a unique handle identifying the pipe:

```
HPIPE hp;

DosMakeNmPipe("\pipe\abc",      /* pipe name                        */
              &hp,              /* pipe handle                      */
              PIPE_DUPLEX | PIPE_PRIVATE | PIPE_NOWRITETHROUGH,
              3 | PIPE_READMODE_BYTE | PIPE_BYTE_TYPE | PIPE_WAIT,
              512,              /* input-buffer size                */
              512,              /* output-buffer size               */
              500L);            /* default timeout for DosWaitNmPipe */
DosConnectNmPipe(hp);
.
.    /* read and write data to the pipe   */
.
DosDisConnectNmPipe(hp);
DosClose(hp);
```

Once the named pipe is created, you can immediately call the **DosConnectNmPipe** function to connect a client process to the pipe. In this example, the pipe is set to wait (PIPE_WAIT) if no client process is immediately available, so **DosConnectNmPipe** does not return until the connection is established.

Once the server connects to the client, the process can read from and write to the pipe. In the preceding example, the pipe is byte type, so you can use the **DosRead** and **DosWrite** functions to read from and write to the pipe.

After the client process finishes using the pipe, the server process can disconnect the pipe by using the **DosDisConnectNmPipe** function. The server can either connect again or close the named pipe for good by using the **DosClose** function.

51.3.3.3 Creating a Client Process

You can create a client process for a named pipe by using the **DosOpen** function. You simply supply the name of the pipe and use the appropriate access modes to open the pipe for reading, writing, or both, as shown in the following code fragment:

```
HPIPE hp;

DosOpen("\pipe\abc", &hp, &usAction, OL, O, 0x01, 0x42, OL);
    .
    . /* read and write data to the pipe */
    .
DosClose(hp);
```

The client process should check the return value from **DosOpen** to be sure the pipe was actually opened. If the pipe has not yet been created by the server process, **DosOpen** returns an error.

The client process can read data from the pipe, write data to the pipe, or both, depending on the access mode used when the pipe was created. To double-check the access mode, the client can call the **DosQNmPHandState** function to retrieve the current mode. If the pipe has byte type, the client process can use the **DosRead** and **DosWrite** functions to read from and write to the pipe. When the client process finishes using the pipe, it should close it by using the **DosClose** function.

51.3.4 Using Queues

Queues are useful in full-screen programs as a means for one process to manage input from many other processes. Named pipes also permit unrelated processes to pass data, but queues have the advantage of letting the owner process choose which data to read and process first.

Note that Presentation Manager applications also have queues, called message queues. Message queues and the queues described in this chapter are not the same.

You can create a queue by using the **DosCreateQueue** function, supplying the queue name and the queue type as arguments. The following code fragment creates the first-in/first-out queue named \queues\sample.que:

```
HQUEUE hqueue;

DosCreateQueue(&hqueue, QUE_FIFO, "\queues\sample.que");
```

Once the owner of the queue has created it, each process that needs to use the queue must open it by using the **DosOpenQueue** function. The function retrieves the queue handle and the process identifier of the process that owns the queue. The following code fragment opens the queue for another process:

```
PID pid;
HQUEUE hqueue;

DosOpenQueue(&pid, &hqueue, "\queues\sample.que");
```

A process that has opened a queue can write to the queue by using the **DosWriteQueue** function. The format of the element written to the queue depends entirely on what the owner process needs. You should identify the process that owns the queue and create elements in a form that the process can read. The following code fragment writes an element consisting of a null-terminated string to a queue:

```
DosWriteQueue(hqueue, 0, 12, "Hello, World", 0);
```

The queue owner can read an element from the queue by using the **DosReadQueue** function. For a queue that has been opened using the QUE_FIFO option, the function reads the oldest element from the queue. The function retrieves a pointer to the element and the length of the element in bytes. It also retrieves the process identifier of the process that wrote the element to the queue. The following code fragment reads an element from the queue:

```
QUEUERESULT qresc;
USHORT cb;
PVOID pv;

DosReadQueue(
    hqueue,       /* queue handle                                    */
    &qresc,       /* queue result, incl. process ID and request      */
    &cb,          /* count of bytes in element                       */
    &pv,          /* address of element                              */
    0,            /* element number (not used for QUE_FIFO)          */
    DCWW_NOWAIT,  /* do not wait for element                         */
    &bPrty,       /* recv. element priority (not used for QUE_FIFO) */
    NULL);        /* semaphore handle (not used)                     */
```

51.4 Summary

The following sections describe the functions that can be used in MS OS/2 programs to create and manage semaphores, signals, pipes, and queues.

51.4.1 Semaphore Functions

MS OS/2 provides the following semaphore functions:

DosCloseSem Closes a system semaphore.

DosCreateSem Creates a system semaphore.

DosFSRamSemClear Releases a fast-safe RAM semaphore.

DosFSRamSemRequest Sets a fast-safe RAM semaphore.

DosMuxSemWait Waits for one or more semaphores to be cleared and returns when any one of the semaphores is cleared, regardless of whether it remains cleared.

DosOpenSem Opens a system semaphore.

DosSemClear Releases a system or RAM semaphore.

DosSemRequest Sets a system or RAM semaphore if the semaphore is cleared.

DosSemSet Sets a system or RAM semaphore.

DosSemSetWait Sets a semaphore (if it is not already set) and waits for it to be cleared.

DosSemWait Waits for a semaphore to be cleared.

51.4.2 Signal Functions

MS OS/2 provides the following signal functions:

DosFlagProcess Sends a signal to the calling process.

DosHoldSignal Suspends or restores signal processing.

DosKillProcess Terminates a child process after sending a termination signal to that process.

DosSendSignal Sends a CTRL+C or CTRL+BREAK signal to the last process that has a corresponding signal handler installed.

DosSetSigHandler Installs or removes a signal handler.

51.4.3 Pipe Functions

MS OS/2 provides the following pipe functions:

DosCallNmPipe Opens a named pipe, writes to and reads from it, and closes it. This function is equivalent to **DosOpen** plus **DosTransactNmPipe** plus **DosClose**.

DosConnectNmPipe Waits for a client to open a named pipe.

DosDisConnectNmPipe Disconnects a named pipe.

DosMakeNmPipe Creates a named pipe.

DosMakePipe Creates a pipe.

DosPeekNmPipe Reads from a named pipe without removing data.

DosQNmPHandState Retrieves information about the state of a pipe handle.

DosQNmPipeInfo Retrieves information about a named pipe.

DosQNmPipeSemState Retrieves information about the named pipes associated with a specified semaphore.

DosSetNmPHandState Sets information about the state of a pipe handle.

DosSetNmPipeSem Associates a semaphore with a named pipe.

DosTransactNmPipe Writes data to and reads data from a named pipe.

DosWaitNmPipe Waits for a named-pipe instance to become available.

51.4.4 Queue Functions

MS OS/2 provides the following queue functions:

DosCloseQueue Closes a queue.

DosCreateQueue Creates and opens a queue.

DosOpenQueue Opens an existing queue for the current process.

DosPurgeQueue Removes all elements from a queue.

DosReadQueue Reads an element from a queue.

DosWriteQueue Writes an element to a queue.

Chapter

52

Timers

52.1 Introduction 695

52.2 About Timers 695

52.3 Using Timers 695
52.3.1 Counting Synchronously 695
52.3.2 Counting Asynchronously 696

52.4 Summary 696

52.1 Introduction

This chapter describes timers. Timers let programs time events by waiting for an interval to elapse or by watching for a semaphore to clear. You should also be familiar with the following topics:

- Window timers
- Multitasking
- Semaphores

52.2 About Timers

A timer is a useful and reliable way for a program to count out a given number of milliseconds. Timers are actually managed by the system. When a program requests a timer, the system monitors the system clock for the program and notifies the program when the interval elapses.

The system clock keeps a count of the number of system-clock interrupts that have occurred since the system was started. System-clock interrupts occur approximately 32 times a second, so timer intervals of less than 50 milliseconds are not recommended. All time values are in milliseconds and are rounded up to the next clock tick. The duration of the clock tick can be determined by using the **DosGetInfoSeg** function and examining the **cusecTimerInterval** field in the **GINFOSEG** structure to which the function points.

The **GINFOSEG** structure also contains the current system time. The system time is the number of milliseconds that have elapsed since the system started. A process that needs to know the precise time between the start and the end of a timer can save the system time before starting the timer and compare that value with the system time after the timer ends. The system time is reset to zero every few weeks, so a process may need to take this into account when comparing starting and ending times. The system time may occasionally lose a millisecond if a process disables interrupts for periods longer than the clock-tick interval. However, the time of day (hours, minutes, and seconds), the time in seconds since January 1, 1970, and the date will remain accurate.

52.3 Using Timers

Timers are typically used to let a program pause before processing user input or to let a program carry out a task at a given time. Since timers for Presentation Manager applications are provided through the message queue, only non-Presentation Manager programs use the MS OS/2 timer functions.

52.3.1 Counting Synchronously

You can cause your program to pause for a given number of seconds by using the **DosSleep** function. The **DosSleep** timer function waits the specified number of milliseconds before returning control. The following code fragment causes the program to pause for 60 seconds (60,000 milliseconds):

```
DosSleep(60000L);
```

DosSleep actually yields execution control to the system, so it is also a convenient way to let other processes or threads execute while you pause. This is useful if two processes (or two threads) need to synchronize their execution. **DosSleep** yields control even if you specify zero milliseconds. As long as the processes have equal priority, control does not return from **DosSleep** until the other process has had an opportunity to execute.

52.3.2 Counting Asynchronously

If a program wants to carry out other tasks while the timer counts out the interval, it can use the **DosTimerAsync** function. This function sets a timer without stopping the program. When the interval elapses, the system clears a given semaphore. The program must monitor the semaphore to determine when the time has elapsed.

The following code fragment creates a system semaphore and then calls the **DosTimerAsync** function to count an interval while the program performs other activities:

```
DosCreateSem(CSEM_PUBLIC, &hSem, "\\sem\\abc.sem");
DosSemSet(hSem);
DosTimerAsync(5000L, hSem, &Timer);     /* start timer          */
    .
    . /* other processing */
    .
DosSemWait (hSem, SEM_INDEFINITE_WAIT); /* wait until timer     */
                                        /* clears the semaphore */
```

If the program wants the timer to count out the interval repeatedly, it can use the **DosTimerStart** function. Unlike the **DosTimerAsync** function, **DosTimerStart** does not stop after the first interval is counted. It repeats the count and clears the semaphore each time the interval elapses.

A process can stop a timer by using the **DosTimerStop** function.

52.4 Summary

MS OS/2 provides the following timer functions:

DosSleep Causes the current thread to pause for a specified interval of time.

DosTimerAsync Creates a timer that allows the program to carry out other tasks during the timer interval.

DosTimerStart Creates a timer that counts out the specified interval repeatedly, until the **DosTimerStop** function is called.

DosTimerStop Stops a timer created by the **DosTimerAsync** or **DosTimerStart** function.

Chapter

53

Window Timers

53.1 Introduction 699
53.2 About Window Timers 699
53.3 Using Window Timers 700
53.4 Summary 701
53.4.1 Functions 702
53.4.2 Messages 702

53.1 Introduction

This chapter describes the functions and messages that let an application post a timer message at a specified time. You should also be familiar with the following topics:

- Windows
- Messages
- Timers

53.2 About Window Timers

A window-timer message is an input message that the system posts to a message queue after a specified period of time elapses. The period of time, called the time-out value, is expressed in milliseconds. An application starts the timer for a given window and specifies the time-out value. The system counts down approximately that number of milliseconds and then posts a WM_TIMER message to the message queue for the corresponding window.

The time-out value can be any value in the range zero through 65,535. However, MS OS/2 cannot guarantee that all values are accurate. The actual time-out depends on how often the application retrieves messages from the queue and on the system clock rate. In many computers, the MS OS/2 system clock ticks about every 50 milliseconds, but this can vary widely from computer to computer. In general, a timer message cannot be posted more frequently than every system clock tick. To make the system post a timer message as often as possible, an application can set the time-out value to zero.

Timer messages are continuous—that is, after timing out, a system starts the countdown again. Once an application starts a timer, the system repeats the timing cycles until the application stops the timer.

An application starts a timer by using the **WinStartTimer** function. If a window handle is given, the timer is created for that window. In this case, the **WinDispatchMsg** function dispatches the WM_TIMER message to the given window when the message is retrieved from the message queue. If a NULL window handle is given, it is up to the application to check for WM_TIMER messages and dispatch them to the appropriate window.

The number of timers an application can start is limited; in fact, the number of available timers for the system is limited. An application can check the total number of available timers by reading the SV_CTIMERS system value. However, this value does not specify how many timers have already been started. In general, an application should not use more than three or four timers at the same time.

Every timer has a unique timer identifier, also called a timer ID. An application can request that a timer be created with a particular identifier or have the system choose a unique value. When a WM_TIMER message is received, the timer identifier is contained in the first message parameter. Timer identifiers allow an application to determine the source of the WM_TIMER message.

Three timer identifiers are reserved and should not be used by applications. The system timer identifiers and their symbolic constants are as follows:

Value	Meaning
TID_CURSOR	The cursor-blinking timer. This controls cursor blinking. Its time-out value is stored in the *os2.ini* file under the CursorBlinkRate keyname in the PM_ControlPanel section.
TID_FLASHWINDOW	The window-flashing timer.
TID_SCROLL	The scroll-bar-repetition timer. This timer controls scroll-bar response when the mouse button or a key is held down. Its time-out value is specified by the system value SV_SCROLLRATE.

WM_TIMER messages, like WM_PAINT and the semaphore messages, are not posted to a message queue. Instead, when the time-out elapses, the system sets a record in the queue indicating which timer message has been posted. The system builds the WM_TIMER message when the application retrieves the message from the queue.

Although a timer message may be in the queue, if there any higher priority messages in the queue, the application retrieves the other messages first. If the time-out elapses again before the message has been retrieved, the system does not create a separate record for this timer. This means the application should not depend upon the timer messages being processed at precise intervals. To check the accuracy of the message, an application can retrieve the actual system time by using the **WinGetCurrentTime** function. Comparing the actual time with the actual time of the previous timer message can be useful in determining what action to take for the timer.

53.3 Using Window Timers

You can start a timer with the **WinStartTimer** function. You supply the window handle and a time-out value. The function associates the timer with the specified window. The new timer starts counting down as soon as it is created. The following code fragment demonstrates starting a timer and setting it for every half second (500 milliseconds):

```
WinStartTimer(hab,     /* anchor-block handle */
    hwnd,              /* window handle       */
    0,                 /* timer ID (not used) */
    500);              /* 500 milliseconds    */
```

You can set the timer anywhere in your application as long as you have a valid window handle. To process the timer message, you need to add a WM_TIMER case to the window procedure for the given window. You can use the case to carry out any actions related to the timer. If the timer is no longer needed, you can stop it by using the **WinStopTimer** function. The following code fragment shows a typical case statement:

```
case WM_TIMER:
        .
        . /* Carry out timer-related tasks. */
        .

    WinStopTimer(hab, hwnd, 0);    /* stops the timer for this window */
    return (0L);
```

You can reset the time-out value for the timer for a given window by calling the **WinStartTimer** function again. You can reset the timer without stopping it.

