Programming the OS/2™ WARP *Version 3* GPI

Stephen A. Knight
Jeffrey M. Ryan

Graphics Editor & Utilities Included

Programming the OS/2® WARP Version 3 GPI

Stephen A. Knight
Jeffrey M. Ryan

John Wiley & Sons, Inc.

NEW YORK / CHICHESTER / BRISBANE / TORONTO / SINGAPORE

Publisher: Katherine Schowalter
Senior Editor: Diane D. Cerra
Managing Editor: Robert S. Aronds
Editorial Production & Design: Claire J. Riley

This text is printed on acid–free paper.

IBM, OS/2, and PS/2 are registered trademarks of International Business Machines Corporation. Operating System/2, C SET/2, WorkFrame/2, and Presentation Manager are trademarks of International Business Machines Corporation. Intel is a registered trademark of Intel Corporation. Helvetica and Courier are trademarks of Linotype Company. Times New Roman is a trademark of Monotype.

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

Copyright © 1995 by John Wiley & Sons, Inc.

All rights reserved. Published simultaneously in Canada.

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

Reproduction or translation of any part of this work beyond that permitted by section 107 or 108 of the 1976 United States Copyright Act without the permission of the copyright owner is unlawful. Requests for permission or further information should be addressed to the Permissions Department, John Wiley & Sons, Inc.

Library of Congress Cataloging–in–Publication Data

Knight, Stephen A., 1955–
Programming the OS/2 Warp Version 3 GPI / Stephen A. Knight, Jeffrey M. Ryan.
p. cm.
Includes bibliographical references.
ISBN 0–471–10718–2 (acid–free paper)
1. Operating systems (Computers) 2. Graphical user interfaces (Computer systems) I. Ryan, Jeffrey M., 1963– . II. Title.
QA76.76.O63K625 1995
005.265—dc20 94–37771
CIP

Foreword

It has been over two years since OS/2 2.0 was introduced. Since that time, OS/2 2.0 has become a major breakthrough because it maintained compatibility with hundreds of popular DOS and Windows applications while providing true, pre-emptive multitasking, and 32-bit programming support. It not only does this better than most people thought possible, it also delivers an enhanced user interface that makes it more intuitive and easier to use than earlier versions.

IBM has continued to enhance OS/2 because it remains committed to providing more and improved device support, easier installation, better connectivity options, new functionality like multimedia, and the best product support possible. IBM has continued to listen to its customers, making changes to best meet their needs. This commitment to listen and enhance the product can be seen in OS/2 Warp, which provides more features while significantly improving operating system performance—while requiring fewer system resources.

Software developers writing applications for OS/2 find a development environment that enables them to write leading-edge applications in a productive, efficient manner. This is because OS/2 has true, pre-emptive multitasking; advanced memory management; built-in application protection mechanisms; inter-process communication mechanisms; and a huge set of well-defined and powerful programming interfaces. The biggest problem most new developers have when writing OS/2 applications is twofold: sort-

ing through the vast number of functions available to them, and knowing how and when to apply them.

This book, written by two experienced IBM programmers, clearly illustrates how a set of programming functions, known as the Graphical Programming Interface (GPI), can be used to generate and manipulate graphical objects. The authors use a simple, but brilliantly effective approach to "teaching by example." They provide a sample program, explain how it is structured, and show why a particular GPI function is used. Surprisingly, this sample program turns out to be a completely functional graphics editor! Not only does the editor help you learn more about the internal structure of graphical objects, it also serves as a useful tool in and of itself.

As the authors explain the OS/2 graphics environment and step you through the graphics editor, you will discover a programming environment that is rich in function—one that helps you generate powerful applications more quickly and easily than you thought possible. In the process, you are also likely to discover that the OS/2 environment can be as much fun as it is productive.

Preface

One of the most exciting aspects of today's computer technology is that ordinary people can do tasks in their own homes that used to be affordable only by large companies. Of course, this relatively new power has been made possible by the introduction of the personal computer (PC). With a PC purchased in your local mall, you can perform tasks like accounting, word processing, maintaining information in your own personal data bases, producing graphics and drawings, and communicating with other computer owners around the world. Even though this is possible at a relatively low cost, many people have yet to discover the opportunities that the PC can provide.

The PC is a huge success with many businesses and home users, but many people are just beginning to discover the power of the PC. Personal computer technology has given us (at relatively low cost) high performance machines with large storage capacities, excellent print quality, excellent connectibility, and most important, a large set of application software. It is widely believed that application software is the key component that really enables people to become more productive; and a measure of how good a piece of application software is, is how easy it is to learn and use. After all, ease of use leads to productivity and high productivity is exactly what an excellent tool provides.

One way which application software can become more readily understood is with a graphical user interface (GUI). This does not mean that usable application software must be graphical or that a graphical user interface automatically adds usability to an application. In fact, a graphical appli-

cation developed carelessly may be less usable than a nongraphical application. If developed correctly, however, a graphical user interface can make an application much more intuitive and hence, easier to use. With the competitive nature of application software, the usability and pizzazz of the application may make or break a software product.

IBM's Operating System/2 (OS/2) provides a graphical user interface called Presentation Manager (PM). The Presentation Manager environment also provides an extremely powerful and full set of programming functions for software developers. Many of these functions deal with advanced graphics and are part of a programming interface called the Graphical Programming Interface (GPI). It is the intention of this book to help programmers who are already somewhat familiar with Presentation Manager and the C programming language to become even more familiar with the advanced functions provided by Presentation Manager's GPI. This book provides an entire application that uses many of the advanced graphical functions of Presentation Manager and discusses how these functions are used in relationship with one another to perform different tasks. If you already have C programming experience but no knowledge of programming Presentation Manager, it is recommended that you first understand basic Presentation Manager programming. Even though this book includes some similar information as most introductory books, it assumes you already have basic knowledge of Presentation Manager concepts and program structure.

The main application included with this book on a 3.5 inch diskette is a graphic editor. This application is not as functional as many commercially available graphic editors or drawing applications but, as you will see, it is still very powerful. In fact, all the artwork shown in this book was produced with the graphic editor provided with and discussed in this book. This graphic editor also has a feature that allows you to view the graphical orders used to produce different graphical objects and a tool that allows you to directly manipulate transformation matrices that are applied to different graphical objects. Furthermore, other utility programs and their C source code are included to help you examine OS/2 Metafiles and printer definition information for your PC. By providing you with these programming examples and development tools, it is hoped that your OS/2 GPI learning experience will be much more productive, fun, and exciting.

Acknowledgments

We would like to acknowledge the hundreds of dedicated software developers and support people who have made OS/2 the outstanding operating system it is today. We especially thank the following people for their technical assistance and thoughts that helped form this book: Joseph Celi, Alan Warren, Cary Bates, Tom Gall, Bret Curran, Brian Curran, Ricardo Hernandez–Muchado, Byron Watts, Steve Roth, Graham Winn, and Diane Cerra.

We especially thank Claire Riley, whose editing and typesetting expertise helped produce this book. Claire's skill and dedication made this book a reality in as short of time as we could have hoped for.

Finally, we thank our management team for their encouragement, support, and interest in this project. In particular, we thank Phil Mayer, Bill Van Vugt, and Linda Thompson.

My third book! I still have a hard time believing I participated in writing even one. Every time I finish a book, I can't imagine having the energy to do another. The only thing I can think of that gives me this type of drive is some lesson I must have learned early in life. Therefore, I hold my family responsible and dedicate this book to my parents, Tom and Barb Knight, and to my brother Mike Knight.

Steve Knight

In memory of Scott.

Loving thanks to my wife Terri, and children Krista and Bradley for their patience and support during this project. Without you, this book would not have been possible. Finally, thanks to my cat Sparky who was often my only companion during the wee hours of the morning.

Jeff Ryan

Contents

Foreword v

Preface vii

Acknowledgments ix

1 Overview 1

The Graphic Editor Application 2
Overview of Graphic Editor Program Structure 19
Working with the Diskette 26
Special Notes about this Book 30

2 OS/2 Architecture 32

The Viewing Pipeline and Picture Construction 39
A Little about Multitasking 45

3 Graphic Primitives 52

Attributes 52
Color and Mix 55
Line and Arc Primitives 59
Area Primitives 77
Area Functions 82
Text Primitives 88
Marker Primitives 99
Image Primitives 102

4 Fonts 122

Font Metrics 128

5 Building Blocks of the GPI 168

Orders 169
Elements 171
Segments 175
Graphic Editor Object Viewer 185
Graphic Segments in the Graphic Editor 204

6 Transformations 207

Transform Matrix Fundamentals 213
Graphic Editor Use of Model Transforms 227
Viewing Transforms 232
Default View Transform 234
Device Transform 238
Converting between Different Coordinate Spaces 240

7 Paths, Regions, Clipping, Boundary Accumulation, and Correlation 243

Paths 244
Regions 248
Clipping 257
Correlation 271

8 Printing Graphics 283

9 OS/2 MetaFiles 311

Appendix A: GPI Functions 337

Appendix B: Working with the Diskette 389

Installing the Example Software on Your PC 383
Running the Sample Programs 390

Index 399

CHAPTER 1
Overview

As a programmer, have you ever looked at a graphical application such as a word processor and wondered how much effort it took to produce it? I know I used to look at some highly graphical applications and began to feel fairly inadequate as a programmer. I couldn't imagine the amount of knowledge and time it must take to zap all those little pixels to a device in just the right way to get the desired effect. For example, I would think about how difficult it might be just to output a line of text to the screen using a non-proportional font. Some of the problems I thought about were things like what type of video monitors would I support and how do each of them operate? How would I obtain font information and render the character images on the device? How would cursor movement be managed? How would I output the same text and fonts to a printer and what printers would I support? What if the font was kerned? And these questions were only relating to text! How about drawing curves and filling them with different colored patterns? Or what about rotating the drawing? Or suppose I wanted to let the user select a graphic object in the drawing and let them move or change it? How could I do all these things?

Of course a project as complex as a competitive word processor is not done by a single programmer. The problems are too big and too numerous. Also, not known to me in my early days as a programmer were all the programming interfaces that exist to help application developers deal with these

kinds of problems, and as time has progressed, there are even more tools and application programming interfaces than ever. Today's OS/2® is an example of one of the leading programming environments that helps programmers deal with these types of complex problems.

This book shows how many of these problems can be solved by using OS/2's Presentation Manager™ programming environment. In particular, this book focuses on a part of Presentation Manager called the Graphical Programming Interface (GPI). This book demonstrates how to use the GPI by providing a complete program that uses many of the advanced features of the GPI and then discussing how the program was written. By doing this, you will have a working example to learn from as well as a working model that may assist you with your own projects. It should be noted, however, that the techniques used in this book are only one way to solve some of your complex programming problems. Do not restrict your thinking to the way our example program works. It is very likely you will find other and better ways to use the GPI for certain parts of your own projects.

The program example written for this book is a graphic editor. (In this book, we often refer to the graphic editor as the *draw application.*) Even though this graphic editor will not be competitive with the leading commercially available graphic editors, you will find it to be very functional and powerful. Remember, this graphic editor was produced by only two programmers in a relatively short period of time! You will see that this statement alone says something about the power of the OS/2 programming environment! So before discussing the major parts of the GPI, let's see what the graphic editor that comes with this book can do!

THE GRAPHIC EDITOR APPLICATION

The graphic editor that comes with this book has a lot of the same features found in commercially available graphic editors. One feature unique to this graphic editor is that you can select part or all of a picture you drew and then use a pull-down menu option to display in text what graphic orders were used to produce it. Obviously, the intention is to help developers understand how graphic objects are generated with the GPI! But, before talking about specific details of how to use the GPI, let's discuss the basic operation of the included graphic editor and list all its features.

After installing the graphic editor from the diskette included with this book (see Installing the Graphic Editor later in this chapter), you can start the application in one of four ways:

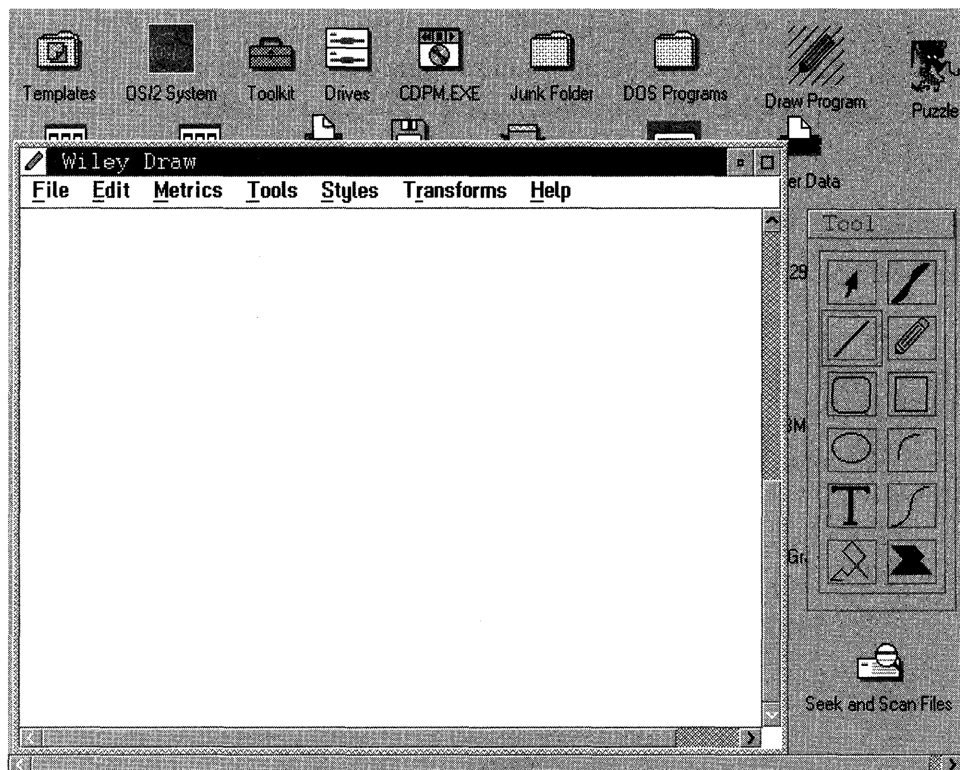
1. Double clicking on the graphic editor application's icon located on the desktop.

2. Drag and dropping a valid WLY draw file on the graphic editor application icon.

3. Entering the command DRAW from a command line prompt. (You must either be in the graphic editor program subdirectory or have your path set correctly.)

4. Associate WLY files with the graphic editor application (`DRAW.EXE`) and then double click on any valid WLY file.

When you start the graphic editor program, a product banner is displayed



SCREEN 1.1 Graphic editor initial screen.

followed by a screen similar to the one shown in Screen 1.1. From Screen 1.1 you can see the graphic editor application window and a modeless Tool dialog box. The Tool dialog box gives you a quick idea of the types of objects you can draw with the graphic editor. Row by row, the Tool dialog box lets you select the following editing tools:

1. **Select tool** – This tool allows you to select one or more objects that you have already drawn to be further manipulated. For instance, you can perform functions like move, rotate, size, copy, cut, delete, change drawing attributes, and so on, once you have selected the objects. To gain access to the Select tool, either click on the arrowhead icon found in the Tool dialog box or choose Select tool from the Tool pull-down menu. Once the Select tool is selected, an arrowhead icon becomes the mouse pointer while in the draw client area. To select an object, just click on the object of interest with mouse button 1. Once you select an object, you will notice that markers appear around it. These markers show which objects are selected and are used to help perform editing functions on that object. To select multiple objects, press and hold the Ctrl key while selecting objects by clicking mouse button 1 on the objects you want to be selected. One final way to select objects is by performing a marque select. Press and hold mouse button 1 and then move the mouse until the tracking rectangle surrounds the objects of interest. Release the mouse button. All objects within the tracking rectangle will be selected.

2. **Fillet Fill tool** – This tool allows you to set a series of points in the client area of the draw application window that a fillet curve will track to. Once all the points are specified, the curve is drawn and filled with the current fill color and pattern. To select the Fillet Fill tool you can either click on the Fillet Fill icon found in the Tool dialog box or choose Fillet Fill tool from the Tool pull-down menu. After you have selected the Fillet Fill tool, a cross-hair icon becomes the mouse pointer while in the draw client area. To draw a curve with this tool, simply press and hold mouse button 1 where you want the curve to start. Then move the mouse and release mouse button 1 where you want to specifiy a point for the curve. Repeat these two actions until you have specified all the points desired for the curve. To stop specifying points, simply double click mouse button 1 on the last point you want to define.