If you supply a window handle, you can start only one timer for that window. If you attempt to create another timer using the same window handle, the system just resets the time-out for the previous timer. If you need more than one timer for a window or application, you can create multiple timers by using a NULL window handle. The **WinStartTimer** function creates a timer and returns a timer identifier that uniquely identifies it. The following code fragment creates two timers:

```
USHORT idTimer1, idTimer2;

idTimer1 = WinStartTimer(hab, NULL, 0, 500);
idTimer2 = WinStartTimer(hab, NULL, 0, 1000);
```

Since there is no window associated with these timers, the application must process the timer messages once they are retrieved from the message queue; since the timer messages have no window handles, the **WinDispatchMsg** function in the message loop cannot dispatch them. The following code fragment shows a message loop that handles the window timers:

```
HWND hwndTimerHandler; /* handle of window for timer messages */
QMSG qmsg;             /* queue-message structure            */

while (WinGetMsg(hab, &qmsg, NULL, 0, 0)) {
    switch(qmsg.msg) {
        case WM_TIMER:
            qmsg.hwnd = hwndTimerHandler;

        default:
           WinDispatchMsg(hab, &qmsg);
    }
}
```

If a window receives multiple timer messages, it can use the first message parameter of the message to identify the timer, because the system copies the timer identifier to this parameter.

If you need to change the time-out value for a timer, you must specify the timer identifier with the new time-out value. The following code fragment sets the second timer to 2 seconds:

```
idTimer2 = WinStartTimer(hab, NULL, idTimer2, 2000);
```

When you first start a timer that has no associated window, the **WinStartTimer** function creates an arbitrary timer identifier unless you explicitly provide one. You can request your own timer identifier when you first start a timer, but you must make sure it is not one of the reserved system timers.

53.4 Summary

This section lists all the functions and messages an application can use to start, stop, and use window timers.

53.4.1 Functions

The following functions allow your application to use window timers:

WinGetCurrentTime Retrieves the system time. (The system time is the number of milliseconds since the system started.)

WinStartTimer Starts or resets a timer. A timer may be associated with a window or a thread.

WinStopTimer Stops a timer. This function should be called for every timer that an application starts.

53.4.2 Messages

The following message is used to communicate the end of a time-out interval:

WM_TIMER Posted when a timer times out. The low word of the *mp1* parameter contains the timer identifier. The value returned by the application that processes this message is zero.

Chapter
54

Device Monitors

54.1 Introduction 705
54.2 About Device Monitors 705
54.3 Using Device Monitors 705
54.4 Summary 705

54.1 Introduction

The chapter describes device monitors and the device-monitor functions. These functions examine and modify input from devices such as the keyboard and the mouse before the input is made available to any other program. You should also be familiar with the following topics:

- The file system
- Devices

54.2 About Device Monitors

A device monitor is a special type of program input in which a program receives raw input directly from a device such as the keyboard or the mouse before the input is passed on to any program that may be reading from the device. A program can use a device monitor to inspect, modify, insert, and remove information as it passes from the device to other programs.

54.3 Using Device Monitors

A program creates a device monitor for a given device by using the **DosMonOpen** function. The program does not receive input from the device until it uses the **DosMonReg** function to register the input and output buffers to be used with the monitor. These buffers are structures into which MS OS/2 copies the input or reads the output. The format of the structures depends on the device. Typically, a program retrieves the correct size of the input and output buffers by trying to register the buffers with an incorrect size. The **DosMonReg** function copies the correct size to the first field in each structure.

Once a monitor is registered, a program can retrieve input from the device by using the **DosMonRead** function. It can modify this information and then pass it on by using the **DosMonWrite** function. If the program does not pass the information on by using the **DosMonWrite** function, that information is lost.

You can discontinue input monitoring by using the **DosMonClose** function.

54.4 Summary

MS OS/2 provides the following device-monitor functions:

DosMonClose	Closes a device monitor.
DosMonOpen	Creates or opens a device monitor.
DosMonRead	Retrieves input from a device.
DosMonReg	Registers the input and output buffers to be used by the monitor.
DosMonWrite	Writes information to a device.

Chapter
55

Atom Tables

55.1 Introduction 709

55.2 About Atom Tables 709
55.2.1 String Atoms 709
55.2.2 Integer Atoms 710

55.3 Using Atom Tables in an Application 710
55.3.1 Obtaining an Atom-Table Handle 710
55.3.2 Creating an Atom 711
55.3.2.1 Creating a Unique Window-Message Atom 711
55.3.2.2 Creating DDE Formats and a Unique Clipboard Format 712
55.3.3 Deleting an Atom 713

55.4 Summary 713

55.1 Introduction

This chapter describes how to use atom tables in your applications. You should also be familiar with the following topics:

- Code pages
- Window messages and message queues
- Clipboard formats

55.2 About Atom Tables

The MS OS/2 atom manager provides a mechanism for converting a string (atom name) into a 16-bit word (atom) that can be used as a constant to represent the string in various data structures for the application and the system. You can save space by converting strings to atoms when the same string must be kept in a number of data structures. You can also save time when searching for a particular string, since after you convert the search string to an atom you can compare words with the atoms stored in the data structures. There are other situations in which you must convert a string into an atom to be sure that the atom is unique throughout the system for example, when creating a new clipboard format of an interapplication-message type.

The atom manager uses an atom table to hold the strings associated with atoms and the control structures needed to query the table to determine if a string is there. Each atom has an associated use count that specifies how many times it is added to the table. Many different applications or threads within a single application can add the same atom string to the system atom table. Each time an atom string is added to the table, the use count for the corresponding atom is incremented. Each time an application or thread deletes an atom, the atom's use count is decremented. An atom is removed from the system atom table when its use count reaches zero.

55.2.1 String Atoms

Applications pass null-terminated strings to atom tables and receive string atoms (16-bit integers) in return. Atom tables have the following properties:

- The maximum length of an atom name is 255 characters. A zero-length string is not a valid atom name.
- Case is significant when searching for an atom name in an atom table and the entire string must match that is, no substring matching is performed.
- The maximum amount of data that can be stored in an atom table is 64K. This includes any control data needed by the atom manager to manage the atom table.
- The maximum number of string atoms allowed is 16K. The values of string atoms can be from 0xC000 through 0xFFFF.
- A use count is associated with each string atom. The use count is incremented each time the atom is added to the table and decremented each time the atom is deleted from the table. This allows different users of the same string atom to avoid destroying each other's atoms.

- Atom tables can be used only by the process that creates them. Only the system atom table can be used by multiple processes.

55.2.2 Integer Atoms

Integer atoms differ from string atoms as follows:

- Integer atoms are values from 0x0001 through 0xBFFF. The values of integer atoms and string atoms do not overlap, so the two types of atoms can be intermixed.
- The string representation of an integer atom is *#ddddd*, where *ddddd* are decimal digits. Leading zeros are ignored. These strings must be specified in the system code page.
- There is no use count or storage overhead associated with an integer atom.
- Integer atoms are useful for predefined system constants exported by a dynamic-link library, because they behave exactly like atoms except that they have no overhead. An example of using integer atoms is in the predefined MS OS/2 window classes. Application-defined window-class names are strings that are converted into atoms and then used to determine if a class name is being defined more than once; this means the predefined window classes implemented by MS OS/2 need to be expressible as atoms. When these atoms are integer atoms, they can be expressed as compile-time constants in an MS OS/2 header file; the application can refer to these classes and create windows by using them without including a string constant in its data segment.

55.3 Using Atom Tables in an Application

Applications that use atom tables can create their own atom tables or use the system atom table. Once an atom table is created or acquired, the applications can add and delete atoms and retrieve information about individual atoms in the table. When the application is finished using the atom table, it should delete the table.

There are two main reasons for using an atom table. First, registering atoms in the system atom table ensures that the resulting atom is unique system-wide. (It is important that atoms be unique when you define window messages or clipboard and DDE formats that are used between applications.) The system atom table is also useful when several applications using a common message or format must use the same atom to identify the message or format.

The second use for an atom table is to allow an application to manage efficiently a large number of strings that are used only within the application. The application can create a private atom table for this purpose.

55.3.1 Obtaining an Atom-Table Handle

You must obtain a handle of an atom table before performing any atom-manager operations. To obtain a handle of the system atom table, call the **WinQuerySystemAtomTable** function. To create your own atom table, call the **WinCreateAtomTable** function. The atom-table handle returned by either of these calls must be used for all other atom-manager functions.

55.3.2 Creating an Atom

To create an atom you call the **WinAddAtom** function, passing an atom-table handle and a pointer to an atom string. The atom manager searches the specified atom table for an occurrence of the atom string. If the string already resides in the atom table, its use count is incremented and the corresponding atom is returned to the caller. Repeated calls to add the same atom string return the same atom. If the atom string does not exist in the table when **WinAddAtom** is called, the string is added to the table, its use count is set to one, and a new atom is returned.

Atom strings can be specified by using a far pointer that can be interpreted in one of the following four ways:

Format	Description
"!",*atom*	The pointer is to a string in which the atom is passed indirectly, as a value.
#ddddd	The pointer is to an integer atom specified as a decimal string.
long word: FFFF(high word)	The atom is passed directly in the low word of the *pszAtomName* parameter of the **WinAddAtom** function. This format is used extensively by MS OS/2 to add predefined window classes and window messages to the system atom table. By adding the atoms as integers, the value of the atoms for these messages can be determined before compiling and included as constants in the MS OS/2 header files.
string atom name	The pointer is to a string atom name. This is the pointer format most often used by applications to add an atom string to an atom table and receive an atom in return.

The "!",*atom* and long word: FFFF(high word) formats are useful when incrementing the use count of an existing atom for which the original atom string is not known. For example, the system clipboard manager uses the long word: FFFF(high word) format to increment the use count of each clipboard-format atom when that format is placed on the clipboard. By using this format, the atom is not destroyed even if the original user of the atom deletes the atom, because the use count still shows that the clipboard is using the atom.

55.3.2.1 Creating a Unique Window-Message Atom

System-defined window messages are identified by word-length constants in the range zero through 0x1000 (WM_USER). An application that defines its own window messages can use WM_USER and higher values as long as it sends those messages only to itself. If an application sends its own window messages to other applications, either directly or by calling the **WinBroadcastMsg** function, it must add its message identifiers to the system atom table to obtain a message identifier that is unique for the entire system.

Typically, an application registers its own window-message types with the system atom table only if those types are likely to be recognized by other applications. For example, two applications might communicate with each other with an agreed-upon message that is not defined by the system. These applications must use the same string identifier for the shared message type—for example, OUR_LINK_MESSAGE. Each time the applications run, they add this string to the system atom table and receive an atom in return. Because both applications register the same string in the system atom table, they both receive the same atom. This atom can then be used to identify the message without conflicting with other system-wide message identifiers.

A consequence of using atoms to identify a window message is that the message cannot be decoded as a C-language case statement, as is typically done, because the value of the atom cannot be known until run time. For example, typical window-procedure code is similar to the following code fragment:

```
/* This procedure does not work for interapplication messages. */

switch (usMessage) {
    case WM_PAINT:
        hps = WinBeginPaint(hwnd, NULL, &rect);
        WinFillRect(hps, &rect, CLR_WHITE);
        WinEndPaint(hps);
        return OL;
}
return (WinDefWindowProc(hwnd, usMessage, mp1, mp2));
```

Each case statement for a message uses a constant value to identify the message. This is not possible for messages registered with the atom manager at run time, since these messages cannot be determined at compile time and cannot be used in case statements. Instead, you must add a default case that checks the value of the message against the value of the atoms you have registered, as shown in the following code fragment:

```
switch (usMessage) {
    case WM_PAINT:
        hps = WinBeginPaint(hwnd, NULL, &rect);
        WinFillRect(hps, &rect, CLR_WHITE);
        WinEndPaint(hps);
        return OL;

    default:
        if (usMessage == usMessageAtom)
            return DoOurMessage(...);
        break;

}
return (WinDefWindowProc(hwnd, usMessage, mp1, mp2));
```

55.3.2.2 Creating DDE Formats and a Unique Clipboard Format

The system defines several standard clipboard and DDE formats, identified by word-length constants. Applications that define their own clipboard or DDE formats must register those formats in the system atom table to avoid conflicting with the predefined formats and any formats used by other applications.

An application must register any nonstandard clipboard format, even if the format is used only for cutting and pasting within the application. This is necessary because numbering conflicts can occur among nonstandard clipboard formats, since all formats on the clipboard are always available to all applications in the system. All nonstandard DDE formats must be registered in the system atom table, since they are always used by more than one application.

All applications that share a clipboard or DDE format must use the same string to identify the format. Each application adds the common atom string to the system atom table and uses the resulting atom when using the clipboard or DDE functions. All applications receive the same atom from the system atom table; there is no conflict with other formats that are registered with the system atom table.

55.3.3 Deleting an Atom

When an application is finished using an atom, it should call the **WinDeleteAtom** function. This function reduces the atom's use count by one. If the use count is greater than zero, the atom remains in the atom table, since other processes are still accessing the atom. **WinDeleteAtom** removes an atom from the atom table only if the use count is zero.

55.4 Summary

The following functions are associated with atom tables:

WinAddAtom Adds an atom string to an atom table, returning an atom. If the atom string exists in the table, the atom's use count is increased by one.

WinCreateAtomTable Creates a new atom table of a specified size and returns its handle.

WinDeleteAtom Reduces the use count of an atom by one. When the use count reaches zero, the atom is removed from the atom table.

WinDestroyAtomTable Reduces the use count of an atom table by one. When the use count reaches zero, the atom table is deleted.

WinFindAtom Finds an atom string in an atom table and returns an atom.

WinQueryAtomLength Returns the length of the atom string associated with the specified atom. This function allows an application to determine the size of the buffer to pass to the **WinQueryAtomName** function.

WinQueryAtomName Retrieves the atom string associated with the specified atom.

WinQueryAtomUsage Returns an atom's use count.

WinQuerySystemAtomTable Returns the handle of the system atom table.

Chapter

56

System Information

56.1 Introduction 717

56.2 About System Information.. 717
56.2.1 The MS OS/2 Initialization File 717
56.2.2 The Shell.. 718
56.2.2.1 The Program List 718
56.2.2.2 The Switch List.................................... 719
56.2.3 System Colors and Values 719
56.2.3.1 System Colors 719
56.2.3.2 System Values 721

56.3 Using System Information 724
56.3.1 The MS OS/2 Initialization File 725
56.3.2 The Shell.. 726
56.3.3 System Colors.. 727
56.3.4 System Values.. 728

56.4 Summary 729
56.4.1 The MS OS/2 Initialization File 729
56.4.2 The Shell.. 730
56.4.3 System Colors and Values 730
56.4.3.1 Functions... 730
56.4.3.2 Messages 731

56.1 Introduction

This chapter describes the portions of MS OS/2 that let you work with the MS OS/2 initialization file, *os2.ini*, that let you add applications to the switch list in Task Manager and to the program list in Start Programs, and that let you set colors for and retrieve information about the Presentation Manager session. You should also be familiar with the following topics:

- The file system
- Windows and window components
- Color

56.2 About System Information

The following sections describe the methods used to set and retrieve system information.

56.2.1 The MS OS/2 Initialization File

The *os2.ini* file contains configuration information for both MS OS/2 and individual applications. MS OS/2 uses information in the file to initialize the Presentation Manager session when it starts. An application can use the information to initialize its windows and data when it starts.

The *os2.ini* file is a binary file, so the user cannot view or edit it directly. The file consists of one or more sections, each section containing one or more settings, or keys. Each key consists of two parts: a name and a value. Both section and key names are null-terminated strings. A key value can be a null-terminated string, a null-terminated string representing a signed integer, or individual bytes of data.

Each section must have a unique name, and within a section, each key must have a unique name. The file can contain up to 6120 sections, with 120 sections reserved for the system. Each section can contain up to 6120 keys.

Section names and key names are each limited to 1024 bytes, including a required terminating null character. Although section names and key names are typically composed of ASCII characters in the range 0 through 255, they can actually be any null-terminated sequence of bytes. Each key value is limited to 65,535 bytes. Individual bytes can have any value, including zero. Thus, key values can contain any sort of binary data, including but not limited to null-terminated character strings.

The following is an example of the PM_Colors section. It shows the key names and values stored by Control Panel for system colors:

```
PM_Colors
  Background = 138 117 202
  AppWorkspace = 244 170 181
  Window = 128 128 128
  WindowText = 0 0 0
  WindowStaticText = 0 0 127
  Menu = 128 128 128
  MenuText = 128 0 0
  ActiveTitle = 255 213 138
  InactiveTitle = 47 47 47
```

```
TitleText = 0 128 128
ActiveBorder = 0 0 0
InactiveBorder = 186 48 170
WindowFrame = 0 0 0
Scrollbar = 48 48 58
HelpBackground = 255 255 255
HelpText = 0 0 127
HelpHilite = 0 127 127
```

Several sections of *os2.ini* are maintained by the system. The names of these system sections begin with the characters PM, as in the preceding example, PM_Colors. Specific applications can add additional sections to *os2.ini*. The application name should be part of the section name.

Several of the *os2.ini* functions can be used to enumerate all of the *os2.ini* section names or all of a particular section's key names. These functions copy the section names to a buffer that the application supplies. The names are copied as a consecutive series of null-terminated character strings, the last string marked with an additional null byte.

56.2.2 The Shell

The Start Programs and Task Manager applications let users start and control applications. The user uses the Start Programs window to start applications listed in the program list and uses the Task Manager window to switch between and terminate processes listed in the switch list. The program list contains the titles of installed applications, organized in groups, and provides the information the system needs to start those applications. The switch list contains a list of currently running processes and provides the information the system needs to switch between and terminate processes.

Both the program list and the switch list are accessible from an application. Typically, though, an application interacts with the switch list and leaves program-list operations to the user.

56.2.2.1 The Program List

Start Programs maintains the program list, which is divided into groups. The root group, known to the user as the Main Group, is the first group displayed when Start Programs starts. It is specified in shell functions with the group-handle constant SGH_ROOT.

Applications can create new groups for the program list, add applications to the list, and retrieve information about applications in the list by using the **WinCreateGroup**, **WinAddProgram**, **WinQueryDefinition**, and **WinQueryProgramTitles** functions. Users can also change and delete applications from the program list, but applications cannot.

When you use these function to create a group or add an application, the functions return a unique program-list handle. The handle is used in all subsequent shell functions to identify the group or application.

To add an application to the program list, the application must fill a program-information-block (**PIBSTRUCT**) structure with information the program list needs about an application, including a program title, a path and filename for the executable file, a default directory, and startup parameters. A pointer to the **PIBSTRUCT** structure is passed as a **WinAddProgram** parameter.

The **XYWINSIZE** structure, used in **PIBSTRUCT**, is a window-size structure. It specifies the initial size and position of an application's main window. It also includes a state variable that can be used to set the window to a minimized, maximized, invisible, or normal state, as well as to specify whether the window closes automatically when the application terminates.

An application can use the **WinQueryProgramTitles** function to retrieve information about an application or group in the program list. This function copies the information to an array of **PROGRAMENTRY** structures.

56.2.2.2 The Switch List

Task Manager maintains a list of current processes, or tasks. This list is called the switch list. An application should register with the switch list before entering its main message loop. Task Manager displays current switch-list tasks to the user. After an application registers with the switch list, a title string representing the task appears in the Task Manager window.

To avoid confusion, an application's switch-list title string should match the string that appears in its main window's title bar. If an application changes its main window's title bar string during operation—for example, to append the name of an open file—it should change its switch-list entry as well by using the **WinChangeSwitchEntry** function.

56.2.3 System Colors and Values

MS OS/2 provides system-wide values that an application can use to help determine the color and dimensions of windows and other objects it draws in its client area. These values are also used by frame windows to set the colors and sizes for title bars, scroll bars, sizing borders, and other frame controls and elements associated with a frame window.

If an application changes the system colors and values, the change affects all applications.

56.2.3.1 System Colors

The system colors are a set of colors that MS OS/2 uses to draw windows. The system colors are associated with the elements of the Presentation Manager session, such as the desktop, title bars, scroll bars, text, and window background. Typically, the system sets the colors of these elements to default values when it first starts. The user can change the values at any time by using the Control Panel application. Any changes the user makes to the system colors are written to both the system logical color table and the *os2.ini* file. Applications can use the system color values to draw elements in their own client windows.

MS OS/2 maintains 17 system colors. A set of symbolic constants, defined in the MS OS/2 include files identifies these colors. These symbolic constants have the prefix SYSCLR and can be used like other color indexes when an application draws in a client window. The following list gives the SYSCLR constants and their meanings:

Value	Meaning
SYSCLR_BACKGROUND	Desktop background
SYSCLR_WINDOWFRAME	Window frame
SYSCLR_TITLETEXT	Title-bar text
SYSCLR_ACTIVETITLE	Active window title-bar background
SYSCLR_INACTIVETITLE	Inactive window title-bar background
SYSCLR_MENUTEXT	Menu text
SYSCLR_MENU	Menu background
SYSCLR_WINDOWTEXT	Window text
SYSCLR_WINDOWSTATICTEXT	Window static text
SYSCLR_WINDOW	Window background
SYSCLR_APPWORKSPACE	Multiple-document-interface window background
SYSCLR_SCROLLBAR	Scroll-bar shaft
SYSCLR_ACTIVEBORDER	Active border
SYSCLR_INACTIVEBORDER	Inactive border
SYSCLR_HELPTEXT	Help-window text
SYSCLR_HELPHILITE	Help-window highlighted text
SYSCLR_HELPBACKGROUND	Help-window background

The system stores system-color data in the PM_Colors section of the *os2.ini* file. Each system color has a key in that section. Each key value is an ASCII string of three integer values in the range 0 through 255, separated by spaces. The integers represent the red, green, and blue components of an RGB color value, in that order.

An application can query and set system colors by using the **WinQuerySysColor** and **WinSetSysColors** functions. Changes made with **WinSetSysColors** are not permanent, since they are not written to the PM_Colors section of the *os2.ini* file. To make permanent system-color changes, you must explicitly write to the PM_Colors section of the *os2.ini* file by using the **WinWriteProfileString** and **WinWriteProfileData** functions. Remember that changes in the system colors affect all applications.

MS OS/2 specifies colors with 24 bits of red-green-blue intensity information, 8 bits for each color. An intensity value can be any value in the range 0 through 255. The 3 bytes are packed into a 4-byte long integer; the fourth byte is currently unused. Although a device may be capable of producing many colors, it typically does not give access to all colors at all times. Instead, it gives access to the number of colors that can fit in an array called the physical palette. Each system-color index maps to one of the entries in the physical palette. When an application changes a system color, it specifies a new RGB value. The system then finds the closest matching color in the physical palette and maps the corresponding system-color index to the new palette entry. How closely the new color matches the color the application requested depends on the current contents of the physical palette.

When an application changes the current system colors by using the **WinSetSysColors** function, that function sends a WM_SYSCOLORCHANGE message to all top-level windows. In addition, it invalidates the desktop window, triggering redrawing of all windows. If color variables in your application are based on the system colors, you can process the WM_SYSCOLORCHANGE message and adjust them.

56.2.3.2 System Values

The system values, maintained by MS OS/2, determine various system attributes that affect the appearance and behavior of elements of the system. These system values can be read and changed by an application.

The system values are referred to with predefined constants that have the prefix SV. The following list shows the system-value constants:

Value	Meaning
SV_CMOUSEBUTTONS	Specifies the number of mouse buttons: 1, 2, or 3.
SV_MOUSEPRESENT	Specifies whether the mouse is present. A value of TRUE means the mouse is present.
SV_SWAPBUTTON	Specifies if the mouse buttons are swapped. TRUE if mouse buttons are swapped.
SV_CXDBLCLK	Specifies the mouse double-click horizontal spacing. The horizontal spatial requirement for considering two mouse clicks a double-click is met if the horizontal distance between two mouse clicks is less than this value.
SV_CYDBLCLK	Specifies the mouse double-click vertical spacing. The vertical spatial requirement for considering two mouse clicks a double-click is met if the vertical distance between two mouse clicks is less than this value.
SV_DBLCLKTIME	Specifies the mouse double-click time in milliseconds. The temporal requirement for considering two mouse clicks a double-click is met if the time between two mouse clicks is less than this value.
SV_CXSIZEBORDER	Specifies the count of pels along the *x*-axis in the left and right parts of a window sizing border.

Value	Meaning
SV_CYSIZEBORDER	Specifies the count of pels along the *y*-axis in the top and bottom sections of a window sizing border.
SV_ALARM	Specifies whether calls to **WinAlarm** generate a sound. A value of TRUE means sound is generated.
SV_CURSORRATE	Specifies the cursor blinking rate in milliseconds. The blinking rate is the time that the cursor remains visible or invisible. Twice this value is the length of a complete cursor visible/invisible cycle.
SV_FIRSTSCROLLRATE	Specifies the delay (in milliseconds) until scroll bar autorepeat activity begins when the mouse is held down on a scroll bar arrow or within a scroll bar.
SV_SCROLLRATE	Specifies the delay (in milliseconds) between scroll bar autorepeat events.
SV_NUMBEREDLISTS	Reserved.
SV_ERRORFREQ	Specifies the frequency (in cycles per second) of a **WinAlarm** WA_ERROR sound.
SV_NOTEFREQ	Specifies the frequency (in cycles per second) of a **WinAlarm** WA_NOTE sound.
SV_WARNINGFREQ	Specifies the frequency (in cycles per second) of a **WinAlarm** WA_WARNING sound.
SV_ERRORDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** WA_ERROR sound.
SV_NOTEDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** WA_NOTE sound.
SV_WARNINGDURATION	Specifies the duration (in milliseconds) of a **WinAlarm** WA_WARNING sound.
SV_CXSCREEN	Specifies the count of pels along the screen's *x*-axis.
SV_CYSCREEN	Specifies the count of pels along the screen's *y*-axis.
SV_CXVSCROLL	Specifies the count of pels along the *x*-axis of a vertical scroll bar.
SV_CYHSCROLL	Specifies the count of pels along the *y*-axis of a horizontal scroll bar.

Value	Meaning
SV_CXHSCROLLARROW	Specifies the count of pels along the *x*-axis of a horizontal scroll-bar arrow.
SV_CYVSCROLLARROW	Specifies the count of pels along the *y*-axis of a vertical scroll-bar arrow.
SV_CXBORDER	Specifies the count of pels along the *x*-axis of a window border.
SV_CYBORDER	Specifies the count of pels along the *y*-axis of a window border.
SV_CXDLGFRAME	Specifies the count of pels along the *x*-axis of a dialog frame.
SV_CYDLGFRAME	Specifies the count of pels along the *y*-axis of a dialog frame.
SV_CYTITLEBAR	Specifies the count of pels along the *y*-axis of a title-bar window.
SV_CXHSLIDER	Specifies the count of pels along the *x*-axis of a horizontal scroll-bar slider.
SV_CYVSLIDER	Specifies the count of pels along the *y*-axis of a vertical scroll-bar slider.
SV_CXMINMAXBUTTON	Specifies the width (in pels) of a minimize/maximize button.
SV_CYMINMAXBUTTON	Specifies the height (in pels) of a minimize/maximize button.
SV_CYMENU	Specifies the height (in pels) of a menu.
SV_CXFULLSCREEN	Specifies the count of pels along the *x*-axis of a maximized frame window's client window.
SV_CYFULLSCREEN	Specifies the count of pels along the *y*-axis of a maximized frame window's client window.
SV_CXICON	Specifies the count of pels along an icon's *x*-axis.
SV_CYICON	Specifies the count of pels along an icon's *y*-axis.
SV_CXPOINTER	Specifies the count of pels along the mouse pointer's *x*-axis.
SV_CYPOINTER	Specifies the count of pels along the mouse pointer's *y*-axis.
SV_DEBUG	Reserved.
SV_CURSORLEVEL	Specifies the cursor display count. The cursor is visible only when the display count is zero.

Value	Meaning
SV_POINTERLEVEL	Specifies the mouse-pointer display count. The mouse is visible only when the display count is zero.
SV_TRACKRECTLEVEL	Specifies the tracking rectangle display count. The tracking rectangle is visible only when the display count is zero.
SV_CTIMERS	Specifies the number of available timers.
SV_CXBYTEALIGN	Set by a device driver at initialization time to indicate any horizontal alignment that is more efficient for the driver.
SV_CYBYTEALIGN	Set by a device driver at initialization time to indicate any vertical alignment that is more efficient for the driver.
SV_CSYSVALUES	Specifies the number of system values.

The **WinQuerySysValue** function lets you read a system value, and the **WinSetSysValue** function lets you set a system value. Both functions specify the target system value with a system-value constant.

Setting a system value with **WinSetSysValue** does just that, and no more. Active processes are not automatically notified of the change. For example, if an application changes the dimensions of frame-window sizing borders by resetting SV_CXSIZEBORDER and SV_CYSIZEBORDER, currently existing frame windows do not automatically reflect the change. An application must send a WM_SYSVALUECHANGED message to notify other processes of a change to a system value. An application that relies on system values should either read them each time they are used or process WM_SYSVALUECHANGED messages.

When the system starts, the system values are set to default values embedded in the system code. With a few exceptions, the system does not store system values in the *os2.ini* file, so any changes that you make to system values are lost when the system shuts down.