3. **Line tool** – This tool allows you to draw straight lines in the client area of the draw application window. To select the Line tool you can either click on the Line icon found in the Tool dialog box or choose Line tool from the Tool pull-down menu. After you have selected the Line tool, a cross-hair icon becomes the mouse pointer while in the draw client area. To draw a line with the Line tool, simply press and hold mouse button 1 where you want the line to start, then move the mouse and release mouse button 1 where you want the line to finish.

4. **Pencil tool** – This tool allows you to draw a series of lines in the client area in the same path as the movement of the mouse. To select the Pencil tool you can either click on the Pencil icon found in the Tool dialog box or choose Pencil tool from the Tool pull-down menu. After you have selected the Pencil tool, a Pencil icon becomes the mouse pointer while in the draw client area. To draw a path with the Pencil tool, simply press and hold mouse button 1 where you want the path to start, then move the mouse in the path you want to create and release mouse button 1 where you want the path to end.

5. **Rounded Box tool** – This tool allows you to draw a box with rounded corners in the client area of the draw application window. To select the Rounded Box icon you can either click on the Rounded Box tool found in the Tool dialog box or choose Rounded Box tool from the Tool pull-down menu. After you have selected the Rounded Box tool, a rounded box with a cross-hair icon becomes the mouse pointer while in the draw client area. To draw a rounded box with the Rounded Box tool, simply press and hold mouse button 1 where you want one corner of the box, then move the mouse and release mouse button 1 where you want the other corner of the box to be positioned.

6. **Box tool** – This tool allows you to draw a box in the client area of the draw application window. To select the Box tool you can either click on the Box icon found in the Tool dialog box or choose Box tool from the Tool pull-down menu. After you have selected the Box tool, a box with a cross-hair icon becomes the mouse pointer while in the draw client area. To draw a box with the Box tool, simply press and hold mouse button 1 where you want one corner of the box, then move the mouse and release mouse button 1 where you want the other corner of the box to be positioned.

7. **Ellipse tool** – This tool allows you to draw an ellipse in the client area of the draw application window. To select the Ellipse tool you can either click on the Ellipse icon found in the Tool dialog box or choose Ellipse tool from the Tool pull-down menu. After you have selected the Ellipse tool, an ellipse with a cross-hair icon becomes the mouse pointer while in the draw client area. To draw an ellipse with the Ellipse tool, simply press and hold mouse button 1 where you want the center of the ellipse to be positioned. Then move the mouse and release mouse button 1 when you get the desired ellipse.

8. **Arc tool** – This tool allows you to draw a partial arc in the client area of the draw application window. To select the Arc tool you can either click on the Arc icon found in the Tool dialog box or choose Arc tool from the Tool pull-down menu. After you have selected the Arc tool, a cross-hair icon becomes the mouse pointer while in the draw client area. To draw an arc with the Arc tool, simply press and hold mouse button 1 where you want the arc to start. Then move the mouse and release mouse button 1 when you get the desired arc. (Our arcs are always 90 degrees of an ellipse even though the GPI allows for more complicated arcs.)

9. **Text tool** – This tool allows you to draw text in the client area of the draw application window. To select the Text tool you can either click on the T icon found in the Tool dialog box or choose Text tool from the Tool pull-down menu. After you have selected the Text tool, a T with a small arrow icon becomes the mouse pointer while in the draw client area. To draw text with the Text tool, simply click mouse button 1 where you want the text to start. Another dialog box then appears to allow you to type in the desired text. After you close this text dialog box, the text will be drawn in the client area in the desired location. To edit existing text, double click mouse button 1 with the Select tool on the text object you want to change. Double clicking on the existing text box causes the edit text dialog box to reappear so you can edit the existing text string.

10. **Fillet tool** – This tool allows you to set a series of points in the client area of the draw application window that a fillet curve will track to. Once all the points are specified, the curve is drawn but **not** filled. To select the Fillet tool you can either click on the Fillet icon found in the Tool dialog box or choose Fillet tool from the Tool pull-down menu. After you have selected the Fillet tool, a cross-hair icon becomes the mouse pointer

while in the draw client area. To draw a curve with this tool, simply press and hold mouse button 1 where you want the curve to start. Then move the mouse and release mouse button 1 where you want to specifiy a point for the curve. Repeat these two actions until you have specified all the points desired for the curve. To stop specifying points, simply double click mouse button 1 on the last point you want to define.

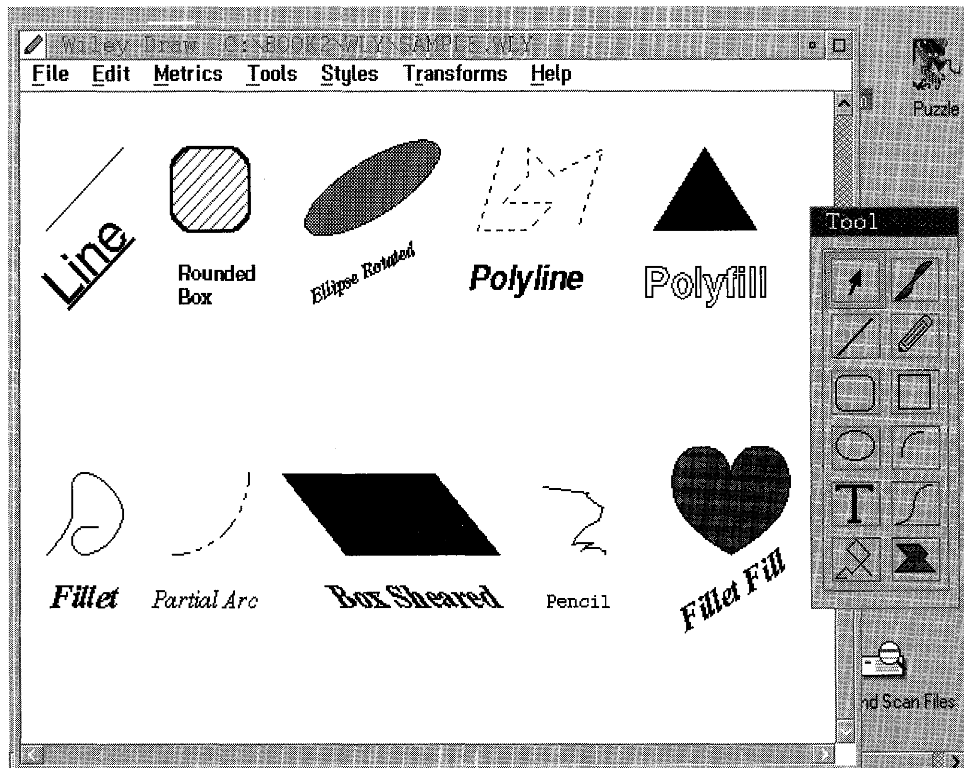
11. **Polyline tool** – This tool allows you to set a series of points in the client area of the draw application window that a series of lines will be drawn between. Once the all the points are specified, the lines are drawn but **no** fill is done. To select the Polyline tool you can either click on the Polyline icon found in the Tool dialog box or choose Polyline tool from the Tool pull-down menu. After you have selected the Polyline tool, a cross-hair icon becomes the mouse pointer while in the draw client area. To draw a series of lines with this tool, simply press and hold mouse button 1 where you want the first line to start. Then move the mouse and release mouse button 1 where you want to specifiy an end point. Repeat these two actions until you have specified all the points desired for the polyline. To stop specifying points, simply double click mouse button 1 on the last point you want to define.

12. **Polyline Fill tool** – This tool allows you to set a series of points in the client area of the draw application window that a series of lines will be drawn between. Once the all the points are specified, the lines are drawn and areas are filled with the current fill color and pattern. To select the Polyline Fill tool you can either click on the Polyline Fill icon found in the Tool dialog box or choose Polyline Fill tool from the Tool pull-down menu. After you have selected the Polyline Fill tool, a cross-hair icon becomes the mouse pointer while in the draw client area. To draw a series of lines with this tool, simply press and hold mouse button 1 where you want the first line to start. Then move the mouse and release mouse button 1 where you want to specify an end point. Repeat these two actions until you have specified all the points desired for the polyline. To stop specifying points, simply double click mouse button 1 on the last point you want to define.

For your convenience, the modeless Tool dialog box is designed to stay on the OS/2 desktop. As stated in the tool descriptions, however, there is an alternate way to select all of the tools. This alternate way is from the Tool

pull-down menu. Hence, it is not really necessary to have the modeless Tool dialog on the desktop, and sometimes the modeless Tool dialog box may seem to get in the way. If you want the Tool dialog to disappear from the desktop, click on Tool Palette from the Tool pull-down menu. To have the Tool dialog reappear, just click on the Tool Palette menu item again.

Screen 1.2 shows a sample of all the different types of objects that can be drawn with the graphic editor. Note that many of these objects have different orientations and attributes associated with them. As you will soon see, there are many ways to edit the objects and save the work that you create by using this application. To understand all the options available with this graphic editor, a review of all the functions available from the pull-down menu items is necessary.



SCREEN 1.2 Sample object types created with the graphic editor.

File Options

The File pull-down menu has several items that allow you to save and retrieve graphics information as well as start new drawings or exit the application. Following is a list and description of all the items found in the File pull-down menu.

1. **New** – This menu item is used when you want to start a new drawing. It will reset all information from the previous drawing and erase the client area. If you made changes to a drawing that you want to save before you start a drawing with the New option, be sure to use the Save or Save As option before using the New option.

2. **Open** – This menu item will retrieve information from a drawing you saved earlier and make it the current drawing that can be edited. When you use the Open menu item, another dialog box will prompt you to enter the name of the file in which you saved an earlier drawing. Note that drawings saved with this editor have a default file extension of WLY and these files are saved with our own file format. If you use the Open option while another drawing is in the client area, the current drawing will be replaced with the drawing found in the WLY file.

3. **Save** – This menu item will save the current drawing to the current filename. The current filename is the name of the file that was opened to retrieve the current drawing into the client area. If the current drawing was not retrieved via the Open menu item, then you must use the Save As menu item to establish a current filename and save the drawing for the first time. When the drawing information is saved, it is saved in a format particular to this application. Furthermore, this application can only retrieve information saved in this special format with the Open menu item.

4. **Save As** – This menu item will save the current drawing to a new filename. Note that the default file extension for drawing files is WLY. You can, however, override the file extension to whatever you like (even though this is not recommended). Once you use the Save As menu item, the current filename is updated to the name you chose with this option.

5. **Capture TIFF** – This menu item allows you to capture either the client area of the drawing or the entire OS/2 desktop into a TIFF file format. When you select this menu item, another dialog appears that allows you

to choose which area to capture, what dimensions the image should be rendered to, and the name of the file in which you want to save the TIFF data. The default extension of the file is TIF, however, you can override this extension to whatever extension you like (even though this is not recommended). By capturing image data using the TIFF format, you can import this image into other major art tools and word processors. This option was used to produce all the screen captures for this book.

6. **Export MetaFile** – This menu item allows you to export the current drawing to an OS/2 MetaFile. When you use this option, another dialog prompts you for the name of the file to which you want to export the drawing. The default extension of this file is MET, however, you can override this with whatever extension you like (even though this is not recommended). By exporting your drawing to an OS/2 MetaFile format, you will be able to import this drawing into other commercially available drawing applications.

7. **Import WLY** – This menu item will retrieve information from a drawing you saved earlier and make it part of the current drawing where it can be edited. When you use the Import WLY menu item, another dialog box prompts you to enter the name of the file in which you saved an earlier drawing. Note that drawings saved with this editor have a default file extension of WLY and these files are saved with our own file format. When a drawing is imported, it is grouped together and placed in the foreground of the current drawing. You can, however, ungroup this object and move its components any place in the Z order you like. (Z order refers to the order in which objects are drawn, hence, which object is drawn on top of another object.)

8. **Print** – This menu item allows you to print the current drawing to a printer known to your OS/2 system. When you select this menu item, a list of printers to which you can route the drawing is displayed. You simply pick the printer where you want the print information to be routed. You will then be prompted with other dialogs to set up the print job. These other dialogs are designed specifically for the chosen printer.

9. **View Selected Objects** – This menu item allows you to see how selected objects in the current drawing were generated using the GPI. When this option is selected, the objects are parsed and their order information is output into English text to a special file in the current directory

(WILEY.SEG). This special filename is then automatically passed to a generic text browser which pops up and displays the text in its own browse window. The View Selected Objects menu item will stay disabled until the browse window is closed. Note that both the browse window and the draw application can work concurrently.

10. **Product Information** – This menu item causes a dialog box to display title and copyright information for this application.
11. **Exit** – When this menu item is selected, the draw application is terminated. Before using this menu item, ensure that all relevant drawing information has been saved.

Edit Options

The Edit pull-down menu has several items to assist you in the editing of existing objects. Following is a list and description of all the items found in the Edit pull-down menu.

1. **Delete** – This menu item deletes the selected objects from the drawing. No information about the deleted objects is saved.
2. **Cut** – This menu item cuts the selected objects from the drawing. However, the object information is saved so the objects can be pasted to another location within the same drawing at a later time. Once a subsequent cut operation is performed, the previous object information is lost; hence, the previous object can no longer be pasted.
3. **Copy** – This menu item produces a copy of the selected objects in the same drawing which can be pasted at a slight offset from the original selected objects. Once this is done, the copy can be edited independently from the original. (The copy is held in the paste list.)
4. **Paste** – This menu item adds the objects found in the paste list to the current drawing.
5. **Group** – This menu item takes all the selected objects and encapsulates them so further editing operations act on all the grouped objects as if they were one.
6. **UnGroup** – This menu item returns previously grouped objects to their previous independent state.

7. **Select All** – This menu item places all objects in the current drawing into a selected state.
8. **Fast Correlate/Draw** – This menu item causes the graphic editor to use an alternate method for selecting objects and drawing them. This alternate method is not as functional but much faster than true correlation.

Metrics Options

The Metrics pull-down menu has several items to assist you in creating and examining objects to exact dimensions. Following is a list and description of all the items found in the Metrics pull-down menu.

1. **Tool Meter** – Most edit tools have a meter that shows the exact position and metric information about an object as it is being created or modified. For instance, the line's tool meter will show the starting location, length, and angle of a line as it is being manipulated. This menu item allows you to toggle the tool meter on and off. (All metric units are in inches.) Area information for rounded boxes is calculated as though they are regular rectangles.
2. **Grid** – This menu item allows you to produce a grid as a backdrop on the drawing area. This grid can be a useful guide when creating or editing objects. This grid is not saved when the drawing is saved and is not printed when the drawing is printed. The grid size is controlled through the Set Grid Size menu item.
3. **Snap To Grid** – When this menu item is activated, it makes edit functions always align with the current grid setting. Note that the grid does not have to be visible for this menu item to operate.
4. **Set Grid Size** – This menu item allows you to set the size of the grid. When Set Grid Size is selected, another dialog is displayed that allows you to set both X and Y grid line spacing. The grid line spacing is measured in units of tenths of inches. This menu item works in conjunction with the Grid and Snap To Grid menu items.
5. **Zoom** – This menu item allows you to scale the entire drawing to a new size. When this menu item is selected, another dialog appears that will let you choose the percentage increase in the drawing size from the origi-

nal size. This menu item can be useful when working with small objects or dimensions.

Tools Options

The Tools pull-down menu has several items that let you select to current tool, as well as determine if the Tool dialog box is visible on the desktop. Following is a list and description of all the items found in the Tool pull-down menu.

1. **Tool Palette** – This menu item determines if the Tool dialog box will be visible on the desktop. If this menu item is checked, then the Tool dialog box is made visible on the desktop.
2. **Select tool** – If this menu item is selected, then the Select tool becomes the current edit tool. Once this item is selected, the Select tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Select tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Select tool from the Tool dialog box.
3. **Fillet Fill tool** – If this menu item is selected, then the Fillet Fill tool becomes the current edit tool. Once this item is selected, the Fillet Fill tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Fillet Fill tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting Fillet Fill tool from the Tool dialog box.
4. **Line tool** – If this menu item is selected, then the Line tool becomes the current edit tool. Once this item is selected, the Line tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Line tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Line tool from the Tool dialog box.
5. **Pencil tool** – If this menu item is selected, then the Pencil tool becomes the current edit tool. Once this item is selected, the Pencil tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Pencil tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Pencil tool from the Tool dialog box.

6. **Rounded Box tool** – If this menu item is selected, then the Rounded Box tool becomes the current edit tool. Once this item is selected, the Rounded Box tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Rounded Box tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Rounded Box tool from the Tool dialog box.

7. **Box tool** – If this menu item is selected, then the Box tool becomes the current edit tool. Once this item is selected, the Box tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Box tool icon will become highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as if the Box tool were selected from the Tool dialog box.

8. **Ellipse tool** – If this menu item is selected, then the Ellipse tool becomes the current edit tool. Once this item is selected, the Ellipse tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Ellipse tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Ellipse tool from the Tool dialog box.

9. **Arc tool** – If this menu item is selected, then the Arc tool becomes the current edit tool. Once this item is selected, the Arc tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Arc tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Arc tool from the Tool dialog box.

10. **Text tool** – If this menu item is selected, then the Text tool becomes the current edit tool. Once this item is selected, the Text tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Text tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Text tool from the Tool dialog box.

11. **Fillet tool** – If this menu item is selected, then the Fillet tool becomes the current edit tool. Once this item is selected, the Fillet tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Fillet tool icon is highlighted and the other edit tool icons are

not highlighted. Choosing this menu item has the very same effect as selecting the Fillet tool from the Tool dialog box.

12. **Polyline tool** – If this menu item is selected, then the Polyline tool becomes the current edit tool. Once this item is selected, the Polyline tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Polyline tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Polyline tool from the Tool dialog box.

13. **Polyfill tool** – If this menu item is selected, then the Polyfill tool becomes the current edit tool. Once this item is selected, the Polyfill tool menu item is checked and the other edit tools are unchecked. If the Tool dialog box is visible, the Polyfill tool icon is highlighted and the other edit tool icons are not highlighted. Choosing this menu item has the very same effect as selecting the Polyfill tool from the Tool dialog box.

Styles Options

Many of the objects that you create or edit can have several attributes associated with them. For instance, a box object can have a line thickness, a line type, a line color, a fill pattern, and a fill color associated with it, whereas a text object can have a font name, an emphasis, a color, and a point size associated with it. The Styles pull-down menu has several items that let you choose the attributes of the object you are creating or editing. Following is a list and description of all the items found in the Styles pull-down menu.

1. **Font** – This menu item determines the current font information that will be associated with text that is about to be edited or created. When this menu item is selected, a dialog box appears and prompts you for a font name, emphasis, and a point size. Note that this application only allows vector fonts to be specified. By doing this, all text can be rotated, sheared, scaled, and so on, like any other graphic object.

2. **Fill Color** – This menu item determines the fill color to use with closed object like a box, rounded box, or an ellipse. The fill color choices will be displayed in another dialog box as soon as this menu item is selected. This fill color will be associated with closed objects that are about to be edited or created.

3. **Fill Pattern** – This menu item determines the fill pattern to use with a closed object such as a box, rounded box, or an ellipse. (Examples of fill patterns are solid, none, vertical lines, diagonal lines, etc.) The fill pattern choices will be displayed in another dialog box as soon as this menu item is selected. This fill pattern will be associated with closed objects that are about to be edited or created.

4. **Line Type** – This menu item determines the type of line to use when drawing an object. (Examples of line types are solid, invisible, dashed, dot–dash, etc.) The line type choices will be displayed in another dialog box as soon as this menu item is selected. This line type will be associated with all objects that are about to be edited or created. Note that if the line thickness is greater than the thinnest line, the line type will always appear solid.

5. **Line Color** – This menu item determines the color of line to use when drawing an object. The line color choices will be displayed in another dialog box as soon as this menu item is selected. This line color will be associated with all objects that are about to be edited or created.

6. **Line Thickness** – This menu item determines the thickness of line to use when drawing an object. The line thickness choices will be displayed in another dialog box as soon as this menu item is selected. This line thickness will be associated with all objects that are about to be edited or created. Note that if the line thickness is greater than the thinnest line, the line type will always appear solid.

7. **Box Radius** – This menu item allows you to set the percent of the X and Y axis used for the rounded box. (100 percent on both the X and Y axis will cause an ellipse to be produced while 0 percent on both X and Y axis will produce a box.) If some rounded boxes are selected, this will cause their rounding percentages to be updated immediately. Otherwise, the new values set will be used for rounded boxes as they are created.

Transforms Options

Once you create an object, you may want to manipulate it by changing its placement in the Z order, rotating it , shearing it, and so on. (Z order refers to the depth or layer that the object is drawn in the drawing. Note that objects in

this environment can overlap.) The menu items found in the Transforms pull-down menu provide you with many function that do these type of transformations. Following is a list and description of all the items found in the Transforms pull-down menu.

1. **Front** – This menu item moves the selected objects to the front or top of the drawing.
2. **Back** – This menu item moves the selected objects to the back or bottom of the drawing.
3. **Forward 1** – This menu item moves the selected objects toward the top of the drawing by one position or level.
4. **Backward 1** – This menu item moves the selected objects toward the bottom of the drawing by one position or level.
5. **Scale** – This menu item allows you to resize the selected objects. Once this menu item is selected, another dialog box appears and prompts you to enter a percentage which represents the amount you want the objects to be resized.
6. **Rotate** – This menu item allows you to rotate the selected objects. Once this menu item is selected, another dialog box appears and prompts you to enter the number of degrees you want the objects rotated.
7. **Shear** – This menu item allows you to shear the selected objects. (A shear will cause the object to slant from vertical a certain amount in degrees.) Once this menu item is selected, another dialog box appears and prompts you to enter a shear angle.
8. **Flip Horizontal** – This menu item allows you to flip the selected objects horizontally.
9. **Flip Vertical** – This menu item allows you to flip the selected objects vertically.
10. **Set Model Transform** – This menu item allows you to change a special matrix called the model transform matrix for a selected object. This option is a special learning tool to help programmers understand what transformation matrices are and how they work. Before you use this option, you should read Chapter 7, "Transforms." Once this menu item is selected, another dialog box with the model transformation matrix appears and lets you change its values.

Mouse Actions on Selected Objects

Besides using the functions available from the pulldown menus, you can also use the mouse to manipulate selected objects. These manipulations include moving and resizing objects. But, as stated earlier, before you can move or resize an object, it must first be selected. The way to select one or more objects is with the Select tool. Remember, selected objects can be recognized in the drawing because markers will appear about the objects.

Besides helping to identify selected objects in a drawing, markers also serve as a way to resize or reshape an object with the mouse. To resize or reshape an object, simply press and hold mouse button 2 on the marker and then move the mouse in the direction in which you want the object to be resized. When the object reaches the new size, just release mouse button 2 and the resizing operation is complete.

To move selected objects, just point the mouse at any of the selected objects **other than on a marker** and simply press and hold mouse button 2. Then move the mouse in the direction in which you want the object to be moved. When the selected objects are in the desired new location, release mouse button 2.

As you use this graphic editor, you will find that the shapes, fonts, and attributes that you can manipulate allow you to produce some fairly detailed and complex drawings. In fact, all of the art work in this book was produced using this graphic editor. You will also find that the software written to produce this editor is **not** very complex (even though this book comes with several thousands of lines of source code). This graphic editor was produced by the authors of this book in a relatively short period of time. What allowed us to write this application in such a productive fashion was the vast amount of function available in the GPI and a good program structure.

This book assumes you already understand the C programming language and the basics of Presentation Manager programming; hence, we will not focus on things like what a Presenation Manager message looks like, or how to create a window, or what a window or dialog procedure does and how they are structured. Instead, we will focus on the different GPI functions and show how they may be used to solve some fairly complex problems that you may bump into in your own graphic programming. Even though we will not review a lot of Presentation Manager basics, a high level overview of the graphic editor program is still appropriate.

OVERVIEW OF GRAPHIC EDITOR PROGRAM STRUCTURE

The graphic editor program has the same basic program structure as most Presentation Manager applications. It has a main application window and its own window procedure, resources such as menu items, icons, and pointers, and numerous dialog boxes so the user can pass information to the application interactively. Like most Presentation Manager programs you've probably seen, this program is written in C (using IBM®'s C Set/2 32-bit compiler). We had thought about writing this application in C++ because we believed it to be well suited for using the object oriented features C++ offers. We decided, however, that the C programming language is currently better known by most programmers and that it would enable us to demonstrate the functions of the GPI just as well.

If you are into object oriented programming or object oriented programming concepts, you have probably noticed similarities between Presentation Manager programming and object oriented programming, for instance, the concept of classes of things such as window classes, the use of messages to communicate event data, subclassing of window procedures, and so on. If you are not familiar with object oriented programming, you will probably find it much easier to learn after you have done some Presentation Manager programming.

One of the major themes of object oriented programming is that an object knows about itself and contains much of the data it needs to maintain itself or provide services. In our application each tool is like an object and provides services based on the facts it knows about itself. For instance, if the Line tool happens to be the current tool and a mouse move message is received while in draw mode, the Line tool will process the mouse move message and draw the line correctly based on what it currently knows. If the current tool were the Box tool, however, a different set of operations (known by the Box tool) would draw a box correctly. Hence, each tool knows about itself and knows how to process messages that are sent to it. However, each tool only knows how to process messages in the context of the services provided by that tool. Therefore, when our main client window procedure receives a message (such as mouse move or button up or down), that message is routed to the current tool which processes the message in its own correct context. The way messages route to the current tool and how each tool does

the right thing will become more apparent when we look at source code later in this book.

When you take a closer look at the tool code (`TOOL.C`), notice that there are actually only two classes of tools. These are the Select tool class and the Graphic Object tool class. The Select tool class is only used by the Select tool which has its own set of unique behavior (selecting and deselecting objects). The graphic editor uses only one instance of the Select tool class. The Graphic Object tool class is more interesting. Essentially, there is one instance of this Tool class for each graphical editing tool in our tool bar (except the Select tool). When a Graphical Object tool is created, the kind of graphical objects that tool is to make are specified (for instance, a box, text, line, ellipses, etc.). This information is then stored inside the Tool object. Thus, when the tool processes an event that is supposed to create a new graphical object, the Tool object knows what kind of graphical object it should create.

So, how could we create one generic Graphical Object tool that knows how to create and edit all the different kinds of graphical objects we support? One way would be to make a big monolithic Graphical Object tool that knows how to do all the operations against all the different kinds of graphic objects. We felt that such an approach would quickly lead to a huge web of unmanageable code that would be difficult to maintain and extend. So instead we took the approach that the graphical object knows best how to handle events for creating and editing itself. Therefore, we introduced another layer of message routing in which the tool routes the events to the Graphical Object. The Graphical Object then determines what the editing operations are and what their output will be. The Graphic Object tool is still responsible for things like starting the creation of an object, updating the display and select lists, and refreshing the display; however, it is the responsibility of the Graphical Object to do specific processing, such as editing. (For example, an object type of polyline is responsible for editing points within a polyline object.) Since editing activities are specific to the type class of graphic object being edited, events are routed directly to that object's event processing routine (method) for detailed processing. As you can see, this approach simplifies the tool's job and lets it focus on things for which it is responsible (creating objects, and managing how those objects are maintained within the editor). In addition, this approach bundles the object-specific processing with the object type, which is where this processing is most efficient. This

packaging of function makes the code easier to understand and easier to maintain.

Another major design concept of our graphical editor program is how it uses lists. When an object such as a line or box is created, attributes are associated with the object, and it is added to a draw list. (Attributes are the set of information needed to create the object in the drawing. This information includes position, line type, color, rotation, shear, scale data, fill pattern, line width, and so on.) Objects are also given unique identifiers so they can easily be found in the draw list or drawing and then be manipulated. Lists also hold other types of information for the graphics editor program. For instance, when you select objects to be manipulated, their identifiers are placed in a select list. By having this select list, it is easy for the graphics editor program to manipulate this list of objects and apply the changes back to the draw list.

By knowing the basic structure of a typical Presentation Manager program, the concept of messages being routed to the currently selected tool, and the concept of using lists to hold object information, you have the basic knowledge of our high-level program design. It is now important to understand where the basic functions of the application are located in the source code and what source code components are used to build the different objects that make up our application. Besides the objects that make up the `DRAW.EXE` component, the draw application also uses one dynamic link loaded (DLL) component too. This DLL runs in its own thread of execution and provides a generic ASCII file Browse utility. Later you will see all the parts used to build the Browse utility. Following is a list describing what each object in the program provides:

- `DRAW.OBJ` – This object has the MAIN entry point for the Draw program. It creates and positions the application window, parses input parameters, obtains system values about the application environment, dispatches messages to the main application window procedure, contains the draw program window procedure, provides the application termination logic, and contains all the program global variables.

- `FUNCS.OBJ` – This object provides all the dialog procedures and miscellaneous functions used by the Draw program. These miscellaneous functions include set point size, draw the background grid, add a font to the drawing, set a widths table for a character string, and so on.

- `ATTR.OBJ` – This module contains functions that manipulate object attributes. All graphical objects contain a common block of attributes. The functions in this package are used to initialize, set and get the values of those attributes. It will also issue the proper GPI functions to establish a given object's attribute block as the current attributes.
- `EDIT.OBJ` – This module contains functions that are used for common editing operations. These functions include Delete, Cut, Paste, Group, and Ungroup. It also contains functions for refreshing all or part of the display.
- `GOL.OBJ` – This module contains functions for creating and manipulating lists. Lists are treated as objects in the graphic editor program; therefore, lists have routines to create them, destroy them, and perform functions with them. For instance, routines (methods) are provided for operations such as adding entries, deleting entries, and retrieving entries. Entries in a list are restricted to graphical objects. Each list is also associated with a retained segment. This segment contains calls to the retained segments of objects that are currently in the list. (Retained segments are discussed in Chapter 5, "Building Blocks of the GPI".)
- `OBJECT.OBJ` – This module contains functions for creating and manipulating graphical objects. This file defines several kinds of graphical objects. A single generic class of graphical object exists and then specific kinds of generic objects are derived (sub-classed) from that generic one. The generic graphic object class has a set of routines (methods) for manipulating any graphical object. These methods include Rotate, Scale, SetAttribute, Dragging, and so on. Specialized (sub-classed) graphic objects include ellipses, rectangles, text, and so forth. The specialized graphic objects must support the same methods as the generic object class. To do this, the sub-class can either use the generic methods (inherit them) or override them if a special function is required.
- `TOOL.OBJ` – This module contains functions for creating and processing various kinds of editing tools. Like the graphic objects, there is a generic tool class and then specific kinds (sub-classes) of tools. Each subclass of tool performs a specific task. Currently the graphic editor only supports two classes of tools (Select tools and Graphic Object tools). Future classes of tools could include Rotation tools, Sizing tools, Eraser tools, and so on.

- PARSESEG.OBJ – This object receives graphic segment information and starts parsing and formatting this information for the View Selected Objects feature.

- PORDERS.OBJ – This object is used by the PARSESEG.OBJ object to parse and format the graphic order data found within a graphic segment.

- WRITETIF.OBJ – This object provides screen or window capture ability. Once an image is captured, this object then writes the image information to a file in a TIFF format.

Now that you've had a high-level view of the graphics editor design and structure, let's trace (at a high level) a few sample events through the program to help you feel more comfortable with its logic flow. For the first example, let's look at what happens when a user has mouse button 1 pressed and is moving the mouse when the current tool selected is the Ellipse tool.

First, realize that when the draw program is started, all the various tools are created so they will be ready to use when the user chooses them from the Tool palette. This would be analogous to a plumber putting tools in the toolbox before going on a house call. To start with, the user first moves over and chooses the Ellipse tool from the Tool palette. When this occurs, a CHANGE_TOOL message is sent from the Tool palette dialog to the main draw window procedure. This results in the draw program selecting the Ellipse tool as the "current tool." Think of this as the plumber picking up his pipewrench (he's not doing any work yet, but it's now the current tool that he's using). Next, when the mouse button 1 down message is received by the draw routine (DRAW.C), it automatically sends it to the current tool for processing (TOOL.C). Hence, in our case, the Ellipse tool receives the message and takes a look at the event. It so happens that the policy of the Graphic Object tool is that mouse button 1 down means we should create a new object. Therefore, the Ellipse tool constructs a new graphic object of type "ellipse." When the ellipse object is created, it is still in an undefined state and needs more messages to complete its definition. The Ellipse tool records the new Ellipse object as being the "currently edited" object. As additional messages are passed to the Ellipse tool, the Ellipse tool forwards them to the Ellipse object (OBJECT.C). Then, as the user drags the mouse to create the ellipse shape, the mouse movement messages are passed to the Ellipse tool and then to the Ellipse object. The Ellipse object understands that it is still in

initial definition state and draws itself in a rubberband form (XORed). When the Ellipse object receives a button 1 up message, it knows that its definition has been completed. Hence, it returns a status to the Ellipse tool indicating this fact. When the Ellipse tool receives this indication, it records the fact that the current Ellipse object is no longer being edited. Hence, future messages will no longer be passed to that object (unless the Ellipse tool decides that the object is processing input again).