The system values stored in *os2.ini* by Control Panel are SV_CXBORDER, SV_CYBORDER, SV_CURSORRATE, SV_ALARM and SV_DBLCLKTIME. These values are kept in the PM_ControlPanel section under the key names BorderWidth, CursorBlinkRate, and DoubleClickSpeed.

System-Value Data Types

For convenience, **WinQuerySysValue** and **WinSetSysValue** pass system values as long integers. However, many system values are actually short integers, and others are Boolean values. Each of these types can be cast to and from a long integer without losing data.

56.3 Using System Information

The following sections describe some specific ways to use MS OS/2 functions in your applications to change or retrieve system information.

56.3.1 The MS OS/2 Initialization File

You can read and write integers, strings, and binary data from the *os2.ini* file. You can also create new sections and keys and delete existing sections and keys. Since writing to the file can affect other applications, you should use care when changing and deleting values.

You can determine the size of a specific value by using the **WinQueryProfileSize** function. The function returns the size, in bytes, of the value corresponding to the given key name. This information is useful for allocating a buffer large enough to read in a key value. Once you know the size of the key value, you can read it by using the **WinQueryProfileData** function. The following code fragment reads the Window color value from the PM_Color section of the *os2.ini* file:

```
USHORT cb;
NPBYTE npb;

WinQueryProfileSize(hab, "PM_Color", "Window", &cb);
npb = WinAllocMem(hOurHeap, cb);
WinQueryProfileData (hab, "PM_Color", "Window", npb, &cb);
```

You can retrieve an integer from the initialization file by using the **WinQueryProfileInt** function, as shown in the following code fragment. The function assumes that the key-value data is a null-terminated character string representing an integer value in the range -32,768 through 32,767 (for example, strings like "12367" and "-5438").

```
sValue = WinQueryProfileInt(hab, "MySection", "MyKey", 0);
```

You can retrieve a null-terminated string from the initialization file by using the **WinQueryProfileString** function, as shown in the following code fragment:

```
CHAR ach[80];

WinQueryProfileString(hab, "MySection", "MyKey", NULL, ach, 80);
```

You can also retrieve a list of the section names in the *os2.ini* file by using the **WinQueryProfileString** function, specifying NULL for the section and key names. The function copies all the section names to the specified buffer. Each section name is null-terminated, and there is an extra null character at the end of the list. The following code fragment retrieves a list of section names:

```
CHAR achSectionNames[1024];

WinQueryProfileString(hab, NULL, NULL, NULL, achSectionNames, 1024);
```

Once you have a list of section names, you can use the names to generate a list of keys for a section name. To list the key names, you choose the section name and set the key name to NULL, as shown in the following code fragment:

```
CHAR achKeyNames[1024];

WinQueryProfileString(hab, "MySection", NULL, NULL, achKeyNames, 1024);
```

You can write values to an existing key name by using the **WinWriteProfileString** or **WinWriteProfileData** function. The following code fragment stores an integer as a null-terminated string:

```
WinWriteProfileString(hab, "MySection", "MyKey", "123");
```

Note Some users may change the file attribute of the *os2.ini* file to read-only. If so, you will need to change the attribute before writing to the file.

You can create a new key or a new section and key by using the **WinWriteProfileString** or **WinWriteProfileData** function. If the key you specify is not in the given section, or if the section does not exist, these functions create new entries. For example, the following code fragment creates a new key entry in MySection:

```
WinWriteProfileString(hab, "MySection", "MyNewKey", "123");
```

You can also delete an existing key by using the **WinWriteProfileString** or **WinWriteProfileData** function. In this case, you specify NULL for the key value, as shown in the following code fragment. The function deletes the existing key value, effectively deleting the entry.

```
WinWriteProfileString(hab, "MySection", "MyKey", NULL);
```

Note that this is different from setting the key value to a single zero byte, as would happen if an application called **WinWriteProfileString** with a pointer to an empty string.

You can delete all keys in a particular *os2.ini* section by invoking the **WinWriteProfileString** or **WinWriteProfileData** function with the key name set to NULL, as shown in the following code fragment:

```
WinWriteProfileString(hab, "MySection", NULL, NULL);
```

56.3.2 The Shell

Most applications use the FCF_TASKLIST style when creating their main windows. This style automatically adds the application to the switch list. For those applications that do not use the FCF_TASKLIST style, you can use the shell functions to add the application to the list.

You can add a program to the switch list by using the **WinAddSwitchEntry** function. Before calling the function, you must fill the fields of a **SWCNTRL** structure. The following code fragment adds the program named My Application to the switch list:

```
SWCNTRL swctl;
PID     pid;
TID     tid ;
HSWITCH hsw;

/* Obtain the process and thread identifiers. */

WinQueryWindowProcess(hwnd, &pid, &tid);

/* Fill in the switch-control data structure. */

swctl.hwnd = hwnd;
swctl.hwndIcon = (HWND) WinSendMsg(hwnd, WM_QUERYICON, NULL, NULL);
swctl.hprog = (HPROGRAM) NULL;
swctl.idProcess = pid;
swctl.idSession = NULL;
swctl.uchVisibility = SWL_VISIBLE;
swctl.fbJump = SWL_JUMPABLE;
strcpy(swctl.szSwtitle, "My Application");

hsw = WinAddSwitchEntry(&swctl);
```

You can use the switch-list handle returned by **WinAddSwitchEntry** to change the switch-list entry. For example, you can add the name of the file your application currently has open by using the **WinChangeSwitchEntry** function and specifying the switch-list handle, as shown in the following code fragment. You must fill a **SWCNTRL** structure or use the structure you used to create the switch-list entry.

```
CHAR szCurrentFile[128];     /* buffer for current filename */
SWCNTRL swctl;               /* switch-control structure    */
HSWITCH hsw;                 /* switch-list handle          */

strcpy(swctl.szSwtitle, "My Application: ");
strcat(swctl.szSwtitle, szCurrentFile);

WinChangeSwitchEntry(hsw, &swctl);
```

You can remove the switch-list entry for your application by using the **WinRemoveSwitchEntry** function.

56.3.3 System Colors

You can use a system color for elements of your own client window. Many applications use the application workspace color as the color for the client-window background. To use this color, you simply use the SYSCLR_APPWORKSPACE index when filling the client window. The following code fragment shows how this is done:

```
LONG iAWSColor = SYSCLR_APPWORKSPACE;
RECTL rcl;

case WM_PAINT:
    hps = WinBeginPaint(hwnd, NULL, &rcl);
    WinFillRect(hps, &rcl, iAWSColor);
    WinEndPaint(hps);
    return (OL);
```

You can use the **WinQuerySysColor** function to retrieve the RGB color value for a system color, as shown in the following code fragment. This information is useful if you want to apply the color to another color index or change the color in some way.

```
COLOR clr;

clr = WinQuerySysColor(HWND_DESKTOP, SYSCLR_APPWORKSPACE, OL);
```

You can use the **WinSetSysColors** function to set a system color. The following code fragment inverts the application workspace color for all applications in the system:

```
COLOR clr;

clr = ~WinQuerySysColor(HWND_DESKTOP, SYSCLR_APPWORKSPACE, OL)
    & OxOOFFFFFF;

WinSetSysColors(HWND_DESKTOP, OL, LCOLF_CONSECRGB,
    SYSCLR_APPWORKSPACE, 1L, &clr);
```

You can also use the **WinSetSysColors** function to set several system colors at once. As with the **GpiCreateLogColorTable** function, there are two ways to specify the RGB values: as an array of consecutive RGB values or as an array of system-color and index RGB-value pairs. The first method is useful when you want to set the entire group of system colors or any consecutive subset. The second method is useful when you want to set nonconsecutive subsets of the system colors. The following code fragment sets the system colors using a consecutive array of RGB values:

```
COLOR aclr[SYSCLR_CSYSCOLORS];

aclr[0] = 0x00FF0020L;
aclr[1] = 0x00767676L;
aclr[2] = 0x001A1A1AL;
aclr[3] = 0x003ABA3AL;
aclr[4] = 0x00FFFFFFL;
aclr[5] = 0x0000FFFFL;

WinSetSysColors(
    HWND_DESKTOP,                /* desktop window            */
    0L,                          /* color options             */
    LCOLF_CONSECRGB,             /* consecutive RGB values    */
    SYSCLR_WINDOWSTATICTEXT,     /* start with window static text */
    6L,                          /* six elements in the array */
    aclr);                       /* array of color values     */
```

The following code fragment inverts the colors for the background, active border, and active title bar. It sets the colors using index-RGB pairs. Note that the **WinSetSysColors** *cclr* parameter is set to count the number of values passed using the *pclr* parameter, which is exactly twice the number of system colors being set. Note also that the high byte of an RGB color value must be cleared to zero.

```
SHORT i;
COLOR aclrIndexRgb[6] = { SYSCLR_BACKGROUND, 0, SYSCLR_ACTIVEBORDER, 0,
                       SYSCLR_ACTIVETITLE, 0 };

for (i = 0; i < 6; i += 2)
    aclrIndexRgb[i + 1] = ~WinQuerySysColor(HWND_DESKTOP,
        aclrIndexRgb[i], 0L) & 0x00FFFFFF;

WinSetSysColors(HWND_DESKTOP, 0L, LCOLF_INDRGB, 0L, 6L, aclrIndexRgb);
```

You can reset the system colors to their default values by calling the **WinSetSysColors** function with the *flOptions* parameter set to LCOL_RESET, as shown in the following code fragment:

```
WinSetSysColors(HWND_DESKTOP, LCOL_RESET, 0L, 0L, 0L, NULL);
```

56.3.4 System Values

You can retrieve a system value by using the **WinQuerySysValue** function. The function always returns a 32-bit value, even if the system value is smaller. You can cast the function result to a more appropriate type. The following code fragment shows two examples of the **WinQuerySysValue** function:

```
SHORT cxSizeBorder;
BOOL  fRightButton;

/* Retrieve the width of a sizing-border vertical element. */

cxSizeBorder = (SHORT) WinQuerySysValue(HWND_DESKTOP, SV_CXSIZEBORDER);

/* Are the mouse buttons switched? */

fRightButton = (BOOL) WinQuerySysValue(HWND_DESKTOP, SV_SWAPBUTTON);
```

You can set a system value by using the **WinSetSysValue** function, as shown in the following code fragment. The value is passed as a long integer.

```
/* Play with the sizing border. */

WinSetSysValue (HWND_DESKTOP, SV_CXSIZEBORDER, lNewWidth);
WinSetSysValue (HWND_DESKTOP, SV_CYSIZEBORDER, lNewWidth);
```

You will usually follow such a call by sending the WM_SYSVALUECHANGED message to the frame windows in the system. You can use the **WinBroadcastMsg** function to send the message, as shown in the following code fragment:

```
WinBroadcastMsg(
    HWND_DESKTOP,                      /* desktop window        */
    WM_SYSVALUECHANGED,                /* system value changed  */
    MPFROMSHORT(SV_CXSIZEBORDER),      /* first system value    */
    MPFROMSHORT(SV_CYSIZEBORDER),      /* last system value     */
    BMSG_FRAMEONLY | BMSG_SEND);       /* send to frame windows */
```

56.4 Summary

The following sections describe the MS OS/2 functions you can use to retrieve and manage system information.

56.4.1 The MS OS/2 Initialization File

MS OS/2 provides the following initialization-file functions:

WinQueryProfileData Retrieves data from an *os2.ini* file as a stream of raw bytes. The function retrieves data from key entry unless a key name or section name is not given. If the names are not given, the function enumerates all the section names or key names in the file.

WinQueryProfileInt Retrieves ASCII-character-string data from an *os2.ini* key value and converts it to a 2-byte signed integer. The key value is assumed to be a number in the range -32,768 through 32,767, expressed as a null-terminated character string.

WinQueryProfileSize Retrieves the size, in bytes, of an *os2.ini* key value. For null-terminated strings, the size includes the terminal null character.

WinQueryProfileString Retrieves *os2.ini* key values, key names, and section names. The function reads a key value unless a key name or section name is not given. If the names are not given, the function enumerates all the section names or key names in the file.

WinWriteProfileData Writes data to an *os2.ini* key entry unless a key name or section name does not exist. If the names do not exist, the function creates a new entry having the given section name and key name. If no key-value data is given, the function deletes the specified entry.

WinWriteProfileString Writes a null-terminated string to an *os2.ini* key entry unless a key name or section name does not exist. If the names do not exist, the function creates a new entry having the given section name and key name. If no key-value data is given, the function deletes the specified entry.

56.4.2 The Shell

MS OS/2 provides the following shell functions:

WinAddProgram Adds an application to the program list. The function returns an **HPROGRAM** program handle, used to specify the application's program-list entry in calls to the **WinQueryDefinition** and **WinQueryProgramTitles** functions. Before calling **WinAddProgram**, an application must put meaningful data into at least three fields of a **PIBSTRUCT** structure: **progt**, **szTitle**, and **szExecutable**. The other fields, if not filled, should be set to zero.

WinAddSwitchEntry Adds a process to the switch list. An application should do this after finishing other initialization activities, just before entering its main message loop. The function takes a pointer to a **SWCNTRL** structure and returns an **HSWITCH** switch-list-entry handle.

WinChangeSwitchEntry Changes a process's switch-list entry. An application should do this when its main window's title-bar string changes, thereby keeping its switch entry title accurate. An **HSWITCH** switch-list-entry handle, as returned by the **WinAddSwitchEntry** function, specifies the target switch-list entry.

WinCreateGroup Creates a program-list group. The function returns an **HPROGRAM** group handle, used to specify the group in calls to the **WinAddProgram**, **WinQueryDefinition**, and **WinQueryProgramTitles** functions.

WinQueryDefinition Retrieves program-list information about an installed application or program-list group. The function fills in the relevant fields of a **PIBSTRUCT** structure.

WinQueryProgramTitles Obtains program-title information about one or more installed applications or a program-list group. The function returns the information in one or more **PROGRAMENTRY** structures.

WinQueryTaskTitle Retrieves the title under which a particular process is registered with the switch list. The process is specified by using the session identifier. The system determines this value if you specify zero. The system returns the title in a buffer whose size is specified by the function's third parameter.

WinRemoveSwitchEntry Removes a process's switch list entry. An application should do this during its shutdown activities. An **HSWITCH** switch-list-entry handle, as returned by the **WinAddSwitchEntry** function, specifies the target switch-list entry.

56.4.3 System Colors and Values

MS OS/2 provides the following system-color and system-value functions and messages:

56.4.3.1 Functions

An application can use the following functions to query and set system colors and values:

WinQuerySysColor Retrieves the RGB value assigned to a particular system color. The system color can be specified with a SYSCLR constant.

WinQuerySysValue Retrieves a particular system value. The function returns the system value, or 0L if either the desktop-window or the system-value selector parameter is invalid. Note that a system value might actually be 0L, so it is important to use valid parameters.