Notice that all of the object-specific details, such as defining the points of the object and drawing the object, are handled within the object code and not the tool code. Similar processing is performed when a user has the Select tool, selects an object, and then edits the object by dragging a marker to one of its defining points.

For our second example, let's look at the program flow when the currently selected object is text and the user selects the font pull-down menu item. In this example, the user has selected a text object that was previously created. To do this, the user first chose the Select tool; thus, as Presentation Manager messages are generated, they are routed to the Select tool first (`TOOL.C`). As in the previous example, the button1 down message is processed by the Select tool. Unlike the Ellipse tool, however, the Select tool policy indicates that this is a select operation. Therefore, the Select tool examines which graphical object the mouse is positioned over and selects it by adding the object to the select list (`GOL.C`). Next, the user brings up the font dialog and chooses a new font and point size. When the user presses okay, a message is sent back to the Draw program (`DRAW.C`) to indicate the change. Like other messages, this message is passed to the Select tool; but, since the Select tool doesn't know what to do with it, it returns the message to the main draw routine as UNHANDLED. The main draw routine then processes the message. To handle the event, the main draw routine (`DRAW.C`) recognizes the message as a change of an object attribute. These types of events are typically applied to all the objects that are currently selected. The editor then loops through each object currently in the select list and calls its setAttribute method twice(`ATTR.C`): once for the setting object's font and once for setting the object's point size. The object thus records its new font and point size attributes and also updates its display segment so it looks correct when the next screen refresh is done. Each object has an attribute block associated with it. That attribute block is defined in `ATTR.H` and functions for setting

and getting the values of attributes in the block are in the ATTR.C file. As the editor updates the attributes of the selected objects, it also records the regions of the screen that need to be updated. Once all selected objects have been processed, the editor refreshes the affected regions on the display.

For our last example, let's see what happens when the user has a few items selected and then selects the Cut pull-down menu item. Because these items are selected, there are references to them in the select list. When the user selects the Cut pull-down menu item, a WM_CUT message is sent to the Draw program (DRAW.C). As in the previous example, the Select tool returns UNHANDLED so the draw program examines this request and recognizes it as a cut operation. The cut operation is slightly more complicated than other operations recognized by Draw, so we have put its implementation in a separate module (EDIT.C) with similar edit operations. The cut function first clears the current contents of the paste list. It then copies all the items currently in the select list over to the paste list. Next, the items in the select list are removed from the display list and also from the select list. Once the cut operation is complete, the objects that were previously displayed and in the select list are removed and placed into a special "paste" list. The objects will remain in the paste list until the next cut operation (at which time they will be deleted) or until the next paste operation(at which time they will be copied and inserted back into the display and select lists).

As you can see, most of the actions performed by the cut operation involve using list functions. These operations are fairly straightforward with one potential curiosity: When an object is inserted in the select list, the select list in turn calls the object and tells it that it is now selected. Likewise, when the objects are removed from this list, it also calls the object and informs it that it is no longer selected. Thus, by simply adding an object to the select list, the object is informed that it is selected and changes its appearance accordingly. In particular, the object will put a dashed box around itself and display any edit handles.

Another thing to note about the select list is that it merely contains references (pointers) to objects that are in the display list. It does not point to a separate copy of the object. The paste list, however, always contains a unique set of objects. Objects in this list are not contained in the display or select list.

WORKING WITH THE DISKETTE

This book comes with a diskette containing both the source code and the executable program objects for the graphic editor application discussed in this book. This diskette also contains source code and the executable program objects for an OS/2 MetaFile parser application, a query printer information application, and a generic text browser application that is used by the graphic editor, the query printer information, and the OS/2 MetaFile parser applications. By having all this source code, you can manipulate these applications and see the effects of your actions. Furthermore, because the source code is included, you are saved from the tedious chore of entering, compiling, and debugging all these programs yourself. Finally, if parts of the source code are similar to what you are developing for your own project, you may use those parts for your project. **You can not, however, resell any part of this diskette content in the context in which it was given to you.**

Important: Be careful to understand and test all source code used from this diskette as part of your own project. There is no guarantee that this code will meet all of your needs or that it does not contain errors.

Before you make modifications to any of the source code that comes with this book, make a copy of the diskette and store it in a safe place. This will insure that you always have access to the original source code in case your working copy becomes lost or damaged.

After you have made a copy of the book diskette, you may choose to install the contents of the diskette on the hard drive of your OS/2 system. To install the book software and source code on your hard file, refer to Appendix B.

As stated, the installation includes the book software and source code from the book diskette, hence, many more files are copied to your new directory than are really needed just to run these programs. Shortly, we will list each of the files on the book diskette and describe their purpose, but first let's finish making our application installation complete by making the application icons appear on the desktop.

To get the application icon for the graphic editor application on the desktop, copy a program template from the template folder found on your desktop. Then, when the notebook dialog appears, reference `DRAW.EXE` from the directory you just created and close the notebook dialog. To get the application icon for the OS/2 MetaFile parser and query printer information

applications on the desktop, perform the same task but this time reference `VIEWMET.EXE` and `PRINTERS.EXE` from the directory you just created. To test if these programs are working, simply double click on the new program icons placed on your desktop. If they are installed correctly, you will see their application windows appear on your desktop.

The graphic editor and OS/2 MetaFile parser applications can have filenames passed to them when they are started. If they do, they will automatically open these files and draw their contents. The files to be opened, however, have to be of a particular type or they will not appear to have an effect on the application startup. In the case of the graphic editor, this file must have been saved by the editor. (The graphic editor saves graphic data in its own file format.) Typically, the file extension for this type of file will be WLY. This is the default extension we give to graphic files saved with the graphics editor. For the OS/2 MetaFile parser, the file to be displayed must be a valid OS/2 MetaFile. In most cases, an OS/2 MetaFile will have an extension of MET. Because these applications can take a filename as an input parameter on startup, you can drag and drop these graphic files on the application icon and this will also start the application program. Or, lastly, because the file extensions of the graphic files are usually WLY or MET, you can make an association of these file types with the application name and start the application. (This association is made via the notebook dialog that was present when we copied a program template for our applications. You can get this notebook dialog back again by doing an open on the application icon found on the desktop.) For example, if you make an association between WLY files and DRAW.EXE, any time you double click on a WLY file, the graphic editor application will start.

What we haven't discussed is how to modify information in CONFIG.SYS to find the different files types for the book programs. This is because we have placed all the book diskette contents in the same directory and assumed the programs will be started from that directory. If this is true, the different file types needed to make these programs work will automatically be found. You can put these book programs in separate directories and move some of the DLLs and HLP files to other standard directories; if you do this, however, you may have to modify the path information in CONFIG.SYS so the DLLs and HLP files can be found. Tables 1.1, 1.2, and 1.3 show the minimum set of files needed to run the three applications that come with the book:

TABLE 1.1 Graphic editor files

File Name	Description
DRAW.EXE	Main program file for the graphic editor application.
DRAW.HLP	Help text for the graphic editor application.
BROWSE.DLL	Dynalink object that will display an ASCII text file that is passed to it. This is used by the graphic editor program to display the contents of the file created by PARSESEG.DLL.
BROWSE.HLP	Help text for the generic file browser, BROWSE.DLL object.

TABLE 1.2 OS/2 MetaFile parser files

File Name	Description
VIEWMET.EXE	Main program file for the OS/2 MetaFile parser application.
VIEWMET.HLP	Help text for the OS/2 MetaFile parser application.
BROWSE.DLL	Dynalink object that will display an ASCII text file that is passed to it. This is used by the OS/2 MetaFile parser program to display the contents of the file created by PARSEMET.DLL.
BROWSE.HLP	Help text for the generic file browser, BROWSE.DLL object.

There are many more files on the book diskette than are listed in the previous three tables. These are the source files used to build all of the program objects for this book. Appendix B shows all the files needed to build the different program components for this book. As you build the program objects, the build process will generate even more files. (OBJ, LIB, and RES files). All program objects that come on the book diskette were built using IBM's Development Toolkit, C SET/2™, LINK386, Resource Compiler, and IPF Compiler. To see how these program objects were built with this tool set, refer to Appendix B.

TABLE 1.3 Query printer information files

File Name	Description
PRINTERS.EXE	Main program file for the query printer information application.
BROWSE.DLL	Dynalink object that will display an ASCII text file that is passed to it. This is used by the query printer information program to display the contents of the file it created that contains the printer information.
BROWSE.HLP	Help text for the generic file browser, BROWSE.DLL object.

Before you do anything else, first play with the graphic editor program to become familiar and comfortable with its function. Second, set up your development environment so you can build and execute all programs provided with this book. Finally, at any given time throughout the reading of this book, experiment with the source code to enhance your learning of the different parts of the GPI.

SPECIAL NOTES ABOUT THIS BOOK

Most listings in this book are excerpts from the source code found on the book diskette. Very often several statements are left out of these listings to help maintain focus on the important points about a particular topic. Therefore, even within a routine, several lines of source code may be missing or declarations may not be shown for variables used. Even though these listings are often a small subset of the total lines of code for a routine, they are still designed to show continuity on a routine basis. If you want to see an entire routine instead of the excerpt, browse the source code found on the book diskette.

Finally, the first time a Presenation Manager or GPI function is mentioned in this book it is *italicized* to help bring it to your attention. As functions are introduced, a brief explanation about what the function does and how it is used by our program is given. Sometimes, the function parameters are discussed in detail and other times they are not. The parameters are only

discussed in detail if this discussion adds to the current topic. Remember, this book is not a replacement for the technical reference material that comes with the OS/2 Toolkit. For a complete explanation of all the parameters for a function and a total list of OS/2 and Presentation Manager functions, refer to the IBM technical reference material.

CHAPTER 2

OS/2 Architecture

Do operating systems sell PCs? I don't think so. For the vast majority of us, applications sell PCs. After all, without applications, a PC is little more than a technological trinket that collects dust. An operating system is, however, the platform on which applications are built. Therefore, the more robust the operating system, the easier it is for software developers to produce top-notch applications. Of course, people have different opinions of what a top-notch application may offer and these opinions change with time.

Some years back, a friend of mine who worked with PCs on a daily basis (as a non-programmer) told me he could see no reason for moving from DOS to OS/2. All he wanted to do was to be able to run more of his applications on the PC at the same time without having memory problems. He also told me he was concerned about the stability of his PC software when he loaded multiple terminate and stay resident (TSR) programs in his PC. Gee, it sure seemed like he was describing why he needed OS/2. Then it suddenly hit me. IBM had not been advertising OS/2 the correct way. IBM had been talking about things like OS/2's advanced multitasking, memory management, and user interface design, but they had not demonstrated these items in a way that let average PC users realize the problems OS/2 would solve. After all, my mom is a fairly good user of word processors and spreadsheets, but she is not a computer scientist and doesn't have a clue as to why multitasking should be important to her. In fact, multitasking isn't directly impor-

tant to her, but it should be important to developers of the application software she uses. Unfortunately, application developers weren't sold on many of these items either (at least at the cost of developing and marketing a product for the OS/2 environment versus the security they felt with DOS). In short, IBM hadn't adequately shared the OS/2 vision with their potential customers and application developers. (*Build it and they shall come.... NOT!*)

In all fairness to IBM, it was and still is a tough job to get people to change from the comfort of DOS to something different. Particularly if that something is perceived as being more complex and expensive than what they currently use. In reality, OS/2 is **not** more expensive for people who have a newer PC (Intel® 80486 based systems) and in many ways is much simpler than DOS. But like any change, a need for change must usually be felt before it will occur. In the case of changing operating systems, this need will probably be felt because of the presence of a killer application that uses that new operating system. Again, applications sell PCs, not operating systems.

Today we are seeing a lot more native OS/2 applications becoming available. We're also seeing how IBM has done an outstanding job allowing DOS applications to work in the OS/2 environment. Because of these facts, millions of people are taking that jump from the comfort of DOS and becoming hooked on OS/2. In fact, most people that have made the jump to OS/2 can not imagine returning to DOS. This is because OS/2 provides an environment where multiple applications can concurrently operate and share resources of the PC without conflicting with one another. Furthermore, the user interface for OS/2 is much more intuitive than DOS and this interface is very consistent between most OS/2 applications. Finally, the level of stability for the system is extremely high.

The more I use OS/2, the more I am impressed with the vast amount of features it offers. This opinion is not only from a user's point of view, but also from a programmer's point of view. Because this book is about the features that OS/2 brings to programmers who are developing advanced graphics applications, we're going to describe (at a high level) the architecture OS/2 provides to enable this to happen.

Figure 2.1 shows a conceptual view of the OS/2 architecture we are about to discuss. It is drawn from the point of view of a graphical application interfacing to the OS/2 graphical environment and shows the logical layers of function or data structures used to draw output on a physical device. As

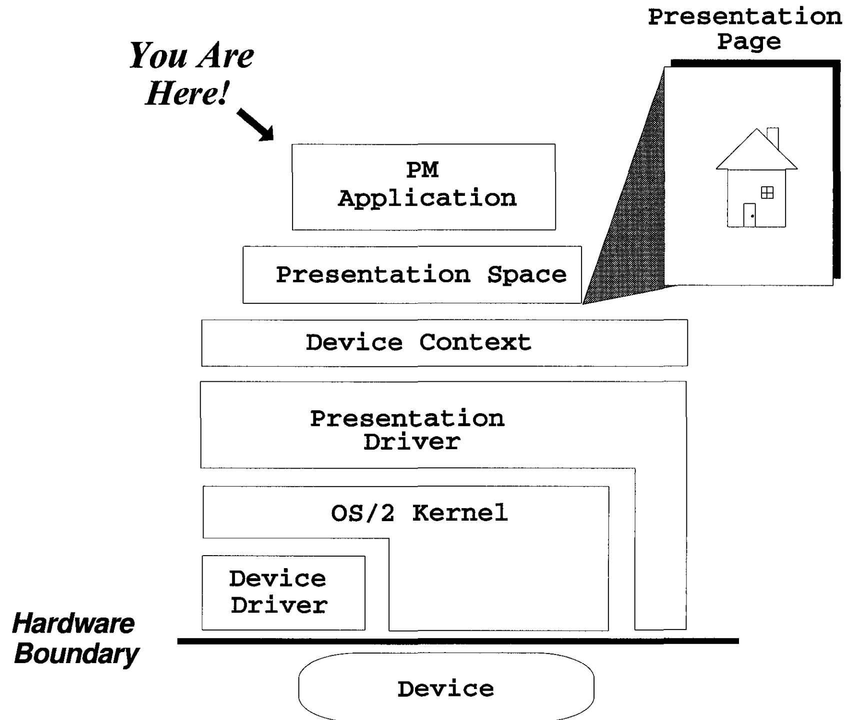


FIGURE 2.1 Conceptual view of OS/2 architecture.

you read the following section, refer to this figure to better understand how these layers relate to each other.

OS/2 is designed in such a way that programmers can develop graphical applications that are device independent. This design allows the programmer to define an electronic sheet of paper called a *presentation page*. This presentation page can be thought of as the target for your drawing orders and is defined via a data structure called a *presentation space*. It is the handle to the presentation space that allows you access to your presentation page. The presentation space not only holds the definition of the presentation page, but it also hold the current drawing environment attributes. The key point about presentation space data is that it is typically device independent.

Actually, there are are three types of presentation space that a program can use in a Presentation Manager application. The smallest and simplest of the presentation space types is called a *cached-micro presentation space*. It

is called a cached-micro presentation space because OS/2 keeps a pool of these presentation space data structures around for quick access. The cached-micro presentation space is designed to be used for a brief period of time (like on a `WM_PAINT` message) and then returned backed to the system. Each time you obtain a cached-micro presentation space it will have default drawing attributes associated with it; therefore, if you have attribute requirements that are not the defaults, you must set them each time you obtain a cached-micro presentation space. This presentation space is also intended for use only with display device types (window context).

A *micro presentation space* is the next biggest and more powerful presentation space. The micro presentation space is intended to be owned by an application for as long as it is needed. It can be associated with multiple types of output devices; however, a presentation space can only be associated with one output device type at a time.

The most functional and largest presentation space is called the *normal presentation space*. The normal presentation space has all the properties of the micro presentation space but allows for a function called *retained graphics*. Retained graphics will be explained in more detail later in this book, but the basic idea is that the presentation space has the ability to record many of the GPI functions that are issued to it. Then, at a later time, the presentation space can be instructed to play the retained information out of the presentation space, thus rendering a picture. The normal presentation space is the type of presentation space used by our graphic editor.

Before digging down into the next layer of the system architecture, let's look closer at how the presentation space allows you to define the presentation page. When you create a micro or normal presentation space with the GpiCreatePS function, you also get to specify information about the presentation page you want. Parameter three (size) of the GpiCreatePS function allows you to specify the presentation page size. This size parameter actually defines a rectangle that is the dimension of the presentation page. Again, think of the presentation page as an electronic sheet of paper. This parameter lets you specify the dimensions of that sheet of paper. The units for the size parameter are dependent on what you specify in parameter four (options) of the GpiCreatePS function. The options parameter actually conveys multiple pieces of information about how you want the presentation space created. Some of the items conveyed in the options parameter are if the presentation space is to be normal or micro, if certain values are to be

saved as SHORT or LONG, and if the presentation space is to be associated with a specific device. Six bits of the options parameter are used to define the units for the presentation page. These six bits have definitions available as follows:

PU_PELS	Pel coordinates (picture elements of the physical device)
PU_LOMETRIC	Units of 0.1 mm
PU_HIMETRIC	Units of 0.01 mm
PU_LOENGLISH	Units of 0.01 inches
PU_HIENGLISH	Units of 0.001 inches
PU_TWIPS	Units of 1/440 inches
PU_ARBITRARY	Application-convenient units

As you can see, your application has a fairly broad choice as to what units it wants the presentation page to use. For our graphic editor program, we chose PU_LOENGLISH; hence, when we are dealing with our presentation page, we are using units of 1/100 of an inch.

As implied by the micro and normal presentation spaces, a logical connection between the presentation space and a device can be made. (A cached-micro presentation space is always associated with a display or window.) The object with which the presentation space is associated to make this connection is called a *device context*. A device context is a logical output device that identifies the target for the physical drawing. Therefore, the device context also identifies the physical output device. This output device can be a shared device such as a printer, or a target such as a memory bitmap or OS/2 MetaFile. Your application is responsible for creating a device context and making an association between it and a presentation space. For something as simple as a window device context, a function called *WinOpenWindowDC* can be used to return a window device context. Creating a device context for something other than a window is somewhat more complicated, but the GPI has functions that still make it fairly easy. In fact, these functions not only help making the creation of a device context fairly easy but give the end user a tremendous amount of flexibility in device selection and setup. Basically, these GPI functions allow you to query device information from the system and then interface with pieces of code called *presentation drivers*. These presentation drivers can provide a dialog with the end user and, on completion of that dialog, build a data structure that

your application can use for creating a device context. (This will be shown in detail in Chapter 10, "Printing".) It turns out that these presentation drivers are also the next key layer of the system that help enable device independence.

A presentation driver is responsible for converting the high-level GPI drawing orders into low-level orders that can be routed to the output device. Hence, the presentation driver needs to be aware of the output device's device-specific characteristics. In some cases, the presentation driver actually interfaces directly to the adapter hardware. (This is common for display adapter presentation drivers.) In other cases, the presentation driver may route device-specific data through the OS/2 kernel to yet another component called a *device driver*. The device driver, which in some cases is part of the OS/2 kernel, is the last layer of software used to route data to the physical device.

A high degree of flexibility and device independence can be achieved from this layering approach and from your application having the ability to create and associate presentation spaces to different device contexts. This is not meant to imply that the GPI does not have functions that show device-specific detail or allow device-specific control. In fact, many functions of the GPI allow for the querying of device-specific information and drawing in device units. For many applications (including those designed for device independence) this level of control is important. But, as you study the different GPI functions, it becomes clear how many of these functions allow you to achieve device independence. Our graphic editor is an example of this concept.

As implied earlier, the presentation page is key in providing device independence. The presentation page, for which you define the size and units, represents your electronic sheet of paper and many of the GPI functions represent your electronic pencil. For example, a GPI function called *GpiLine* allows you to draw a line from the current drawing position to a given point. Just supply this function with a point that is to be the end of the line. Another cool thing about the OS/2 graphic environment (illustrated with the GpiLine function) is that this environment tracks the current drawing position. Thus, you don't have to constantly state a starting position for each object you want to draw! If you do want to change the current drawing position, however, a GPI function called *GpiMove* allows you to do this. (Actually, which

GPI function you use and how you use it determines how the current position is managed.)

The obvious question left about drawing with GPI functions is how to specify points or coordinates. The coordinate system for your presentation page places the origin (0,0) at the bottom left corner. The units of this coordinate system for your presentation page are whatever you defined them as when you created the presentation space. Figure 2.2 shows an example of the presentation page coordinate system. In this particular example, the presentation page for our graphic editor program is represented. You may think that once you have defined your presentation page with a certain unit of measure, then you simply supply the different GPI functions with values in those units. That would be a good guess, but for many GPI functions, this is the wrong answer! It turns out that most GPI drawing functions take coordinate information in units called *world coordinates*. World coordinates are abstract units that provide a large number range. Shortly, you will see why this is done, but first let's discuss a very important GPI function that helps us deal with the added complexity of different units.

A function called *GpiConvert* is used to transform units from one coordinate space to units of another coordinate space. So far, we have hinted about two coordinate spaces. These two coordinate spaces are called world and

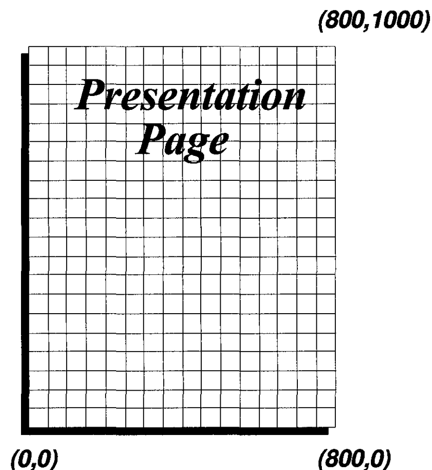


FIGURE 2.2 Graphic editor presentation page definition.

presentation page. To show how we might use the GpiConvert function in our graphics editor program, suppose we want to display a clever message at a given location in our client area. Let's suppose this clever message is "Hello World" and we want to start this message 1.5 inches to the right and 2 inches up. The GPI function we might choose to write our clever message with is GpiCharStringAt. As it turns out, however, the GpiCharStringAt function requires a parameter to specify a starting point for the text string; furthermore, this starting point must be specified in world coordinates. Now, because our presentation page units are `PU_LOENGLISH` (1/100 of an inch), we first want to define our message starting point in presentation page units (150,200). Next, we need to convert these presentation page units to world coordinates by using the GpiConvert function. Following is an example of a GpiConvert statement to do our conversion:

GpiConvert(*hpsEditor*, `CVTC_DEFAULTPAGE`, `CVT_WORLD`, 1, &*startPoint*);

where

hpsEditor is the graphic editor presentation space.
`CVTC_DEFAULTPAGE` defines the source coordinate space as presentation page.
`CVT_WORLD` defines the target coordinate space is world.
1 indicates only 1 point is to be converted.
startPoint is a `POINTL` data type which holds the coordinate to be converted.

Once this GpiConvert function completes, we can use the returned coordinates as an input parameter to the GpiCharStringAt function. As you look at the definition of the GpiConvert function in the IBM technical reference material, you will notice that several other values for the source and target coordinate spaces exist. This leads us into our next architectual topic, which is one of the neatest parts of the GPI.

THE VIEWING PIPELINE AND PICTURE CONSTRUCTION

It turns out that to produce a drawing with the OS/2 GPI, multiple transformations on the different graphics objects are done before they are output to the physical device. These transformations occur between a series of coordi-

nate spaces which exist in this graphics environment and are the means by which objects are transferred from space to space. This series of coordinate spaces and transformations is called the *viewing pipeline*. Besides the transformations between coordinate spaces, each coordinate space can have an area defined within it where *clipping* will occur. Any object or part of an object that is located outside of this area and is to be clipped will not be transferred to the next coordinate space.

As implied by the previous section, one reason for these transformations is to help you produce device-independent software. However, as you'll soon see, the viewing pipeline adds a lot more function to this graphics environment than just device independence. In Chapter 6, "Transformations," we will discuss transformations and how they are produced in detail. But here, we will be looking at the viewing pipeline (and transformations) at a high level. Hence, to understand why the viewing pipeline adds so much to the OS/2 graphical environment, let's look at each of the basic coordinate spaces in the pipline. As you read about the different coordinate spaces, refer to Figure 2.3 so you can better see how these spaces relate to one another.

The first coordinate space in the viewing pipeline is called *world space*. Each graphical object generated using the GPI is first defined in this space. The units used in this space are abstract in value and have a very large range (up to 28 bits). Therefore, the different graphics objects can be defined with great precesion independent from its physical size. For example, suppose you want to create an object that represents a rivet that is used as part of a bridge and another object that represents an I-beam for the body of the bridge. The units in the world space give you the flexibility to define each of the objects with adequate precision without regard to the definition of the other object. As these objects are pieced together in a later coordinate space, their relative sizes and location will be accounted for via transformations. In fact, because the rivet will be so small in comparison to the body of the bridge, it probably won't been seen unless a zoom function is applied to part of the picture.

The clip area that can be defined in the world space is called a *clip path*. Usually, clip areas are thought of as rectangular in shape. What is unique about clipping in world coordinate space, however, is that the clip area can be defined as an irregularly shaped object. That is to say that the shape of the clip path can be defined with a series of curves. The clip path can also be rotated. During picture constructions, you may define several different clip

paths. There are several GPI functions that allow you to define and work with the clip path. Information about these GPI functions is given in Chapter 7, "Paths, Regions, Clipping, Boundary Accumulation, and Correlation."

The second coordinate space in the viewing pipeline is called *model space*. Model space is where different graphic objects from world space can be manipulated and pieced together to build higher level objects. This is done by performing transformations on the objects from world space so that their size, position, and orientation can be changed as they enter model space. Note that an object from world space can be used more than once in model space. Therefore, an object that is used several times in a drawing can be defined once in world space and then used muliple times in model space. Of course, each time the object is used in model space from world space, a different transformation may be applied to it. For example, suppose you have a drawing of a calculator which has several keys of the same size or shape. All you really need to do is create one object in world space that represents the key object and then use that key object multiple times. Each time the key object enters model space, apply a different transform to it so the key is drawn in the correct relationship to the body of the calculator. An application can have more than one model space, but most applications only have one.

The clip area that can be defined in model space is called the *viewing limit*. The viewing limit is always rectanglar in shape and can also be used several times during picture construction. Rotating the viewing limit does not change the orientation of the rectangular area used for clipping but does change the size of this rectangular area.

The third coordinate space in the viewing pipeline is called *page space*. Page space is where the entire picture for output is constructed. This is done by applying transformations to the different model spaces so the objects in these spaces are pieced together with the correct size, location, and orientation with respect to each other. The page space is also where the picture's physical units of measure are determined. Remember, when your application creates a presentation space, it also defines the units for this coordinate space and the size of the presentation page. Note, however, that these units can be arbitrary if you want to create your own units of measure. Of course, you may also have to control the transformation between the presentation page space and the device space to get the desired size for the final physical output. If you don't control this last transformation, the GPI will provide a

default transformation that will make your presentation page size fill the device space while still maintaining the aspect ratio.

The clip area that can be defined in page space is called the *graphics field* and is always rectangular in shape. Like all the other clip areas, any part of the drawing found outside of the graphics field will be clipped (not shown) in the next coordinate space. If you use the graphics field, you must define it before picture construction and leave it alone. (The graphics field can not be changed during picture construction.) Also, the graphics field can not be rotated.

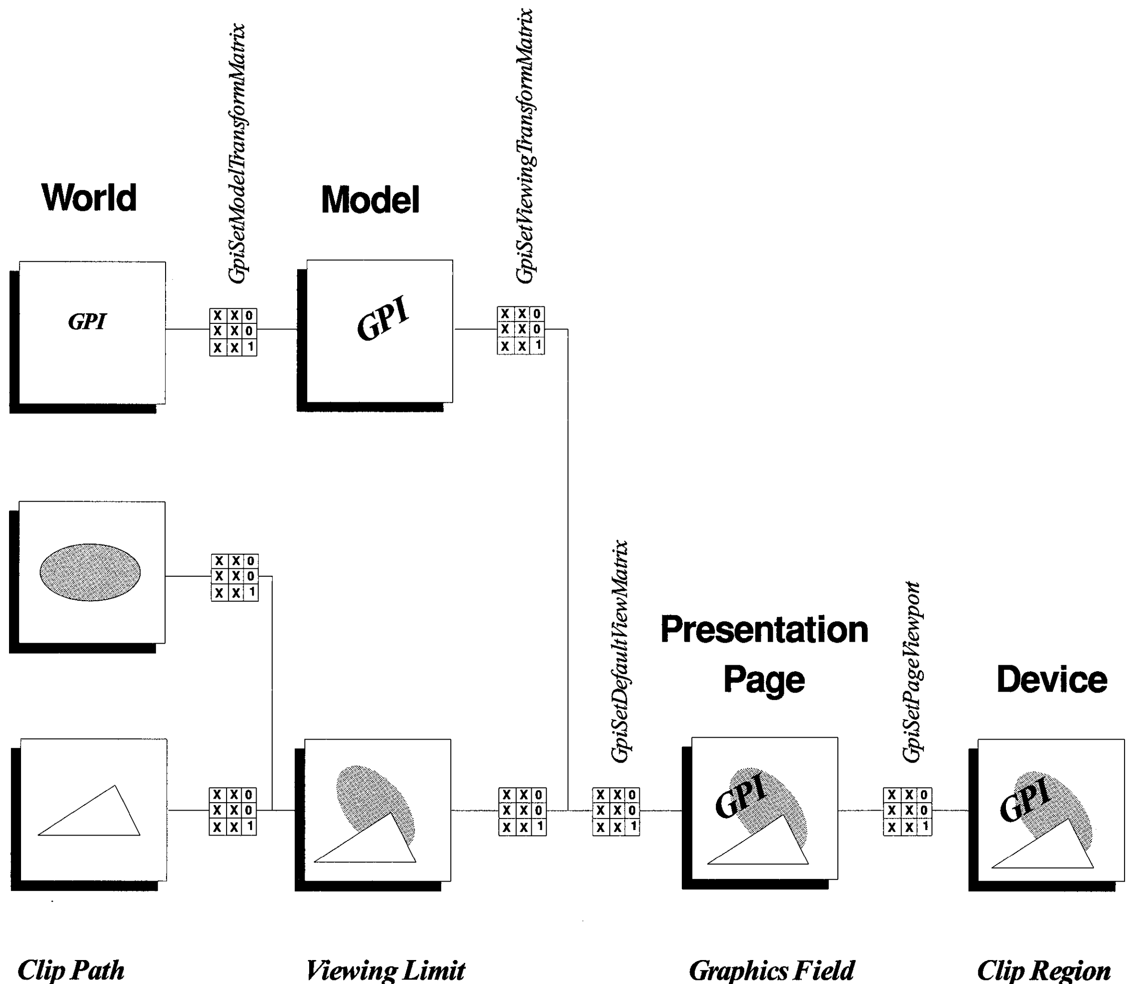


FIGURE 2.3 Viewing pipeline.

The fourth and last coordinate space that we are going to discuss that is part of the viewing pipeline is called *device space*. Device space is where the picture units are finally converted to device-specific units. After the picture has been constructed in this space it can be routed to the output device.

The clip area that can be defined in device space is called a *clip region*. The clip region is defined by a set of rectangles specified in device units. Any drawing requested outside of any of these rectangles is clipped. Hence, two nonintersecting rectangles that define a clip region would appear to define two different clip areas. However, if the rectangles that define the clip region intersect, the resulting clip area will appear as intersecting rectangles, not just a bigger rectangle. The clip region can be changed during picture construction but can not be rotated.

Now that we've discussed each of the different coordinate spaces, let's look at a hypothetical drawing so we can see how the viewing pipeline helps with picture construction. As we discuss this hypothetical drawing, refer to Figure 2.4 to better understand how picture construction works. The discussion of our drawing will focus on the type of things that can occur in the viewing pipeline by using the GPI rather than the details of how this might be implemented.