WinSetSysColors Sets one or more system colors. Function parameters are similar to those of the **GpiCreateLogColorTable** function. You can set a single color, a set of consecutive colors, or a set of nonconsecutive colors. Note that the setting is not permanent—that is, when the machine is restarted, it reverts to the system color values stored in the PM_Colors section of the *os2.ini* file.

WinSetSysValue Sets a particular system value. The function returns TRUE if you send it a valid system-value selector parameter, FALSE otherwise. In most cases, you will follow this call by broadcasting a WM_SYSVALUECHANGED message.

56.4.3.2 Messages

An application can use the following messages to query and set system colors and values:

WM_SYSCOLORCHANGE Sent while the system is executing the **WinSetSysColors** function. An application that ties particular variables to the system colors may want to process this message. The default window procedure and the default frame-window procedure do nothing upon receipt of this message.

WM_SYSVALUECHANGED Sent when one or more system values are changed. The *mp1* and *mp2* parameters are SV values and together denote a contiguous range of system values. If only one value has changed, the two parameters are identical. A handler should return 0L. The default window procedure ignores this message. The frame-window procedure handles the message by checking to see if the contiguous range includes SV_CXSIZEBORDER and SV_CYSIZEBORDER. If so, it sends the window a WM_SETBORDERSIZE message so that it will adjust its sizing borders.

Index

A

Accelerator tables, 237–243
Advanced video I/O, 647–657
Arc primitives, 425–437
Area primitives, 493–504
Atom table, 707–713

B

Base type, 10
Bitmap, 511–526
Bit mask, 39
BM_CLICK, 165
BM_LEAVEALONE, 473, 489
BM_OR, 473
BM_OVERPAINT, 473
BM_QUERYCHECK, 165
BM_QUERYCHECKINDEX, 165
BM_QUERYHILITE, 165
BM_SETCHECK, 88, 165
BM_SETDEFAULT, 165
BM_SETHILITE, 165
BM_XOR, 473
Button control, 157–168

C

Character primitives, 439–466
C language, 8, 40
Class, window, 91–98
Clipboard, 331–344
Clipping, 539–547
Color mode, 467–477
Command line, 37
Control
  button, 157–168
  entry-field, 169–178
  list-box, 179–191
  scroll-bar, 201–211
  static, 193–199
  title-bar, 151–155
Control window, 143–149
Coordinate space, 403–423
CS_CLIPCHILDREN, 95
CS_CLIPSIBLINGS, 95, 96
CS_FRAME, 95, 96
CS_HITTEST, 95, 96
CS_MOVENOTIFY, 64, 94, 95
CS_PARENTCLIP, 95, 96
CS_PUBLIC, 95, 96, 97
CS_SAVEBITS, 95
CS_SIZEREDRAW, 65, 95, 96, 138
CS_SYNCPAINT, 55, 94, 95, 96, 268
Cursor, 295–299

D

Dev functions, 25
DevCloseDC, 273, 275, 397, 553, 554
DevEscape, 307, 312, 317
Device context, 391–401
Device monitor, 703–705
DevOpenDC, 273, 274, 275, 306, 307, 309, 313, 397, 421, 516, 519, 552, 554
DevPostDeviceModes, 307, 313, 314
DevQueryCaps, 310, 311, 316, 398, 471, 507, 652
Dialog window, 245–262
Dos functions, 25
DosAllocHuge, 35, 36, 605, 606, 607, 609
DosAllocSeg, 35, 36, 321, 322, 323, 329, 334, 604, 607, 609
DosAllocShrSeg, 608
DosBeep, 35, 42
DosBufReset, 35, 625
DosCallNmPipe, 681
DosCaseMap, 35, 36
DosChDir, 35, 624
DosChgFilePtr, 35, 625, 632
DosClose, 35, 520, 626, 629, 630, 634, 681, 687
DosCloseSem, 677
DosConnectNmPipe, 680, 681, 687
DosCreateCSAlias, 35, 36, 37, 610
DosCreateQueue, 682, 688
DosCreateSem, 677, 684
DosCreateThread, 597
DosCwait, 590, 594, 595
DosDelete, 35, 624
DosDevConfig, 35
DosDevIOCtl, 35, 36, 634
DosDisConnectNmPipe, 681, 687
DosDupHandle, 35, 626
DosErrClass, 35
DosError, 35, 36
DosExecPgm, 35, 36, 589, 593, 594, 595, 596
DosExit, 35, 36, 588, 595, 596, 597
DosExitList, 588, 596, 685
DosFileLocks, 35, 625
DosFindClose, 35, 624, 632
DosFindFirst, 35, 36, 624, 631, 632
DosFindNext, 35, 36, 624, 631, 632
DosFlagProcess, 679
DosFreeModule, 372, 620
DosFreeSeg, 35, 37, 329, 351, 592, 605, 606, 608, 610
DosFSRamSemClear, 685
DosGetCollate, 35, 37
DosFSRamSemRequest, 685
DosGetCp, 652
DosGetCtryInfo, 35, 37
DosGetDateTime, 10, 35, 40
DosGetDBCSEv, 35, 37
DosGetEnv, 35, 39, 590
DosGetHugeShift, 35, 605, 606
DosGetInfoSeg, 695
DosGetMachineMode, 35
DosGetMessage, 35, 37
DosGetModHandle, 620
DosGetModName, 620
DosGetPID, 598
DosGetProcAddr, 94, 620
DosGetPrty, 589, 598
DosGetSeg, 609
DosGetShrSeg, 608
DosGetVersion, 35
DosGiveSeg, 351, 609
DosHoldSignal, 35, 37, 686
DosInsMessage, 35, 37
DosKillProcess, 595, 679
DosLoadModule, 94, 372, 373, 591, 620
DosLockSeg, 608
DosMakeNmPipe, 680, 687
DosMakePipe, 679, 680, 687
DosMemAvail, 606
DosMkDir, 35, 624
DosMonClose, 705
DosMonOpen, 705
DosMonRead, 705
DosMonReg, 705
DosMonWrite, 705
DosMove, 35
DosMuxSemWait, 678, 682, 683
DosNewSize, 35
DosOpen, 35, 37, 520, 625, 626, 627, 628, 631, 633, 680, 681, 688

DosOpenQueue, 688
DosOpenSem, 677, 683, 684
DosPeekNmPipe, 681
DosPeekQueue, 683
DosPTrace, 44
DosPurgeQueue, 683
DosPutMessage, 35, 38
DosQCurDir, 35, 624
DosQCurDisk, 35, 624
DosQFHandState, 35, 626
DosQFileInfo, 35, 624
DosQFileMode, 35, 624
DosQFSInfo, 35, 624
DosQHandType, 626
DosQNmPHandState, 682, 688
DosQNmPipeInfo, 682
DosQNmPipeSemState, 682
DosQVerify, 35
DosRead, 35, 38, 520, 625, 626, 628, 629, 630, 631, 632, 634, 670, 672, 679, 681, 687, 688
DosReadAsync, 625, 629
DosReadQueue, 592, 683, 689
DosReallocHuge, 35, 38, 606
DosReallocSeg, 35, 38, 605, 608, 610
DosResumeThread, 588, 598
DosRmDir, 35, 624
DosScanEnv, 40, 590
DosSearchPath, 590
DosSelectDisk, 35, 624
DosSelectSession, 591, 592
DosSemClear, 678, 684
DosSemRequest, 678
DosSemSet, 678, 684
DosSemSetWait, 678
DosSemWait, 678, 682, 684, 685
DosSendSignal, 679
DosSetCp, 652
DosSetDateTime, 35
DosSetFHandState, 35, 38
DosSetFileInfo, 35, 624
DosSetFileMode, 35, 624
DosSetFSInfo, 35, 624
DosSetMaxFH, 625
DosSetNmPHandState, 682
DosSetNmPipeSem, 682
DosSetPrty, 589, 592, 598
DosSetSession, 591, 592
DosSetSigHandler, 35, 38, 679, 686
DosSetVec, 35, 38
DosSetVerify, 35
DosSizeSeg, 605
DosSleep, 35, 598, 684, 695, 696
DosStartSession, 591, 592
DosStopSession, 591, 592
DosSubAlloc, 35, 610
DosSubFree, 35, 610
DosSubSet, 35, 609, 610
DosSuspendThread, 588, 598
DosTimerAsync, 696
DosTimerStart, 696
DosTimerStop, 696
DosTransactNmPipe, 681
DosUnlockSeg, 608
DosWaitNmPipe, 681
DosWrite, 31, 35, 41, 520, 616, 625, 626, 628, 629, 630, 631, 632, 634, 640, 642, 679, 681, 687, 688
DosWriteAsync, 625, 629
DosWriteQueue, 683
Drawing, 263–286
Dynamic data exchange, 345–361
Dynamic linking, 23, 613–620

E

EM_CLEAR, 176
EM_COPY, 176
EM_CUT, 176
EM_MEMERROR, 176
EM_PASTE, 176
EM_QUERYCHANGED, 176
EM_QUERYFIRSTCHAR, 175
EM_QUERYSEL, 173, 176
EM_SETFIRSTCHAR, 175
EM_SETSEL, 176, 262
EM_SETTEXTLIMIT, 173, 176
EM_TIMEOUT, 685
Entry-field control, 169–178

F

FAPI, 32
FCF_ACCELTABLE, 131, 133, 134, 241
FCF_BORDER, 130
FCF_DLGBORDER, 130
FCF_HORZSCROLL, 128, 130, 208
FCF_ICON, 131, 133, 134, 136
FCF_MAXBUTTON, 129, 130
FCF_MENU, 128, 130, 131, 133, 134, 222
FCF_MINBUTTON, 129, 130
FCF_MINMAX, 129, 130, 131
FCF_MOUSEALIGN, 131
FCF_NOBYTEALIGN, 64, 131
FCF_NOMOVEWITHOWNER, 131
FCF_SCREENALIGN, 131
FCF_SHELLPOSITION, 130, 131, 132
FCF_SIZEBORDER, 129, 130, 131
FCF_STANDARD, 131, 133, 134, 140, 241
FCF_SYSMENU, 128, 130, 131
FCF_SYSMODAL, 131
FCF_TASKLIST, 130, 131, 726
FCF_TITLEBAR, 128, 130, 131
FCF_VERTSCROLL, 128, 130, 208
File system, 23, 621–635
Font, 439–466
Frame window, 125–142
FS_ACCELTABLE, 132, 133, 134, 138, 226
FS_BORDER, 94, 132
FS_DLGBORDER, 132
FS_ICON, 132, 133, 134, 138, 270
FS_MOUSEALIGN, 132
FS_NOBYTEALIGN, 133
FS_NOMOVEWITHOWNER, 62, 133
FS_SCREENALIGN, 132
FS_SHELLPOSITION, 133
FS_SIZEBORDER, 132
FS_STANDARD, 133, 134
FS_SYSMODAL, 133
FS_TASKLIST, 133
Full-screen programs, 29
Functions
  Dev, 25
  Dos, 25
  Gpi, 26
  Kbd, 26
  Mou, 26
  notational conventions, 11
  Vio, 26
  Win, 26

G

Gpi functions, 26
GpiAssociate, 57, 274, 394, 397, 553, 554
GpiBeginArea, 499, 500
GpiBeginElement, 394, 482, 496
GpiBeginPath, 413, 482, 484, 490, 491, 546
GpiBitBlt, 316, 518, 519, 520, 522
GpiBox, 304, 415, 481, 482, 484, 496
GpiCallSegment, 417
GpiCallSegmentMatrix, 394, 482, 496, 565, 571

GpiCharString, 18, 21, 461, 482
GpiCharStringAt, 461, 482
GpiCharStringPos, 21, 461, 482
GpiCharStringPosAt, 461, 482
GpiCloseFigure, 482
GpiCloseSegment, 394, 567, 571, 572
GpiCombineRegion, 531, 534, 544
GpiComment, 482, 496
GpiConvert, 310, 311
GpiCorrelateChain, 394, 568, 572
GpiCorrelateFrom, 394, 568
GpiCorrelateSegment, 394, 568
GpiCreateBitmap, 515, 516, 517, 522
GpiCreateClipRegion, 544
GpiCreateLogColorTable, 471, 474, 727
GpiCreateLogFont, 22, 457, 458, 459, 460, 482, 501
GpiCreatePS,. 271, 273, 274, 275, 393, 394, 397, 398, 417, 421, 452, 554
GpiCreateRegion, 487, 529, 530, 531, 533, 544, 545
GpiDeleteElement, 394, 482, 497, 570, 572
GpiDeleteElementRange, 394, 482, 497, 570, 572
GpiDeleteElementsBetweenLabels, 394, 482, 497, 570
GpiDeleteSegment, 394, 567
GpiDeleteSegments, 394, 568
GpiDeleteSetId, 482
GpiDestroyPS, 57, 271, 273, 275, 396
GpiDestroyRegion, 534
GpiDrawChain, 394, 415, 553, 565, 566, 572
GpiDrawDynamics, 394, 565
GpiDrawFrom, 394, 415, 553, 565, 566
GpiDrawSegment, 22, 394, 415, 553, 565, 566
GpiElement, 394, 482, 497, 570
GpiEndArea, 496, 497
GpiEndElement, 394, 482, 497
GpiEndPath, 413, 482, 484, 490, 491, 546
GpiEqualRegion, 534
GpiErase, 394
GpiErrorSegmentData, 394
GpiExcludeClipRectangle, 544
GpiFillPath, 485, 491
GpiFullArc, 428, 429, 430, 434, 481, 482, 484, 496, 497
GpiGetData, 394, 482, 497, 570
GpiImage, 518, 519
GpiIntersectClipRectangle, 544
GpiLabel, 394, 482, 497, 569
GpiLine, 19, 428, 432, 472, 482, 483, 490, 496, 497, 552, 563, 564
GpiLoadBitmap, 517, 523
GpiLoadFonts, 22, 457, 458
GpiLoadMetaFile, 554, 556
GpiMarker, 482, 509
GpiMove, 20, 428, 429, 432, 461, 482, 483, 490, 491, 497
GpiOffsetElementPointer, 394, 482, 497, 569, 572
GpiOffsetRegion, 535
GpiOpenSegment, 394, 413, 567, 571, 572
GpiPaintRegion, 529, 530, 532, 535
GpiPartialArc, 428, 429, 430, 482, 483, 496, 497
GpiPlayMetaFile, 553, 554, 557
GpiPointArc, 428, 429, 482, 496, 497
GpiPolyFillet, 428, 482, 496, 497
GpiPolyFilletSharp, 428, 482, 496, 497
GpiPolyLine, 428, 472, 482, 483, 484, 496, 497, 509
GpiPolyMarker, 483, 509
GpiPolySpline, 428, 436, 483, 496, 497
GpiPop, 483, 497
GpiPtInRegion, 535
GpiPutData, 395, 483, 497, 570
GpiQueryArcParams, 430, 483, 497
GpiQueryAttrMode, 483, 497
GpiQueryAttrs, 432, 452, 490, 497, 500, 501, 508, 509, 519, 533
GpiQueryBackColor, 497
GpiQueryBackMix, 497
GpiQueryBitmapBits, 520
GpiQueryBoundaryData, 395, 497
GpiQueryCharAngle, 452, 497
GpiQueryCharBox, 452, 497
GpiQueryCharDirection, 497
GpiQueryCharMode, 454, 497
GpiQueryCharSet, 497
GpiQueryCharShear, 453, 497
GpiQueryCharStringPos, 497
GpiQueryCharStringPosAt, 497
GpiQueryClipBox, 497, 545
GpiQueryClipRegion, 497
GpiQueryColor, 498
GpiQueryColorData, 475, 498
GpiQueryColorIndex, 475, 498
GpiQueryCp, 451, 498
GpiQueryCurrentPosition, 483, 498
GpiQueryDefaultViewMatrix, 312, 418, 498
GpiQueryDefCharBox, 498
GpiQueryDevice, 498
GpiQueryDeviceBitmapFormats, 498, 515, 516
GpiQueryDrawControl, 395
GpiQueryDrawingMode, 395, 567
GpiQueryEditMode, 395, 498, 569
GpiQueryElement, 395, 498, 570
GpiQueryElementPointer, 395, 498, 569
GpiQueryElementType, 395, 498
GpiQueryFontFileDescriptions, 498
GpiQueryFontMetrics, 452, 498
GpiQueryFonts, 457, 458, 459, 460, 498
GpiQueryGraphicsField, 498
GpiQueryInitialSegmentAttrs, 395, 498, 566, 571
GpiQueryKerningPairs, 448, 498
GpiQueryLineEnd, 498
GpiQueryLineJoin, 498
GpiQueryLineType, 498
GpiQueryLineWidth, 498
GpiQueryLineWidthGeom, 498
GpiQueryLogColorTable, 470, 498
GpiQueryMarker, 498, 508
GpiQueryMarkerBox, 498
GpiQueryMarkerSet, 498, 508
GpiQueryMetaFileBits, 553
GpiQueryMetaFileLength, 553
GpiQueryMix, 498
GpiQueryModelTransformMatrix, 415, 498
GpiQueryNearestColor, 498
GpiQueryNumberSetIds, 498
GpiQueryPage, 419
GpiQueryPageViewport, 498
GpiQueryPattern, 498, 500, 533
GpiQueryPatternRefPoint, 498
GpiQueryPatternSet, 498
GpiQueryPel, 499
GpiQueryPickAperturePosition, 395, 499, 568
GpiQueryPickApertureSize, 395, 499, 568
GpiQueryPS, 418
GpiQueryRealColors, 470, 499
GpiQueryRegionBox, 499
GpiQueryRegionRects, 499, 536

GpiQueryRGBColor, 499
GpiQuerySegmentAttrs, 395, 499, 567
GpiQuerySegmentNames, 395, 499, 567
GpiQuerySegmentPriority, 395, 499
GpiQuerySegmentTransformMatrix, 416, 499
GpiQuerySetIds, 499
GpiQueryStopDraw, 395, 499
GpiQueryTag, 395, 499, 568
GpiQueryViewingLimits, 499
GpiQueryViewingTransformMatrix, 418, 499
GpiQueryWidth, 499
GpiRemoveDynamics, 395, 565
GpiResetBoundaryData, 395
GpiRestorePS, 396
GpiSaveMetaFile, 554, 556
GpiSavePS, 395
GpiSetArcParams, 430, 483, 499
GpiSetAttrMode, 483, 499
GpiSetAttrs, 20, 432, 452, 471, 475, 483, 490, 499, 500, 501, 508, 509, 519, 533
GpiSetBackColor, 475
GpiSetBitmap, 316, 520
GpiSetBitmapBits, 520
GpiSetBitmapId, 500, 517, 522
GpiSetCharAngle, 452, 483
GpiSetCharBox, 452, 483
GpiSetCharDirection, 483
GpiSetCharMode, 454, 483
GpiSetCharSet, 22, 458, 459, 460, 483
GpiSetCharShear, 453, 483
GpiSetClipPath, 413, 487, 491, 546
GpiSetClipRegion, 487, 531, 545
GpiSetColor, 20, 475, 483, 499
GpiSetCp, 451, 483
GpiSetCurrentPosition, 432, 483, 499
GpiSetDefaultViewMatrix, 312, 418, 421
GpiSetDrawControl, 395
GpiSetDrawingMode, 395, 553, 567
GpiSetEditMode, 395, 483, 499, 569, 572
GpiSetElementPointer, 395, 483, 499, 569, 572
GpiSetElementPointerAtLabel, 395, 483, 499, 569, 572
GpiSetInitialSegmentAttrs, 395, 571
GpiSetLineEnd, 483, 484, 487, 490, 499
GpiSetLineJoin, 483, 484, 488, 499
GpiSetLineType, 483, 499
GpiSetLineWidth, 483, 499
GpiSetLineWidthGeom, 484, 487, 490
GpiSetMarker, 483, 508
GpiSetMarkerBox, 483, 508
GpiSetMarkerSet, 483, 508
GpiSetMetaFileBits, 553
GpiSetMix, 483, 499
GpiSetModelTransformMatrix, 415, 483, 499
GpiSetPageViewport, 419
GpiSetPattern, 500, 501, 522, 533
GpiSetPatternSet, 500, 501
GpiSetPickAperturePosition, 395, 568
GpiSetPickApertureSize, 395, 568, 572
GpiSetSegment, 414
GpiSetSegmentAttrs, 395, 567, 568
GpiSetSegmentPriority, 395, 565
GpiSetSegmentTransformMatrix, 416, 499
GpiSetStopDraw, 395
GpiSetTag, 395, 568
GpiSetViewingLimit, 414
GpiSetViewingTransformMatrix, 418
GpiStrokePath, 484, 490
GpiWCBitBlt, 316, 518, 519, 520, 522, 523
Graphics, retained, 561–579
Graphics programming interface
  device context, 19
  drawing, 21
  graphics
    primitives, 19
    retained, 22
    segments, 22
    tools, 20
  metafile, 22
  presentation space, 19

H

Header files, 40
Heap, 319–330
Help, 377–386
Hook, 363–375

I

Icon, mouse, 287–294
Input
  keyboard, 109–124
  mouse, 109–124
  video, 637–645, 647–657
Interprocess communication, 24, 675–691

K

Kbd functions, 26
KbdCharIn, 35, 38, 640, 669, 670, 671, 672
KbdClose, 673
KbdFlushBuffer, 35, 38, 669, 673
KbdFreeFocus, 673
KbdGetFocus, 669, 673
KbdGetStatus, 35, 38
KbdOpen, 669, 673
KbdPeek, 35, 38, 669, 671
KbdSetStatus, 35, 38, 669, 673
KbdStringIn, 35, 38, 39, 669, 672
Keyboard, 667–674
  full-screen operation, 24
  input, 109–124

L

Line primitives, 425–437
List-box control, 179–191
LM_DELETEALL, 184, 189
LM_DELETEITEM, 183, 184, 188
LM_INSERTITEM, 183, 188
LM_QUERYITEMCOUNT, 188
LM_QUERYITEMHANDLE, 189
LM_QUERYITEMTEXT, 186, 189
LM_QUERYITEMTEXTLENGTH, 189
LM_QUERYSELECTION, 185, 189
LM_QUERYTOPINDEX, 188
LM_SEARCHSTRING, 189
LM_SELECTITEM, 188
LM_SETITEMHANDLE, 189
LM_SETITEMHEIGHT, 185, 189
LM_SETITEMTEXT, 189
LM_SETTOPINDEX, 188

M

Marker primitives, 505–510
Memory management, 23
Memory manager, 601–611
Menu, 213–236

Messages, 77–90
Metafile, 549–559
Mix mode, 467–477
MM_DELETEITEM, 227, 228, 229
MM_INSERTITEM, 219, 227, 228, 229
MM_QUERYITEM, 217, 219, 227
MM_QUERYITEMATTR, 217, 226, 227
MM_QUERYITEMTEXT, 227
MM_REMOVEITEM, 229
MM_SETITEM, 217, 219, 227
MM_SETITEMATTR, 217, 226, 227
MM_SETITEMTEXT, 227
Mou functions, 26
MouClose, 663
MouDrawPtr, 661, 663
MouFlushQue, 661, 665
MouGetEventMask, 661
MouGetNumButtons, 665
MouGetNumMickeys, 661
MouGetNumQueEl, 661, 665
MouGetPtrPos, 662
MouGetPtrShape, 662
MouOpen, 661, 662, 663
MouReadEventQue, 661, 664
MouRemovePtr, 662, 663
Mouse, 659–666
  full-screen operation, 24
  icon, 287–294
  input, 109–124
  pointer, 287–294
MouSetEventMask, 661, 665
MouSetPtrPos, 662
MouSetPtrShape, 662
MouSetScaleFact, 654
MS OS/2
  functions, 25, 26
  initialization file, 717, 725, 730
  introduction, 7–11
  naming conventions, 9
  overview, 13–26
  programming models, 27–44
Multitasking, 22

N
Naming conventions, 9
Notational conventions, 11

O
Output
  advanced video, 647–657
  video, 637–645

P
Painting, 263–278
Path, 479–492
Pointer, mouse, 287–294
Prefix type, 10
Presentation Manager
  applications, 30
  device-independent graphics, 16
  overview, 15
  queued input, 16
  shared resources, 16
Presentation space, 391–401
Primitives
  arc, 425–437
  area, 493–504
  character, 439–466
  line, 425–437
  marker, 505–510
Printing, 301–317
Procedure, window, 99–107
Programming models
  bit mask, 39
  command line, 37
  FAPI, 32
  full-screen, 29
  header files, 40
  Presentation Manager, 30
  shared resources, 40
  structures, 38
Process, 585–596

R
Region, 527–537
Retained graphics, 561–579

S
Scroll-bar control, 201–211
SBM_QUERYPOS, 204, 209
SBM_QUERYRANGE, 210
SBM_SETPOS, 204, 209
SBM_SETSCROLLBAR, 204, 209
Segment, 561–579
Session, 585–599
Shell, 718, 726, 730
SM_QUERYHANDLE, 195, 196, 197
SM_SETHANDLE, 195, 196, 197
Static control, 193–199
Structures, 38
System colors, 719, 727, 730
System information, 715–731
System services
  dynamic linking, 23
  file system, 23
  interprocess communication, 24
  keyboard operation, 24
  memory management, 23
  mouse operation, 24
  multitasking, 22
  video operation, 24
System values, 721, 728, 730

T
TBM_QUERYHILITE, 155
TBM_SETHILITE, 155
Thread, 585–592, 597–599
Timer, 693–702
Title-bar control, 151–155
Transformation, 403–423

V
Video
  advanced input and output, 647–657
  full-screen operation, 24
  input and output, 637–645
Vio functions, 26
VioCreateLogFont, 651
VioCreatePS, 649
VioGetBuf, 35
VioGetConfig, 642
VioGetCurPos, 35, 639, 642
VioGetCurType, 35, 642
VioGetDeviceCellSize, 652
VioGetFont, 639
VioGetMode, 35, 639
VioGetPhysBuf, 35
VioQueryFonts, 652
VioQuerySetIds, 652
VioReadCellStr, 35, 643
VioReadCharStr, 35
VioScrLock, 35, 39
VioScrollDn, 35, 640, 643
VioScrollLf, 35, 640, 643
VioScrollRt, 35, 640, 643
VioScrollUp, 35, 640
VioScrUnLock, 35
VioSetAnsi, 643
VioSetCurPos, 35, 639, 642
VioSetCurType, 35, 639, 642

VioSetDeviceCellSize, 649, 652
VioSetFont, 639
VioSetMode, 35, 639, 652
VioSetOrg, 650
VioShowBuf, 35, 650
VioShowPS, 650
VioWrtCellStr, 35, 639, 641
VioWrtCharStr, 35, 639
VioWrtCharStrAtt, 35, 639, 641
VioWrtNAttr, 35, 639, 641
VioWrtNCell, 35, 639, 641
VioWrtNChar, 35, 639, 641
VioWrtTTY, 35, 639, 640, 642, 643, 653

W

WC_BUTTON, 96, 160
WC_ENTRYFIELD, 96, 171, 172
WC_FRAME, 60, 62, 96, 127, 129, 247
WC_LISTBOX, 96, 182
WC_MENU, 96, 215
WC_SCROLLBAR, 62, 96, 203
WC_STATIC, 96, 196
WC_TITLEBAR, 62, 96
Win functions, 26
WinAddAtom, 711
WinAddProgram, 718
WinAddSwitchEntry, 726, 727
WinAlarm, 722
WinAllocMem, 324, 325, 326, 327, 328
WinAvailMem, 325, 328
WinBeginEnumWindows, 69
WinBeginPaint, 266, 269, 271, 272, 273, 274, 283, 298, 394, 396
WinBroadcastMsg, 83, 88, 711, 729
WinCalcFrameRect, 137
WinCallMsgFilter, 368
WinCatch, 43
WinChangeSwitchEntry, 719, 727
WinCloseClipbrd, 336
WinCopyAccelTable, 242
WinCreateAccelTable, 242
WinCreateAtomTable, 710
WinCreateCursor, 297, 298
WinCreateDlg, 54, 66, 130, 250
WinCreateFrameControls, 54, 137, 138
WinCreateGroup, 718
WinCreateHeap, 322, 323, 325, 328, 329, 610
WinCreateMenu, 54
WinCreateMsgQueue, 33, 52, 80, 86
WinCreatePointer, 290
WinCreateStdWindow, 54, 66, 67, 129, 130, 132, 134, 139, 140, 154, 208, 222, 223
WinCreateWindow, 52, 54, 55, 63, 66, 68, 93, 96, 97, 129, 134, 140, 145, 146, 160, 172, 173, 182, 196, 203, 205, 208, 209, 229
WinDdeInitiate, 349, 350, 353
WinDdePostMsg, 355
WinDdeRespond, 350, 353
WinDefAVioWindowProc, 653
WinDefDlgProc, 103, 253, 257, 381
WinDefWindowProc, 60, 83, 102, 103, 104, 105, 113, 114, 115, 138, 139, 155, 187, 198, 223, 290, 349, 371, 372, 380, 381
WinDeleteAtom, 713
WinDestroyCursor, 298
WinDestroyHeap, 329
WinDestroyWindow, 56, 57, 67, 254, 255
WinDismissDlg, 135, 253, 254, 257, 261
WinDispatchMsg, 82, 86, 87, 173, 368, 699, 701
WinDlgBox, 54, 66, 253, 254, 255, 257
Window, 49–75
  class, 91–98
  control window, 143–149
  dialog window, 245–262
  drawing, 279–286
  frame window, 125–142
  procedure, 99–107
  timer, 697–702
Window manager
  dialog window, 18
  menu, 18
  message loop, 18
  window, 17
WinDrawBitmap, 284, 291, 518, 519
WinDrawPointer, 291
WinDrawText, 285
WinEmptyClipbrd, 335, 339, 340
WinEnableWindow, 58, 206, 258
WinEnableWindowUpdate, 184
WinEndEnumWindows, 69
WinEndPaint, 269, 271, 272, 298, 396
WinEnumDlgItem, 260
WinFillRect, 283
WinFlashWindow, 135, 153
WinFormatFrame, 137, 138, 139
WinFreeMem, 325, 328
WinGetCurrentTime, 700
WinGetKeyState, 366
WinGetMaxPosition, 64
WinGetMsg, 80, 81, 82, 85, 86, 87, 88, 111, 268, 315, 365, 366, 368
WinGetNextWindow, 69
WinGetPhysKeyState, 369
WinGetPS, 57, 270, 271, 272, 340, 394, 396, 452
WinGetSysBitmap, 291
WinInflateRect, 182
WinInitialize, 32, 52
WinInSendMsg, 84
WinInvalidateRect, 268
WinInvalidateRegion, 268
WinInvertRect, 283
WinIsWindowEnabled, 58
WinIsWindowVisible, 72
WinLoadAccelTable, 242
WinLoadDlg, 54, 130, 254, 255, 261
WinLoadMenu, 54, 217, 223, 255
WinLoadPointer, 290
WinLockHeap, 324
WinLockWindow, 57, 58, 59, 68
WinMapDlgPoints, 250, 261
WinMapWindowPoints, 283
WinMessageBox, 54, 248, 251, 252, 261
WinOpenClipbrd, 335, 336
WinOpenWindowDC, 273, 274, 275, 397, 421, 516
WinPeekMsg, 80, 81, 86, 87, 111, 268, 315, 365, 366
WinPostMsg, 18, 83, 84, 85, 88
WinProcessDlg, 254, 255, 256, 257, 261
WinQueryAccelTable, 242
WinQueryActiveWindow, 58, 112, 114, 120
WinQueryClassInfo, 95, 97
WinQueryClassName, 97
WinQueryClipbrdData, 336
WinQueryClipbrdFmtInfo, 338
WinQueryClipbrdOwner, 338
WinQueryClipbrdViewer, 337
WinQueryDefinition, 718
WinQueryDlgItemShort, 173, 261
WinQueryDlgItemText, 261
WinQueryFocus, 120
WinQueryProfileData, 725
WinQueryProfileInt, 725
WinQueryProfileString, 305, 306, 725
WinQueryProgramTitles, 718, 719
WinQueryQueueInfo, 81