For our example, let's suppose we want to construct a very simple drawing of some nuts and bolts holding a couple of plates together. For this drawing, we really only need to define a few objects. These objects are a nut, a bolt, a spacer, and a plate. Each of these objects can be defined in world space independent of the other objects. Then, as the objects enter model space, transformations are applied to them so that they are sized and oriented as required for the drawing. Note that one instance of an object in world space can be used multiple times in model space. You can see in our example (illustrated in Figure 2.4) that we have multiple nuts, bolts, spacers, and plates defined in model space and their size, location, and orientation have been adjusted. As we go from model space to page space, we can join different model spaces together. Like going from world space to model space, another transformation is used to join the objects defined in the different model spaces. These different model spaces may be considered subpictures; however, our example and most other applications do not join multiple model spaces. After the different model spaces have tranformations applied to them, but before the resulting picture actually enters page space, yet another transformation occurs. This tranformation is used to scale and trans-

late your picture (*zoom* and *pan*). This scaling and translation in page space is shown in Figure 2.4 by showing the picture as being larger and offset in this space. Once the object is in page space, the units should be whatever was defined for the presentation page when the presentation space was created. Now, to get the object to print or display correctly on the desired output device, yet one last transformation must occur. This last transformation converts the units defined in page space to map to the physical device units. Hence, the transformation from page space to device space provides this final function. In Figure 2.4 this is illustrated by showing the device space as a printer. (Of course the actual transformation for device space is done in OS/2 and not by the physical device.)

In addition to the basic coordinate spaces of the viewing pipeline, a couple more coordinate spaces still exist. A coordinate space called the *notional font definition space* exists before world space and is used to define a special kind of font called an *outline font*. Outline fonts are discussed in Chapter 4, "Fonts." The last coordinate space that exists is called the *media space* and is after the device space. The media space is used to produce windows on a

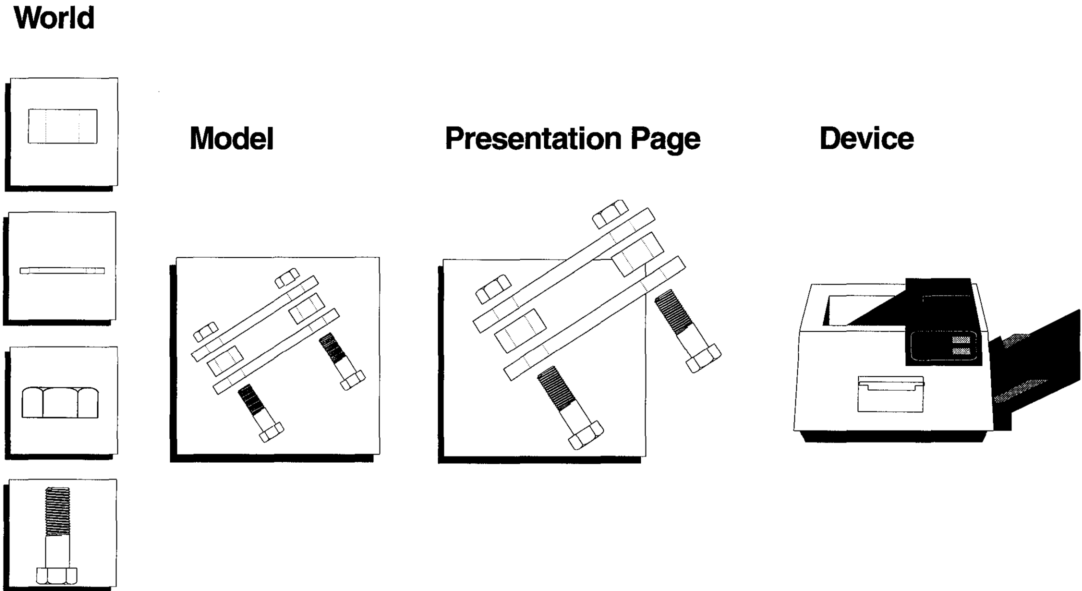


FIGURE 2.4 Example drawing in viewing pipeline.

device such as a terminal. Because the terminal is a device and the device space applies to the entire terminal, we need the media space to transform our drawings into areas of the device we call windows!

A LITTLE ABOUT MULTITASKING

If one of my kids were to become interested in computer programming, I can imagine how some of our conversations would sound. "Son, when I started programming PCs in the old days, 64KB of memory was considered a lot of memory and we were glad to have it!" Actually, when I started programming my first home–built computer, 16KB was considered a lot of memory and paying $450.00 for this memory was cheap. (For historical interest, the year was 1978!) And the way most of us used our PCs then was quite different from now. We typically ran a single non-graphical application at a time. Most of these applications seem pretty lame by today's standards, but back then they were considered pretty slick! Of course, it wasn't too long until some clever programmers were making their printers print while they were using some other application on their PC. Or they would start up their own special popup programs that were already loaded in the PC with some special keyboard sequences. Back then, I thought this was really neat and wanted to know how I could write programs to do this too!

"Well son, in the old days, real programmers took control the PC! We stole interrupts and grabbed storage when we needed it . We took over devices and programmed them to meet our specific application needs. We did all of this while trying not to trash other applications that might also be running in the PC and trying to do the same things! And by the way son, even though there were some great efforts to make this work, it never really worked that well at all."

Actually, due to the lack of good standardized rules (architecture) for sharing resources in the PC, it was amazing things worked as well as they did. And if you think about how or where resources should be managed so multiple applications can run at a time, you will probably quickly conclude that the operating system needs to be a key part of the solution. And indeed OS/2 is!

Today it isn't so difficult to imagine running multiple applications on your PC at the same time. But why would an application need multiple threads of execution? Well, perhaps many simple applications only need a

single thread of execution, but many programs can become more functional if they can process data or wait for some events in a background thread while doing other activities in a foreground thread.

From your previous Presentation Manager programming experience, you should already be aware of the case where you should process some Presentation Manager messages in a background thread. This occurs when you receive a message in your message queue that requires considerable processing time. When this happens, you should process the message in a background thread so you can be responsive processing other messages that may still be arriving in your message queue. If you don't do this, your application may appear slugglish because some of these delayed messages could be mouse messages, change focus messages, keyboard messages, and so on.

The Browse utility's (`BROWSE.C`) `WM_PAINT` message is an example of a message that could take a long time to process. Because of this, the `WM_PAINT` message for the browser interacts with a background thread in order to draw text into the client area. Listing 2.1 shows the `WM_PAINT` message for the Browse utility while Listing 2.2 shows the background thread that draws text on the screen for the Browse utility. As you can see from Listings 2.1 and 2.2, the `WM_PAINT` message and the background thread communicate with global variables and semaphores. Each time the `WM_PAINT` message needs to refresh the window, it sets a global flag named filling to FALSE to get the background thread's attention and then waits on a semaphore that is cleared by the background thread so the `WM_PAINT` message can proceed. Because the background thread checks this global flag after each line it draws, it doesn't take very long for it to recognize it should stop drawing. Once the background thread stops drawing, it unblocks the semaphore the `WM_PAINT` message is waiting for and then waits for a semaphore that the `WM_PAINT` message unblocks when it wants the background thread to start drawing again. Once the `WM_PAINT` message is allowed to run, it sets up global variables needed by the background thread to start drawing again and then unblocks the background thread's semaphore so it can draw the text independent from other browse activities. Hence, the `WM_PAINT` and background threads are interlocked and sequenced with semaphores and global variables. The result is a much more responsive Browse utility. This is particulary noticeable when a user selects a font that takes a long time to draw. If you pick a font that is fairly slow to draw and page up or down quickly three or four times, you will notice that the window

is not fully updated before that page output is stopped and the next page is started!

```
/*************************/
/* Process paint message. */
/*************************/
case WM_PAINT:
   hpsBrowse=WinBeginPaint(hwnd,hpsBrowse,&rectPaint);
   filling=FALSE;
   DosWaitEventSem(donePainting,SEM_INDEFINITE_WAIT);
   DosResetEventSem(donePainting,&doneEventCount);
   WinFillRect(hpsBrowse,&rectPaint,CLR_BACKGROUND);
   gptPaint.x=ptTopLeft.x;
   gptPaint.y=ptTopLeft.y;
   /****************************************************/
   /* Get the first line to display and the count of them  */
   /* to display.                                          */
   /****************************************************/
   lineOut=curYPos;
   gCount=((lineCount-curYPos)<rows) ? (lineCount-curYPos) : rows;
   /******************************************************/
   /* Get the minimum and maximum file offset values needed. */
   /******************************************************/
   minOffset=*((PULONG)fileOffsets+(lineOut));
    if((lineOut+count+1)<lineCount)
      maxOffset=*((PULONG)fileOffsets+(lineOut+count+1))
   ; else
      maxOffset=*((PULONG)fileOffsets+(lineCount-1));
   /*********************************************/
   /* If needed, read in new chunk of input file. */
   /*********************************************/
   if(minOffset<baseOffset || maxOffset>(baseOffset+FILE_BUF_SIZE)){
      if(minOffset<(FILE_BUF_SIZE/2))
         baseOffset=0;
      else
         baseOffset=minOffset-(FILE_BUF_SIZE/2);
      DosSetFilePtr(fileHandle,baseOffset,FILE_BEGIN,&local);
      DosRead(fileHandle,bufferPtr,FILE_BUF_SIZE,&bytesRead);
   }
   /*********************************************************/
   /* Put cursor up here if it can be found within the window. */
   /*********************************************************/
   gOffsetIndex=*((PULONG)fileOffsets+(cursorY));
   gTemp=bufferPtr+gOffsetIndex-baseOffset;
   SetCursor(hwnd,gTemp);
   filling=TRUE;
   DosPostEventSem(startPainting);
   WinEndPaint(hpsBrowse);
   return 0;
```

LISTING 2.1 Browse utility paint message processing.

```
/*********************************************/
/* Background thread for drawing browse text. */
/*********************************************/
VOID backGroundPaint(ULONG dummy){
   LONG i;
   SHORT x,y,toggle;
   HAB habt;
   /***************************************************************/
   /* Get an anchor block handle so thread can access PM functions. */
   /***************************************************************/
    habt=WinInitialize(0);
   for(;;){
      DosPostEventSem(donePainting);
      /*********************************************/
      /* Wait for main process to clear semaphore. */
      /*********************************************/
      DosWaitEventSem(startPainting,SEM_INDEFINITE_WAIT);
      /********************************/
      /* Display full window of data. */
      /********************************/
      while((gCount--) && (filling)){
         gOffsetIndex=*((PULONG)fileOffsets+(lineOut));
          gTemp=bufferPtr+gOffsetIndex-baseOffset;
         gstr=stringout;
         *gstr='\0';
         /***************************************************/
         /* Replace tab characters with up to eight spaces. */
         /***************************************************/
         for(gLength=0; (*gTemp!='\n') && (gLength<250 ); *gTemp++){
         if(*gTemp=='\t'){
         gblkCnt=8-(gLength%8);
         gLength=gLength+gblkCnt;
         while(gblkCnt>0){
            *gstr++=' ';
            gblkCnt--;
         }
      }
      else *gstr++ =*gTemp;
      gLength++;
   }
   /***************************/
   /* Remove carriage returns. */
   /***************************/
   if(*(gstr-1)==13)gLength--;
   if(gLength>curXPos){
      hIndex=curXPos;
      gLength-=curXPos;
   }
   else {
      hIndex=gLength;
      gLength=0;
   }
```

LISTING 2.2 Browse utility background paint processing.

```
      /*************************************************************/
      /* Output (partial) line to window and update loop variables. */
      /*************************************************************/
      SetWidthsTable(hpsBrowse,widthValues,
         widthTable,&stringout[hIndex],&gstrWidth);
      GpiMove(hpsBrowse,&gptPaint);
      GpiCharStringPos(hpsBrowse,NULL,CHS_VECTOR,
         (LONG)gLength,&stringout[hIndex],widthValues);
      gptPaint.y-=yChar;
      // Move down 1 row lineOut++;
      // Increment to line offset.
   }
   if(filling){
      /******************************/
      /* Output line count position. */
      /******************************/
      stringout[0]=0;
      strcat(stringout,"Line ");
      CvtInt((cursorY+1),&stringout[strlen(stringout)]);
      strcat(stringout," of ");
      CvtInt((lineCount),&stringout[strlen(stringout)]);
      strcat(stringout,",  Column ");
      CvtInt((cursorX+1),&stringout[strlen(stringout)]);
      strcat(stringout,".  ");
      strcat(stringout,pBrws->filename);
      GpiSetColor(hpsBrowse,CLR_RED);
      SetWidthsTable(hpsBrowse,widthValues,
         widthTable,&stringout[0],&gstrWidth);
      gptPaint.y=0;
      GpiMove(hpsBrowse,&gptPaint);
      GpiCharStringPos(hpsBrowse,NULL,CHS_VECTOR,
         (LONG)strlen(&stringout[0]),&stringout[0],widthValues);
      }
      GpiSetColor(hpsBrowse,CLR_BLACK);
      DosResetEventSem(startPainting,&startEventCount);
      filling=FALSE;
      }
   }
```

LISTING 2.2 (Continued).

Another good example of when your application may find multiple threads of execution useful is when it is waiting for data from another source. What makes this an even better example is when your application doesn't even know when or if data may arrive, and perhaps when the data does arrive, it can be pre-processed before it is passed to your application's foreground thread. This example is not really very farfetched. This is a common requirement for applications that depend on receiving data from some communications support. Conversely, the same application may want to pass data back to a communications link in a background thread while processing more interesting user-related activity in a foreground thread. It may be the

case, however, that data is not really being sent or received from a communications link, but rather to or from another application in the same machine. From your applications point of view, perhaps it shouldn't matter where the data is coming from. What is key is that much of the data processing can be managed outside your application's main thread of execution. Furthermore, your design may be such that the data is formatted and encapsulated in such a way that makes the data access consistent and safe. (This is a major theme of object oriented design and programming!)

In our graphic editor program, we show one more way that multitasking may be used. When our graphic editor wants to display the selected graphic objects order stream, it generates an ASCII file with the data for viewing. After the ASCII file is generated, the graphic editor disables the View Selected Objects pull-down menu item and uses the Browse utility to display the content of the ASCII file. Note that one of the parameters passed to the Browse utility when it is invoked is the window handle of the invoking process. Because the Browse utility runs in its own thread of execution, the graphic editor resumes operation shortly after the Browse utility is invoked. Hence, while the user is browsing the ASCII file that was generated by the graphic editor, the graphic editor can proceed independently from the Browse utility. When the user quits the Browse utility, the utility sends a user-defined message back to the Presentation Manager application that started it via the window handle. This message is processed in the same fashion as other messages and is an indicator to the invoking application that the user is done browsing the graphic objects ASCII file. Hence, it is the user-defined message sent from the Browse utility that re-enables the View Selected Objects pull-down menu item!

CHAPTER 3
Graphic Primitives

Most applications make use of a set of basic building blocks for graphics construction. These building blocks consist of lines, curves, rectangles, text, and images. The OS/2 GPI provides these capabilities and more in what it calls *graphic primitives.*

The set of OS/2 graphic primitives is fairly large and quite functional. Using these primitives, one can construct both simple and complicated figures. This chapter centers around the commonly used graphic primitives.

ATTRIBUTES

Before jumping into the actual drawing commands, let's first discuss the topic of drawing attributes. Attributes deal with how figures and text appear when they are drawn. Text color, font style, and point size are examples of text attributes.

The GPI breaks drawing primitives into five categories. Each of these categories has a set of attributes that are specific to the drawing primitives in that category. The GPI refers to this collection of attributes as a *bundle*. By setting the values of attributes in a bundle, you can affect how drawing primitives that use that bundle will look when they are drawn to the device. For example, by setting the *color* attribute in the *line bundle* to blue, any

subsequent lines that you draw will be displayed in blue. However, drawing primitives in other categories will not be affected since they use their own bundle. For example, text primitives have their own attribute bundle which also has a color attribute. Therefore, text primitives will not be affected by a change to the color attribute in the line bundle.

The five categories of the drawing primitives are:

1. Line and Arc primitives
2. Area primitives
3. Text primitives
4. Image primitives
5. Marker primitives

As previously mentioned, each of these categories has its own bundle of attributes. Table 3.1 shows the attributes in each bundle.

You can set one or more attributes of each bundle by using a function called *GpiSetAttr*. For example, let's say you want to set the line color and line type attributes in the line attribute bundle. To do this fill we tell the GPI what attributes are being set and what the new values are. This is done with a bundle attribute record and a call to the GpiSetAttr function. Listing 3.1 shows a quick example of how this function is used.

```
LINEBUNDLE lineAttr;
/* Set line attributes */
lineAttr.lColor = CLR_BLUE;
lineAttr.usType = LINETYPE_DOT;
GpiSetAttrs(hps, PRIM_LINE, LBB_COLOR | LBB_TYPE, 0L, &lineAttr);
```

LISTING 3.1 Using GpiSetAttr.