WinQueryQueueStatus, 81, 87, 88, 369
WinQuerySysColor, 720, 727
WinQuerySysModalWindow, 58
WinQuerySysPointer, 290
WinQuerySystemAtomTable, 710
WinQuerySysValue, 724, 728
WinQueryWindow, 58, 68, 141
WinQueryWindowLockCount, 58
WinQueryWindowPos, 70
WinQueryWindowProcess, 351
WinQueryWindowRect, 64
WinQueryWindowText, 163, 173
WinQueryWindowULong, 67
WinQueryWindowUShort, 59, 67, 135, 136
WinReallocMem, 325, 328
WinRegisterClass, 93, 97, 98, 104
WinRegisterWindowDestroy, 57
WinReleaseHook, 372, 384
WinReleasePS, 57, 270, 271, 272, 273, 275, 396
WinRemoveSwitchEntry, 727
WinScrollWindow, 284
WinSendDlgItemMsg, 261
WinSendMsg, 83, 88, 366
WinSetAccelTable, 226, 241, 242
WinSetActiveWindow, 114
WinSetCapture, 115
WinSetClipbrdData, 335, 339, 342
WinSetClipbrdOwner, 339
WinSetClipbrdViewer, 337
WinSetDlgItemShort, 173, 256, 261
WinSetDlgItemText, 256, 261
WinSetFocus, 114, 119, 120, 262
WinSetHook, 372, 373, 384
WinSetMultWindowPos, 71, 138
WinSetOwner, 69
WinSetParent, 68
WinSetPointer, 290
WinSetSysColors, 720, 721, 727, 728
WinSetSysModalWindow, 58, 131, 133
WinSetSysValue, 724, 729
WinSetWindowPos, 64, 70, 71, 72, 132, 162, 195
WinSetWindowText, 146, 154, 163, 173, 195, 258
WinSetWindowULong, 55
WinSetWindowUShort, 59
WinShowWindow, 55, 72, 146, 184, 256
WinStartTimer, 699, 700, 701
WinStopTimer, 700
WinSubclassWindow, 59, 60, 104
WinSubstituteStrings, 255, 261
WinTerminate, 31
WinThrow, 43
WinTrackRect, 44
WinUpdateWindow, 269
WinWaitMsg, 88
WinWindowFromID, 69, 129, 141, 154, 161, 162, 163, 172, 183, 196, 208, 216, 217, 223, 227, 257, 258
WinWindowFromPoint, 69
WinWriteProfileData, 720, 725, 726
WinWriteProfileString, 720, 725, 726
WM_ACTIVATE, 57, 68, 113, 114, 119, 120, 137
WM_ADJUSTWINDOWPOS, 54, 64, 155, 175, 188, 197
WM_BUTTON1DBLCLK, 137, 155, 164, 174
WM_BUTTON1DOWN, 114, 137, 155, 164, 165, 174, 369
WM_BUTTON1UP, 137, 164, 165, 174, 369
WM_BUTTON2DBLCLK, 137
WM_BUTTON2DOWN, 137, 174, 188, 369
WM_BUTTON2UP, 137, 369
WM_BUTTON3DBLCLK, 137
WM_BUTTON3DOWN, 137, 174, 188, 369
WM_BUTTON3UP, 137, 369
WM_BUTTONCLICKFIRST, 86
WM_BUTTONCLICKLAST, 86
WM_CALCVALIDRECTS, 65, 137, 138
WM_CHAR, 16, 18, 86, 87, 115, 116, 117, 118, 119, 134, 164, 173, 174, 187, 207, 255, 366, 368, 369, 370
WM_CLOSE, 66, 138
WM_COMMAND, 86, 119, 134, 159, 160, 161, 162, 163, 164, 216, 220, 223, 224, 225, 226, 239, 240, 242, 253, 257, 315
WM_CONTROL, 160, 161, 162, 163, 164, 173, 175, 176, 183, 184, 189
WM_CONTROLHEAP, 138
WM_CREATE, 54, 102, 103, 138, 154, 164, 174, 187, 197, 256
WM_DDE_ACK, 350, 354, 355, 356, 357, 358, 359
WM_DDE_ADVISE, 349, 356
WM_DDE_DATA, 350, 356, 357
WM_DDE_EXECUTE, 358
WM_DDE_FIRST, 86
WM_DDE_INITIATE, 349, 350, 351, 353, 356
WM_DDE_INITIATEACK, 349, 350, 351
WM_DDE_LAST, 86
WM_DDE_POKE, 355
WM_DDE_REQUEST, 355, 358
WM_DDE_TERMINATE, 350, 354, 359
WM_DDE_UNADVISE, 350, 357, 358
WM_DESTROY, 18, 57, 61, 136, 138, 155, 164, 174, 187, 197
WM_DESTROYCLIPBOARD, 339
WM_DRAWCLIPBOARD, 337, 338
WM_DRAWITEM, 182, 185, 186, 230, 231
WM_ENABLE, 138, 165, 175, 188, 197
WM_ERASEBACKGROUND, 138, 139
WM_FLASHWINDOW, 135
WM_FORMATFRAME, 137, 138, 139
WM_HELP, 119, 134, 216, 220, 222, 223, 224, 226, 239, 240, 242, 370, 371, 372, 379, 380, 381, 382, 383, 384
WM_HITTEST, 95, 113, 137, 155, 198
WM_HSCROLL, 204
WM_HSCROLLCLIPBOARD, 340
WM_INITDLG, 103, 183, 184, 253, 255, 256, 261, 262
WM_JOURNALNOTIFY, 369
WM_MATCHMNEMONIC, 165, 198
WM_MEASUREITEM, 185, 230
WM_MENUSELECT, 224, 382
WM_MINMAXFRAME, 138
WM_MOUSEFIRST, 86
WM_MOUSELAST, 86
WM_MOUSEMOVE, 114, 139, 164, 174, 175, 188, 197, 290, 291, 369
WM_MOVE, 64, 95
WM_NEXTMENU, 225
WM_OTHERWINDOW-DESTROYED, 57

WM_PAINT, 56, 65, 81, 84, 86, 104, 138, 139, 155, 164, 174, 175, 187, 197, 266, 267, 268, 269, 270, 272, 273, 274, 283, 284, 298, 309, 315, 338, 700
WM_PAINTCLIPBOARD, 340
WM_QUERYDLGCODE, 155, 165, 175, 197, 262
WM_QUERYICON, 139
WM_QUERYTRACKINFO, 139, 154
WM_QUERYWINDOWPARAMS, 155, 165, 175, 197
WM_QUIT, 18, 19, 32, 33, 82, 83, 88
WM_RENDERALLFMTS, 57, 339, 342
WM_RENDERFMT, 339, 341
WM_SEM1, 85
WM_SEM2, 85, 86
WM_SEM3, 85, 86
WM_SEM4, 85, 86
WM_SETFOCUS, 119, 120, 164, 175, 187, 195, 197, 297
WM_SETSELECTION, 119, 120, 175
WM_SETWINDOWPARAMS, 155, 165, 175, 197
WM_SHOW, 64, 139
WM_SIZE, 64, 139, 162, 204, 653
WM_SIZECLIPBOARD, 339
WM_SUBSTITUTESTRING, 255, 261, 262
WM_SYSCOLORCHANGE, 721
WM_SYSCOMMAND, 65, 119, 134, 137, 139, 216, 220, 223, 224, 239, 240, 242
WM_SYSVALUECHANGED, 83, 88, 724, 729
WM_TIMER, 86, 175, 188, 699, 700
WM_TRACKFRAME, 154, 155
WM_TRACKMOVE, 174
WM_UPDATEFRAME, 137, 139, 223, 255
WM_USER, 85, 102, 711
WM_VIOCHAR, 369, 370
WM_VSCROLL, 188, 204
WM_VSCROLLCLIPBOARD, 340
WS_CLIPCHILDREN, 56, 61, 95
WS_CLIPSIBLINGS, 56, 61, 95
WS_DISABLED, 56, 58
WS_GROUP, 56, 160, 161, 248, 250
WS_MAXIMIZED, 56, 65
WS_MINIMIZED, 56, 65, 270
WS_PARENTCLIP, 56, 95
WS_SAVEBITS, 56, 95
WS_SYNCPAINT, 55, 56, 94, 268, 284
WS_TABSTOP, 56, 248, 250
WS_VISIBLE, 55, 63, 72, 94, 255

Also Available From Microsoft Press

Authoritative Information for OS/2 Programmers

INSIDE OS/2

Gordon Letwin

"The best way to understand the overall philosophy of OS/2 will be to read this book."

— Bill Gates

Here — from Microsoft's Chief Architect of Systems Software — is an exciting technical examination of the philosophy, key development issues, programming implications, and role of OS/2 in the office of the future. And Letwin provides the first in-depth look at each of OS/2's design elements. This is a valuable and revealing programmer-to-programmer discussion of the graphical user interface, multitasking, memory management, protection, encapsulation, interprocess communication, and direct device access. You can't get a more inside view.

304 pages, $7^{3}/_{8}$ x $9^{1}/_{4}$, softcover, $19.95.
[Order Code 86-96288]

ADVANCED OS/2 PROGRAMMING

Ray Duncan

Authoritative information, expert advice, and great assembly-language code make this comprehensive overview of the features and structure of OS/2 indispensable to any serious OS/2 programmer. Duncan addresses a range of significant OS/2 issues: programming the user interface; mass storage; memory management; multitasking; interprocess communications; customizing filters, device drivers, and monitors; and using OS/2 dynamic link libraries. A valuable reference section includes detailed information on each of the more than 250 system service calls in version 1.1 of the OS/2 kernel.

800 pages, $7^{3}/_{8}$ x $9^{1}/_{4}$, softcover, $24.95
[Book Code 86-96106]

PROGRAMMING THE OS/2 PRESENTATION MANAGER

Charles Petzold

New! Here is the first full discussion of the features and operation of the OS/2 1.1 Presentation Manager. If you're developing OS/2 applications, this book will guide you through Presentation Manager's system of windows, messages, and function calls. Petzold includes scores of valuable C programs and utilities.

Endorsed by the Microsoft Systems Software group, this book is unparalleled for its clarity, detail, and comprehensiveness. Petzold covers: managing windows ■ handling input and output ■ controlling child windows ■ using bitmaps, icons, pointers, and strings ■ accessing the menu and keyboard accelerators ■ working with dialog boxes ■ understanding dynamic linking ■ and more.

864 pages, 7⅜ x 9¼, softcover, $29.95
[Order Code 86-96791]

ESSENTIAL OS/2 FUNCTIONS: Programmer's Quick Reference

Ray Duncan

Concise information on the essential OS/2 function calls within the application program interface (API). Entries are included for all kernel API functions for OS/2 version 1.0: Dos, Kbd, Mou, and Vio. Brief descriptions of each function are included, as well as a list of the required parameters, returned results, programming notes and warnings, family API call identification, and error codes. Conveniently arranged to provide quick access to the information you need.

172 pages, 4¾ x 8, softcover, $9.95
[Order Code 86-96866]

For the Windows Programmer

PROGRAMMING WINDOWS

Charles Petzold

Your fastest route to successful application programming with Windows.
Full of indispensable reference data, tested programming advice, and page after page of creative sample programs and utilities. Topics include getting the most out of the keyboard, mouse, and timer; working with icons, cursors, bitmaps, and strings; exploiting Windows' memory management; creating menus; taking advantage of child window controls; incorporating keyboard accelerators; using dynamically linkable libraries; and mastering the Graphics Device Interface (GDI). A thorough, up-to-date, and authoritative look at Windows' rich graphical environment.

864 pages, 7⅜ x 9¼
$24.95 (sc) [Order Code 86-96049]
$34.95 (hc) [Order Code 86-96130]

Solid Technical Information for MS-DOS® Programmers

ADVANCED MS-DOS® PROGRAMMING, 2nd ed.

Ray Duncan

The preeminent source of MS-DOS information for assembly-language and C programmers — now completely updated with new data and programming advice covering: ROM BIOS for the IBM® PC, PC/AT®, PS/2®, and related peripherals; MS-DOS through version 4.0; version 4.0 of the LIM EMS; and OS/2 compatibility considerations. Duncan addresses key topics, including character devices, mass storage, memory allocation and management, and process management. In addition, there is a healthy assortment of updated assembly-language and C listings that range from code fragments to complete utilities. And the reference section, detailing each MS-DOS function and interrupt, is virtually a book within a book.

512 pages, 7⅜ x 9¼, softcover, $24.95
[Order Code 86-96668]

THE MS-DOS® ENCYCLOPEDIA

General Editor, Ray Duncan

The ultimate reference for insight, data, and advice to make your MS-DOS programs reliable, robust, and efficient. 1600 pages packed with version-specific data. Annotations of more than 100 system function calls, 90 user commands, and a host of key programming utilities. Hundreds of hands-on examples, thousands of lines of code, and handy indexes. Plus articles on debugging, writing filters, installable device drivers, TSRs, Windows, memory management, the future of MS-DOS, and much more. Researched and written by a team of MS-DOS experts — many involved in the creation and development of MS-DOS. Covers MS-DOS through version 3.2, with a special section on version 3.3.

1600 pages, 7¾ x 10
hardcover $134.95 [Order Code 86-96122]
softcover $ 69.95 [Order Code 86-96833]

Programmer's Quick Reference Series

MS-DOS® FUNCTIONS

Ray Duncan

The kind of information every seasoned programmer needs right at hand. Includes detailed information on MS-DOS system service calls, along with valuable programming notes. Covers MS-DOS through version 4.

128 pages, 4¾ x 8, softcover, $5.95
[Order Code 86-96411]

IBM® ROM BIOS

Ray Duncan

Essential for every assembly-language or C programmer at any experience level. Designed for quick and easy access to information, this guide includes all the core information on each of the ROM BIOS services.

128 pages, 4¾ x 8, softcover, $5.95
[Order Code 86-96478]

MS-DOS® EXTENSIONS

Ray Duncan

Brings together the hard-to-find programming information on the Lotus®/Intel®/ Microsoft® Expanded Memory Specification (EMS) version 4.0, the Lotus/ Intel/Microsoft/AST Extended Memory Specification (XMS) version 2.0, the Microsoft CD-ROM Extensions version 2.1, and the Microsoft Mouse driver, version 6. An overview of each function is accompanied by a list of its required parameters, returned results, and applicable programming notes.

128 pages, 4¾ x 8, softcover, $6.95
[Order Code 86-97229]

Solid Language References

MICROSOFT® C: SECRETS, SHORTCUTS & SOLUTIONS

Kris Jamsa

Here is a fact-filled, example-packed resource for any current or aspiring Microsoft C programmer working in the DOS environment. Each chapter highlights specific C programming facts, tips, and traps so that key information or

items of special interest are immediately accessible. Hundreds of short sample programs support Jamsa's instruction and encourage experimentation.
If you're new to C, Microsoft C, or even Microsoft QuickC, Jamsa's fast-paced, highly readable style will help you quickly master the fundamentals. If you're a seasoned programmer, you'll find page after page of advanced information that will hone your programming skills and make your Microsoft C programs fast, clean, and efficient. Jamsa shows you how to:
access the DOS command line ■ expand wildcard characters into matching filenames ■ use I/O redirection ■ master dynamic memory allocation ■ take advantage of C's predefined global variables ■ optimize your programs for increased speed ■ enhance your program's video appearance ■ make full use of the MAKE and LIB tools

500 pages, $7\frac{3}{8}$ x $9\frac{1}{4}$, softcover, $24.95
[Order Code 86-97112]

PROFICIENT C

Augie Hansen

"A beautifully-conceived text, clearly written and logically organized... a superb guide." ***Computer Book Review***

An information-packed handbook for intermediate to advanced DOS programmers that includes dozens of file-oriented and screen-oriented C programs and specially developed utilities. A successful blend of programming advice and practical example programs.

512 pages, $7\frac{3}{8}$ x $9\frac{1}{4}$, softcover, $22.95
[Order Code 86-95710]

VARIATIONS IN C

Steve Schustack

Foreword by Gerald Weinberg

A superb guide for experienced programmers who want to develop efficient, portable, high-quality application software using C in the DOS environment.
In addition to an overview of the basic syntax of C, Schustack provides valuable techniques for structured programming. A complete, 1500-line source code sample program illustrates key topics. Special comments and cautions are highlighted throughout.

368 pages, $7\frac{3}{8}$ x $9\frac{1}{4}$, softcover, $19.95
[Order Code 86-95249]

STANDARD C: Programmer's Quick Reference

P.J. Plauger and Jim Brodie

All the basic information needed to read and write Standard C programs that conform to the recently approved ANSI and ISO standard for the C programming language. Scores of diagrams illustrate the syntax rules. Whether you're new to C or familiar with an earlier dialect, this will prove a handy companion.

224 pages, 4¾ x 8, softcover, $7.95
[Order Code 86-96676]

MICROSOFT® QUICKC PROGRAMMING

The Waite Group

Your springboard to the core of the Microsoft QuickC. This book is loaded with practical information and advice on *every* element of QuickC, along with hundreds of specially constructed listings. Included are the tools to help you master QuickC's built-in libraries; manage file input and output; work with strings, arrays, pointers, structures, and unions; use the graphics modes; develop and link large C programs; and debug your source code.

624 pages, 7⅜ x 9¼, softcover, $19.95
[Order Code 86-96114]

MICROSOFT® QUICKBASIC®, 2nd ed.

Douglas Hergert

"No matter what your level of programming experience, you'll find this book irreplaceable when you start to program in QuickBASIC." ***Online Today***

Here's a great introduction to all the development tools, features, and user-interface enhancements in Microsoft QuickBASIC. And there's more — six specially designed, full-length programs including a database manager, an information-gathering and data-analysis program, and a chart program that reenforce solid structured programming techniques.

464 pages, 7⅜ x 9¼, softcover, $19.95
[Order Code 86-96387]

THE MICROSOFT® QUICKBASIC PROGRAMMER'S TOOLBOX

John Clark Craig

This essential library of subprograms, functions, and utilities—developed to supercharge your QuickBASIC programs—addresses common and unusual programming tasks: ANSI.SYS screen control ■ mouse support ■ pop-up windows ■ graphics ■ string manipulations ■ bit manipulation ■ editing routines ■ game programming ■ interlanguage calling ■ and more. Each program takes maximum advantage of QuickBASIC's capabilities. You're guaranteed to turn to this superb collection again and again.

512 pages, $7^{3}/_{8}$ x $9^{1}/_{4}$, softcover, $22.95
[Order Code 86-96403]

Unbeatable Programmer's References

PROGRAMMER'S GUIDE TO PC & PS/2® VIDEO SYSTEMS

Richard Wilton

No matter what your hardware configuration, here is all the information you need to create fast, professional, even stunning video graphics on IBM PCs, compatibles, and PS/2s. No other book offers such detailed, specialized programming data, techniques, and advice to help you tackle the exacting challenges of programming directly to the video hardware. And no other book offers the scores of invaluable source code examples included here. Whatever graphic output you want—text, circles, region fill, alphanumeric character sets, bit blocks, animation—you'll do it cleaner, faster, and more effectively with Wilton's book.

544 pages, $7^{3}/_{8}$ x $9^{1}/_{4}$, softcover, $24.95
[Order Code 86-96163]

THE 80386 BOOK

Ross P. Nelson

A clear, comprehensive, and authoritative introduction for every serious programmer. Included are scores of superb assembly-language examples along with a detailed analysis of the 80386 chip. Topics covered include: the CPU, the memory architecture, the instructions sets of the 80386 microprocessor and the

80387 math coprocessor, the protection scheme, the implementation of a virtual memory system through paging, and compatibility with earlier Intel microprocessors. Of special note is the comprehensive, clearly organized instruction set reference — guaranteed to be a valuable resource.

464 pages, $7\frac{3}{8}$ x $9\frac{1}{4}$, softcover, $24.95
[Order Code 86-96494]

THE PROGRAMMER'S PC SOURCEBOOK

Thom Hogan

At last! A reference to save you the time required to find key pieces of technical data. Here is important factual information — previously published in scores of other sources — organized into one convenient reference. Focusing on IBM PCs and compatibles, PS/2s and MS-DOS, the hundreds of charts and tables cover: ■ numeric conversions and character sets ■ DOS commands and utilities ■ DOS function calls and support tables ■ DOS BIOS calls and support tables ■ other interrupts, mouse, and EMS support ■ Microsoft Windows ■ keyboards, video adapters, and peripherals ■ chips, jumpers, switches, and registers ■ hardware descriptions ■ and more.

560 pages, $8\frac{1}{2}$ x 11, softcover, $24.95
[Order Code 86-96296]

THE *NEW* PETER NORTON PROGRAMMER'S GUIDE TO THE IBM® PC & PS/2®

Peter Norton and Richard Wilton

A must-have classic on mastering the inner workings of IBM micros — now completely updated to include the PS/2 line. Sharpen your programming skills and learn to create simple, clean, portable programs with this successful combination of astute programming advice, proven techniques, and solid technical data. Covers 8088, 80286 and 80386 microprocessors; ROM BIOS basics and ROM BIOS services; video, disk and keyboard basics; DOS basics, interrupts, and functions (through version 4); interrupts, device drivers, and video programming. Accept no substitutes; this is the book to have.

528 pages, $7\frac{3}{8}$ x $9\frac{1}{4}$, softcover, $22.95
[Order Code 86-96635]

The Microsoft Press CD-ROM Library

THE MICROSOFT® CD-ROM YEARBOOK: 1989/1990

Microsoft Press

Foreword by Bill Gates

A dynamic, fact-filled portrait and analysis of the wide-ranging, fast-paced CD-ROM industry. Indispensable for anyone involved in the industry as well as an information-packed compendium for those curious about CD-ROM. Readers can use the book as a valuable sourcebook of facts, statistics, and forecasts, or dip into it for fascinating articles, reviews, and analyses of the industry. Articles include:

- an absorbing history — in text and pictures — of the CD-ROM industry
- reviews of products — hardware and software — considered outstanding or standard-setting
- profiles of the leading companies and people in the industry
- an overview of the process of developing a CD-ROM product
- a review of the legal issues of protection, rights and permissions, contracts and royalties surrounding CD-ROM publishing
- the strategies and pitfalls involved in getting a CD-ROM product to market

The breadth of accurate, up-to-date information in THE MICROSOFT CD-ROM YEARBOOK is impressive including:

- comprehensive reference listings of the people, equipment, available titles, sources, and resources in the CD ROM industry
- a glossary of industry terms
- a calendar of industry events and conferences
- specialized bibliographies

This is *the* reference of fact and opinion on the industry.

960 pages, $7\frac{3}{8}$ x $9\frac{1}{4}$, softcover, $79.95
[Order Code 86-97203]

CD ROM: THE NEW PAPYRUS

Edited by Steve Lambert and Suzanne Ropiequet

"This 619-page compendium, with contributions from more than 30 optical-memory specialists, promises to become the bible of CD ROM." **David Bunnell, *Macworld***

This special compendium of 45 articles by leading authorities examines every facet of compact disc read only memory technology: hardware, software, applications, publishing systems, marketing, and the user interface. Includes introductory as well as technical information.

608 pages, 7⅜ x 9¼, softcover, $21.95
[Order Code 86-95454]

CD ROM 2: OPTICAL PUBLISHING

Edited by Suzanne Ropiequet with John Einberger and Bill Zoellick

"Recommended reading for any information professional." ***Online Today***

This is a comprehensive overview of the entire optical publishing process. Topics include: evaluating and defining storage and retrieval methods; collecting, preparing, and indexing data; updating strategies; data protection and copyrighting; and more. Plus information on the High Sierra Logical Format. In addition, the editors trace the development of two CD ROM projects from initial concept to final product. For publishers, technical managers, and entrepreneurs.

384 pages, 7⅜ x 9¼, softcover, $22.95
[Order Code 86-95686]

INTERACTIVE MULTIMEDIA

Foreword by John Sculley

Edited by Sueanne Ambron and Kristina Hooper

Apple Computer Corp. brought together leading researchers and developers to produce this informative collection of 21 articles. The result is a sourcebook of ideas and inspiration for software and hardware developers, educators, publishers, and information providers. The contributors, including Doug Englebart, Sam Gibbon, and Peter Cook, represent the industries — computers, television, and publishing — whose products will provide the content and media for education in the future. Filled with examples and pilot projects that define the new meaning of multimedia. Published with Apple Computer, Inc.

352 pages, 7⅜ x 9¼, softcover, $24.95
[Order Code 86-96379]

Also of Note

COMPUTER LIB/DREAM MACHINES

Ted Nelson

''An exuberant, multifont compendium of computing proverbs, anecdotes, jokes, predictions, and politics. Still as fresh and relevant as it was a dozen years ago, Computer Lib is a browser's gold mine.'' ***PC World***

Published in 1974, Ted Nelson's COMPUTER LIB was an original, off-the-wall compendium of Nelson's visionary wisdom on the state of computing. Immediately embraced by hackers, COMPUTER LIB/DREAM MACHINES provided inspiration to today's industry greats. Nelson anticipated the personal computer revolution, made outlandish predictions (many of which have proven true), and expounded on his vision of non-sequential data storage — something he dubbed hypertext. Long unavailable, COMPUTER LIB has been updated with new commentaries and insights from Nelson.

336 pages, 9¼ x 9¾, softcover, $18.95
[Order Code 86-96031]

Microsoft Press books are available wherever books and software are sold.
Or you can place a credit card order by calling **1-800-638-3030** *(8 AM to 4:30 PM EST).*
In Maryland, call collect: 824-7300.

MICROSOFT® OS/2 Programmer's Reference

Including Presentation Manager

The Microsoft® Operating System/2 Programmer's Reference Library should be the cornerstone of every OS/2 developer's programming library. These volumes are required references for professional developers creating applications for the retail market; for corporate programmers creating in-house software programs; for hardware manufacturers creating software to support their products; and for all other experienced programmers working in the OS/2 environment.

Each volume in the series is written by a team of OS/2 specialists — many involved in the development and ongoing enhancement of OS/2 at Microsoft. These books provide in-depth, accurate, and up-to-date information from the Microsoft OS/2 Presentation Manager Toolkit — the software development kit essential for creating OS/2 applications.

Volume 1

Volume 1 details the conceptual framework of the MS® OS/2 Application Programming Interface (API). Included are thorough descriptions of MS® OS/2 programming models, overviews of basic programming considerations, and explanations of the interaction between the API and the rest of the MS® OS/2 system. Sections include *Introducing MS® OS/2, Window Manager, Graphics Programming Interface,* and *System Services.*

Volume 2

Volume 2 is a comprehensive, alphabetic listing of MS® OS/2 Presentation Manager functions as well as the structures and file formats used with these functions. Each function entry includes information on syntax; descriptions of the function's actions and purpose; parameters and field definitions; return values, error values, and restrictions; source-code examples; and programming notes. Appendix included.

Volume 3

Similar in format to Volume 2, Volume 3 is a comprehensive alphabetic listing of MS® OS/2 base functions, including their structures and file formats. Appendixes included.

ISBN 1-55615-220-5

52995

9 781556 152207

U.S.A.	**$29.95**
U.K.	£24.95
Austral.	$44.95

(recommended)

Computers/Operating Systems/OS/2