Sometimes you may want to determine an attribute's current value. You can do this with a function called *GpiQueryAttr*. This function works almost exactly like the GpiSetAttr function only in reverse. You need to indicate what attributes you are interested in and then the GpiQueryAttr function will fill in the attribute record with the values of the attributes you asked for. Simple enough. As you might expect, GpiQueryAttr lets you ask for one or more attribute values as long as they are in the same bundle (just like GpiSetAttr). Listing 3.2 shows how you can use this function.

TABLE 3.1 Attribute bundles

Bundle Name	**Attributes**
Lines and Arcs (LINEBUNDLE)	Line Color, Line Mix, Line Width, Geometric Line Width, Line Type, Line End, Line Join
Areas (AREABUNDLE)	Area Color, Area Background Color, Area Mix, Area Background Mix, Pattern Set, Pattern Symbol, Pattern Reference Point
Text (CHARBUNDLE)	Character Color, Character Background Color, Character Mix, Character Background Mix, Character Set, Character Mode, Character Box, Character Angle, Character Shear, Character Direction, Character Text Alignment, Character Extra, Character Break Extra
Image (IMAGEBUNDLE)	Image Color, Image Background Color, Image Mix, Image Background Mix
Marker (MARKERBUNDLE)	Marker Color, Marker Background Color, Marker Mix, Marker Background Mix, Marker Set, Marker Symbol, Marker Box

```
LINEBUNDLE lineAttr;
/* Get line attributes and write them to a log file*/
GpiQueryAttrs(hps, PRIM_LINE, LBB_COLOR | LBB_TYPE, 0L, &lineAttr);
fprintf(log, "Line color %ld, Line type=%ld\n", lineAttr.lColor,
lineAttr.usType);
```

LISTING 3.2 Using GpiQueryAttr.

Although the GpiSetAttr is flexible and can be used to set most drawing primitive attributes, the GPI has included some shortcut or helper functions for some of the more frequently set drawing attributes. For example, the following functions are used to simplify setting of the basic line attributes:

- GpiSetLineEnd
- GpiSetLineJoin
- GpiSetLineType
- GpiSetLineWidth
- GpiSetLineWidthGeom

These functions are equivalent to using GpiSetAttr, but they are easier to use. Those attributes with helper functions for setting their values usually also have helper functions for querying their values. For example, the function called *GpiQueryLineEnd* returns the current value of the *Line End* attribute.

Besides helper functions that operate on attributes within a bundle, there are also helper functions that operate on attributes across all of the attribute bundles. When these functions are called, they set the value of the specified attribute in each attribute bundle. The function called *GpiSetColor,* for example, will set the color attribute in each drawing category (i.e., Lines, Areas, Characters, Text, Markers, and Images). All subsequent drawing operations will have the same foreground color. The common helper functions are:

- GpiSetColor
- GpiSetBackColor
- GpiSetMix
- GpiSetBackMix

Of course, there are matching query functions for each of these helper functions. Note, however, that the query functions only return the current value of the attribute found in the *character* bundle. The values of that attribute found in the other bundle are ignored.

COLOR AND MIX

As previously mentioned, the color and mix attributes exist in each attribute bundle. For that reason we will briefly describe them here and avoid repeating discussion of them in the following sections on each drawing category.

Color attributes are specified as in an index with values zero through *n* where *n* is the number of entries in the color index table. Table 3.2 lists the index values for the default color.

The color index table can change, however, so bear in mind that picking CLR_BLUE might not result in the color blue if the color table has been changed. If only a few different colors are needed, choose the colors CLR_DEFAULT, CLR_BACKGROUND, and CLR_NEUTRAL, so your application can attempt to remain device independent. Although in practice this is not very realistic as usually more than three colors are required.

TABLE 3.2 Default color table values

Index	Value	Description
0	CLR_FALSE	Monochrome (all bits off, black)
1	CLR_TRUE	Monochrome (all bits on, white)
2	CLR_DEFAULT	System defined foreground color
3	CLR_WHITE	White
4	CLR_BLACK	Black
5	CLR_BACKGROUND	System defined background color
6	CLR_BLUE	Blue
7	CLR_RED	Red
8	CLR_PINK	Pink
9	CLR_GREEN	Green
10	CLR_CYAN	Cyan
11	CLR_YELLOW	Yellow
12	CLR_NEUTRAL	System defined neutral color
13	CLR_DARKGRAY	Dark gray
14	CLR_DARKBLUE	Dark blue
15	CLR_DARKRED	Dark red
16	CLR_DARKPINK	Dark pink
17	CLR_DARKGREEN	Dark green
18	CLR_DARKCYAN	Dark cyan
19	CLR_BROWN	Brown
20	CLR_PALEGRAY	Pale gray

TABLE 3.3 Foreground mix attributes

Attribute	Description
FM_DEFAULT	Default foreground mix mode (usually FM_OVERPAINT unless changed).
FM_OR	Logical OR foreground color with current screen contents.
FM_OVERPAINT	Paint foreground color over the current screen contents.
FM_XOR	Exclusive OR foreground color with current screen contents.
FM_LEAVEALONE	Leave the current screen contents alone (draw invisible).
FM_AND	Logical AND foreground color with current screen contents.
FM_SUBTRACT	Invert the foreground color and logical AND with current screen contents.
FM_MASKSRCNOT	Logical OR foreground color with the inverse of the current screen contents.
FM_ZERO	Resulting value is to be set to zero.
FM_NOTMERGESRC	Invert the logical OR of the foreground color and the current screen contents.
FM_NOTXORSRC	Invert the exclusive OR of the foreground color and the current screen contents.
FM_INVERT	Invert the current screen contents.
FM_MERGESRCNOT	Logical OR foreground color with the inverse of the current screen contents.
FM_NOTCOPYSRC	Invert the foreground color and paint result over current screen contents.
FM_NOTMASKSRC	Invert the logical AND of the foreground color and the current screen contents.
FM_ONE	Resulting value is to be set to one.

Drawing primitives often have both a foreground and a background color. The foreground color specifies in which color the front of the object is to be drawn. For example, the foreground color of text is the color in which each character will be drawn. Background color, as implied by the name, specifies in what color the background of the object will be drawn. For text, the background color specifies the color with which the character box will be filled (the character will be drawn on top of this filled in box). Note that Line and Arc primitives do not have a background color.

Mix attributes control how a figure is combined with the rest of the picture when it is drawn. As with color, there are both foreground mix and background mix attributes. These attributes control how a figure's foreground and background colors of a drawing primitive are combined with the existing drawing. The mix attributes specify bit-wise operations in which each pixel of the primitive to be drawn is combined with the current contents of the pixel over which it is to be drawn. New and current pixel values are numeric. The numeric values represent an index into the current color table. Therefore, by combining the new and existing pixel values, a resulting value is produced to be used as the new color table index for that pixel.

Tables 3.3 and 3.4 list the foreground and background mix attributes (values) that are available.

TABLE 3.4 Background mix attributes

Attribute	Description
BM_DEFAULT	Default background mix mode.
BM_OR	Logical OR background color with current screen contents.
BM_OVERPAINT	Paint background color over the current screen contents.
BM_XOR	Exclusive OR background color over the current screen contents.
BM_LEAVEALONE	Leave the current screen contents alone (draw background invisible).

TABLE 3.5 Line and arc GPI functions

Function	Description
GpiLine	Draw a single line.
GpiPolyLine	Draw a series of connected lines.
GpiPolyLineDisjoint	Draw a series of disjoint lines.
GpiBox	Draw a rectangle.
GpiFullArc	Draw a circle or ellipse.
GpiPartialArc	Draw an arc along an ellipse.
GpiPointArc	Draw an Arc from three points.
GpiSpline	Draw a series of Bezier splines.
GpiPolyFillet	Draw a series of fillets.
GpiPolyFilletSharp	Draw a series of fillets with specified sharpness values.

LINE AND ARC PRIMITIVES

This category of drawing operations includes basic line and arc drawing functions. Table 3.5 lists the functions in this category and a brief description of the operations they perform.

As mentioned above, there is an attribute bundle that is specific for this category of drawing primitives. The setting of attributes in this bundle will affect how figures drawn using these operations are displayed. Table 3.6 describes each attribute in this bundle.

The *Line Color* attribute defines what color is used to draw any of the line or arc primitives.

The *Line Mix* attribute defines how the line color is mixed with the color of other figures on the drawing surface. If you want a solid line of the specifed line color, use `FM_OVERPAINT`. If a line mix of `FM_XOR` is used, the line will be drawn by logically XOR the line's color values with the color values of the figures beneath the line. The interesting thing about this mode is that if you draw the line twice it will disappear and the figure will look as it did before the line was first drawn. This is the technique we use in the graphic editor to produce the rubberbanding effect when objects are drawn or moved.

TABLE 3.6 Line Attribute Bundle

Attribute Description	LINEBUNDLE Field Name	flAttrMask Value	Default Value	Helper Function
Line color	lColor	LBB_COLOR	CLR_NEUTRAL	GpiSetColor*
Line mix	usMixMode	LBB_MIX_MODE	FM_OVERPAINT	GpiSetMix*
Line width.	fxWidth	LBB_WIDTH	1.0	GpiSetLineWidth GpiQueryLineWidth
Geometric line width	lGeomWidth	LBB_GEOM_WIDTH	None	GpiSetLineGeomWidth GpiQueryLineGeomWidth
Line type	usType	LBB_TYPE	LINETYPE_SOLID	GpiSetLineType GpiQueryLineType
Line end	usEnd	LBB_END	LINEEND_FLAT	GpiSetLineEnd GpiQueryLineEnd
Line join	usJoin	LBB_JOIN	LINEJOIN_BEVEL	GpiSetLineJoin GpiQueryLineJoin

*** These helper functions set the color and mix attributes in all bundles.**

You can draw lines with the Line and Arc primitives in two different modes. Cosmetic lines are virtual lines that have no thickness associated with them. Cosmetic lines are usually drawn one pel wide on the display device. The *Line Width* attribute lets you specify whether the line is to be drawn a single pel wide (LINEWIDTH_NORMAL) or two pels wide (LINEWIDTH_THICK).

Geometric lines have a measurable width that you can change. The *Geometric Line Width* attribute specifies in world coordinates the thickness of the line. Geometric lines must be drawn using paths. For details on using paths to draw geometric lines see Listing 3.18 in the Area Primitives section.

The *Line Type* attribute defines the style in which the line is drawn. Figure 3.1 shows what line type choices are available and what effect they produce.

The *Line End* attribute defines how the ends of geometric lines are formed. Figure 3.2 shows what line end choices are available and what effect they produce.

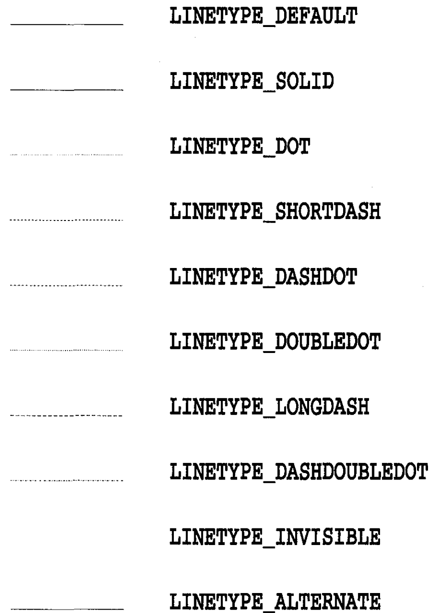


FIGURE 3.1 Line type options.

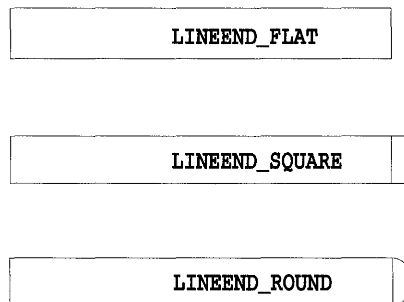


FIGURE 3.2 Line end options.

The *Line Join* attribute defines how geometric lines are joined at their endpoints. Figure 3.3 shows what line join choices are available and what effect they produce.

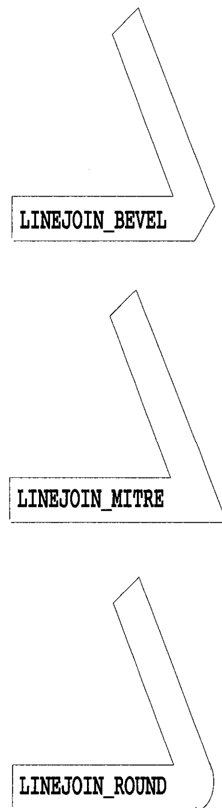


FIGURE 3.3 Line join options.

Note: Although lBackColor and usBackMixMode fields are found in the `LINEBUNDLE` structure, they are not used when drawing line and arc primitives.

Line Functions

The simplest drawing primitive to understand and use is probably the line. A line is simply a connection between two points in the world drawing coordinate space. Using the GPI, a line is drawn from the current position of the graphics pen to another designated point. This operation is performed by a call to the *GpiLine* function as shown in Listing 3.3.

```
HPS hps;
POINTL point;
...
point.x = 100;
point.y = 100;
GpiMove(hps, &point);
point.x = 200;
point.y = 200;
GpiLine(hps, &point);
```

LISTING 3.3 Using GpiLine.

This fragment of code draws a line from point 100,100 to point 200,200. The line will be drawn using the current line color, mix, width, and type attributes of its bundle. After this operation has been performed, the current position is automatically adjusted to be at the new end point of the line(i.e. 200,200). Note, however, that not all GPI functions automatically change the current position. Figure 3.4 shows the output generated by this code fragment.

In fact, as you can see in Listing 3.4, GpiLine is the very call used by the line tool of the graphic editor. To explore this code, look at the GoLineDraw-Details( ) function in the `OBJECT.C` module.

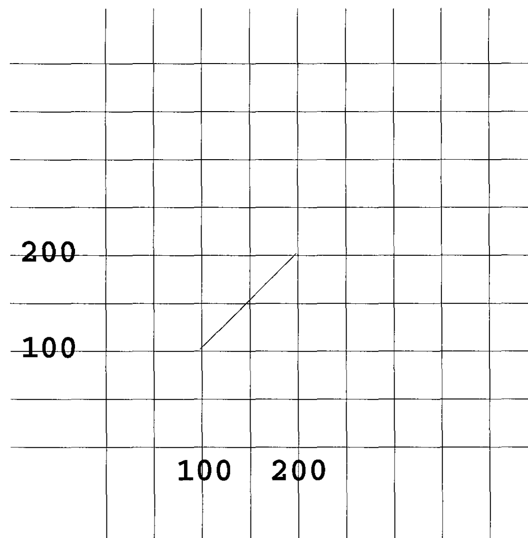


FIGURE 3.4 GpiLine output.

```
void GoLineDrawDetails(GOBJ self)
{
    GpiMove(hps, &((*self->Points)[0]));
    /* Move to first point  */
    GpiLine(hps, &((*self->Points)[1]));
    /* Draw to second point */
}
```

LISTING 3.4 GpiLine used in the graphic editor.

Note that in the graphic editor, endpoints of the line are stored in an array pointed to by the variable *self–>Points*. To draw the line, move to the first point and draw a line to the second point (elements zero and one of the array).

Using lines you can construct quite complicated drawings. Since the current position is automatically adjusted to the new end point of the line, by issuing many line calls in succession you can draw a long sequence of connected line segments. Another way to get this effect, however, is to use the function called *GpiPolyLine*. Listing 3.5 shows how to draw a series of connected lines using this function.

```
HPS hps;
POINTL curPosition
POINTL points[4] = {  0,   100,
                     50,     0,
                     100,  100,
                     100,    0  };
...
curPosition.x = 0;
curPosition.y = 0;
GpiMove(hps, &curPosition);
GpiPolyLine(hps, 4L, &points);
```

LISTING 3.5 Using GpiPolyLine.

This function accepts an array of points that are to be connected (starting from the current position) and a count that tells how many points are in the array. As with GpiLine, the current position is set to the last point after the call. The code fragment above generates a figure that looks like the capital letter M (as shown in Figure 3.5).

As you probably guessed, GpiPolyLine is used to implement the PolyLine tool of the graphic editor. The PolyLine code fragment shown in Listing 3.6 is very similar to that above. As with the line object, the points of the polyline are stored in the array pointed to by self–>Points. The number of points in the polyline is stored in *self–>PointCount*.

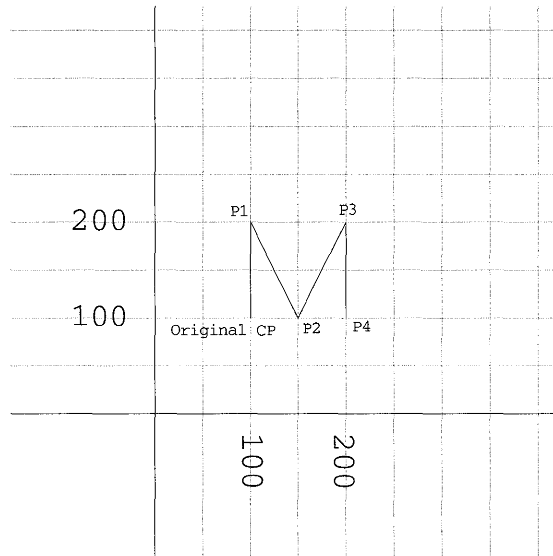


FIGURE 3.5 GpiPolyLine output.

```
void GoPolylineDrawDetails(GOBJ self)
{
    GpiMove(hps, &((*self->Points)[0]));
    GpiPolyLine(hps, self->PointCount-1, &((*self->Points)[1]));
}
```

LISTING 3.6 GpiPolyLine as used in the graphic editor.

The GpiPolyLine function is also used to implement the pencil tool in the graphic editor. Although the pencil tool seems to generate curved lines, it is actually a series of points connected using a polyline. Since the points are so close together, it looks as though curves are being drawn. Try turning on the *snap to grid* feature of the graphic editor and you'll see that the pencil really does draw polylines.

The graphic editor code that uses this function can be found in the GoPencilDrawDetails( ) function in the `OBJECT.C` module. It is identical to the GoPolylineDrawDetails() function previously shown. Later you'll see how the GpiPolyLine is also used to implement the filled polygon tool.

Although not used by our editor, the GPI also provides a function called *GpiPolyLineDisjoint*. This function draws a series of straight lines whose endpoints are passed in as an array of endpoint pairs. Thus, to call it you merely pass in an array of line segment endpoints and the number of points.

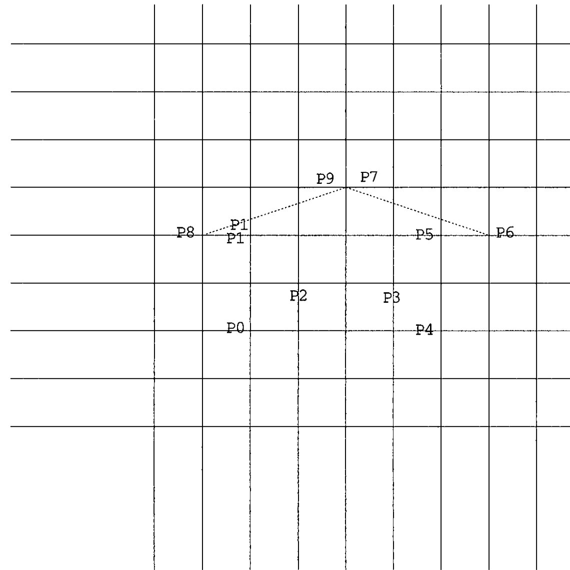


FIGURE 3.6 GpiPolyLineDisjoint output.

This function will then connect points as a series of line segments (i.e. point0–point1, point2–point3, point4–point5, and so on). Figure 3.6 shows how the points passed into this function are used to draw the disjointed polylines.

Boxes and rectangles are actually drawn using the same function. This function is called *GpiBox*. A call to GpiBox is much like a call to GpiLine except that the given point designates the opposite corner of the rectangle being drawn. This function is used to implement the rectangle tool in the editor. Listing 3.7 shows how we use the GpiBox function.

```
void GoRectDrawDetails(GOBJ self)
{
    GpiMove(hps, &((*self->Points)[0]));
    GpiBox(hps, DRO_OUTLINE, &((*self->Points)[1]), 0L, 0L);
}
```

LISTING 3.7 GpiBox as used in the graphic editor.

Note that the GpiBox function has two additional parameters following the opposing corner point. These parameters are used to designate the rounding effect on the corners of the box. The corners are rounded by defining an ellipse shape. This shape is then fit into each corner of the box and the corners are rounded to the curve of the ellipse. Figure 3.7 shows how the ellipses are mapped on to the box to define its rounded corners.

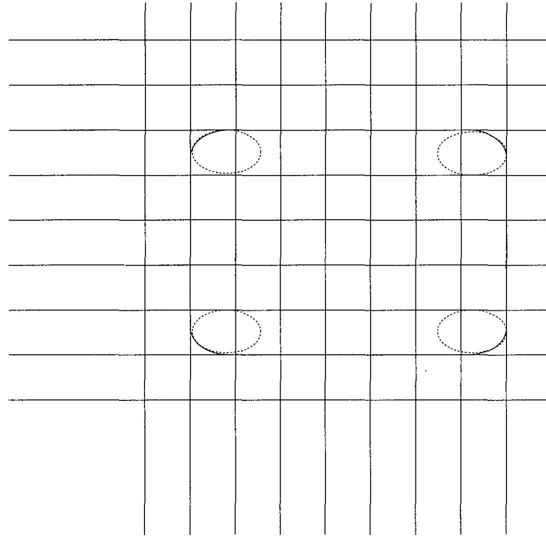


FIGURE 3.7 GpiBox rounded corners.

The first rounding parameter specifies the horizontal length of the ellipse, while the second rounding parameter specifies the vertical height of the ellipse. If either of these parameters is set to zero, the corners of the box will be square.

The second parameter of the GpiBox call controls how the interior of the box is to be filled. It can take on the values `DRO_FILL`, `DRO_OUTLINE`, or `DRO_OUTLINEFILL`. If `DRO_OUTLINE` or `DRO_OUTLINEFILL` are specified, the outline of the box will be drawn using the current line bundle attributes. If `DRO_FILL` or `DRO_OUTLINEFILL` are specified, the interior of the box will be filled using the current area bundle attributes. See the area primitives section for more detail on area bundle attributes.

Simple Arc Functions

The GPI provides several functions to enable your application to generate arcs in a drawing. These functions can be broken into two groups, *simple arc* primitives and *multi-arc* primitives.

The simple arc primitives make use of a set of special attributes to define the general shape of the arc. The shape and orientation of the simple arcs are defined by an imaginary ellipse. That ellipse is constructed using four parameters: p, q, r, and s. These parameters combine as shown in Figure 3.8 to

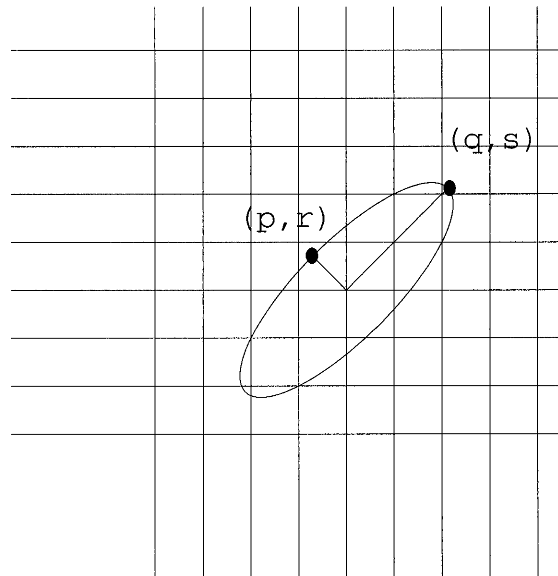


FIGURE 3.8 Definition of arc parameters.

define the major and minor axis of the ellipse, as well as its width to height ratio.

By adjusting the values of these parameters, the orientation and shape of the ellipse can be widely varied to produce the desired arc. The function called *GpiSetArcParams* is used to inform the GPI of the shape and orientation of this imaginary ellipse. Once set, the simple arc primitives will make use of the ellipse to determine the path the arc follows when it is actually drawn.

Circles and ellipses are both drawn using the function called *GpiFullArc*. A circle is just a special case of an ellipse or FullArc (as the GPI refers to it). When the FullArc is actually drawn it will be similar in shape to the ellipse defined by GpiSetArcParams, although its origin and scale can be adjusted. Listing 3.8 shows how our graphic editor uses the GpiFullArc function to implement the ellipse tool.

In our ellipse tool, the width and height of the ellipse parameters are set to exactly those of the ellipse that we want to draw. Because of this, we choose a scale factor of 1.0 when we issue the GpiFullArc call. By using a scale factor other than 1.0, the resulting ellipse can be made larger or smaller than the ellipse defined by the arc parameters. Figure 3.9 shows the various effects that can be achieved by using different scaling factors.

```
void GoEllipseDrawDetails(GOBJ self)
{
    ARCPARAMS arcParams;
    GpiMove(hps, &((*self->Points)[0]));
    arcParams.lP = (*self->Points)[1].x - (*self->Points)[0].x;
    arcParams.lQ = (*self->Points)[1].y - (*self->Points)[0].y;
    arcParams.lR = 0L;
    arcParams.lS = 0L;
    GpiSetArcParams(hps, &arcParams);
    GpiFullArc(hps, DRO_OUTLINE, MAKEFIXED(1,0));
}
```

LISTING 3.8 GpiFullArc as used in the graphic editor.

Note that we also assume that the major and minor axes line up with the X and Y axis of the world coordinate systems. By choosing different values for the r and s parameters we could have made the ellipse appear to be rotated. As you'll see in Chapter 6, "Transformations," there are other ways to produce a rotation effect as well.

As with the GpiBox function, the second parameter of the GpiFullArc function controls how the interior of the ellipse is to be filled. It can take on the values DRO_FILL, DRO_OUTLINE, or DRO_OUTLINEFILL. If DRO_OUTLINE or DRO_OUTLINEFILL are specified, the outline of the ellipse will be drawn using the current line bundle attributes. If DRO_FILL

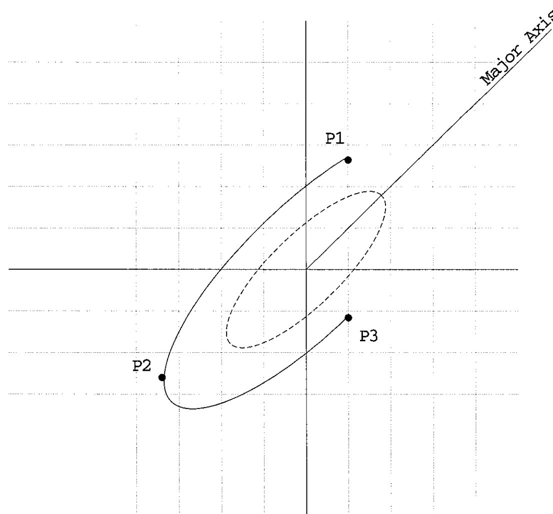


FIGURE 3.9 Scaled arc variations.

or DRO_OUTLINEFILL are specified then the interior of the ellipse will be filled using the current area bundle attributes. See the Area Primitives section for more detail on area bundle attributes.

The *GpiPartialArc* function lets you draw arcs that follow the ellipse defined by the arc parameters. When using this function, simply define the starting angle of the arc (measured counterclockwise from the major axis of the ellipse) and a sweep angle (measured in degrees to sweep around the ellipse). Figure 3.10 shows how start angle and sweep angle affect the drawing of a partial arc.

The arc parameters p, q, r, and s are also used to determine the direction (clockwise or counterclockwise) that the arc is drawn . Although not so important when drawing the entire 360 degrees of an ellipse, when drawing partial arcs this is significant since it determines the direction of the arc sweeping along the ellipse. Table 3.7 defines which direction the ellipse will be swept based on the arc parameters.

As you might suspect, the graphic editor uses the GpiPartialArc function to implement the 90 degree arc tool. Listing 3.9 shows a clip of that source code.

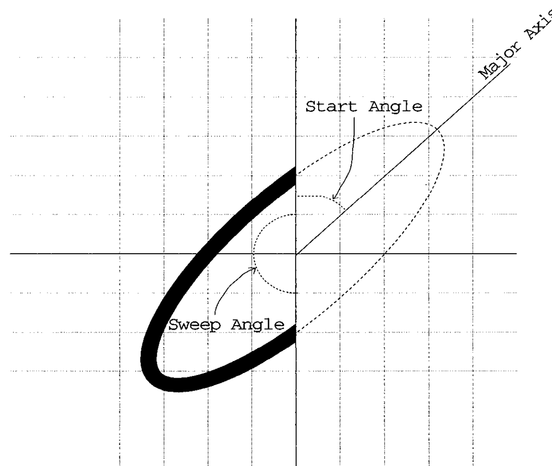


FIGURE 3.10 Start angle and sweep angle.

TABLE 3.7 Sweep direction

Arc Parameters	Trace Direction
pq < rs	Clockwise
pq > rs	Counterclockwise
pq = rs	Draw straight line instead of an ellipse

```
void GoArcDrawDetails(GOBJ self)
{
    ARCPARAMS arcParams;
    POINTL origin, *p1, *p2;
    FIXED startAngle, sweepAngle;
    p1 = &(*self->Points)[0];
    p2 = &(*self->Points)[1];
    /* define arc parameters such that result is a 90 degree arc   */
    /* drawn from p1 to p2. Start angle of 270 ensures that the    */
    /* arc will start out in a horizontal direction. Define arc    */
    /* such that its major axis lies along the X axis.             */
    origin.x = p1->x;
    origin.y = p2->y;
    arcParams.lP = p2->x - p1->x;
    arcParams.lQ = p2->y - p1->y;
    arcParams.lR = 0L;
    arcParams.lS = 0L;
    startAngle = MAKEFIXED(270,0);
    sweepAngle = MAKEFIXED(90,0);
    GpiSetArcParams(hps, &arcParams);
    GpiMove(hps, p1);
    /* move to 1st point (no chord wanted) */
    GpiPartialArc(hps, &origin, MAKEFIXED(1,0), startAngle,sweepAngle);

    /* draw from 1st point to 2nd point    */
}
```

LISTING 3.9 GpiPartialArc as used in the graphic editor.

Notice how the partial arc function accepts the origin as a parameter, while the full arc expects the origin to be the current position. That is because the partial arc function lets you begin drawing the arc at some place other than on the ellipse. When the function call is made, a straight line is drawn from the current point to the beginning point on the arc, from there the arc is swept as previously described. Since we didn't want a straight line to show

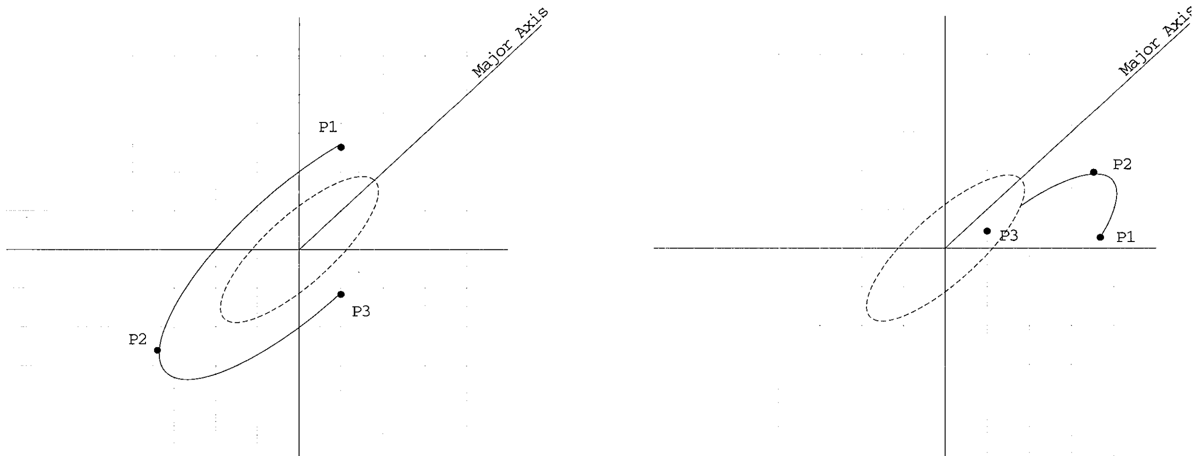


FIGURE 3.11 GpiPointArc scenarios.

up on our arc tool, we explicitly set the current position to be the same as the beginning point of the partial arc.

Another way to generate a partial arc is by using the function called *GpiPointArc*. This function is similar to the other arc function in that it uses the arc parameters to define the shape and orientation of the arc; however, the size and location of the point arc are defined by three additional points. Once given these three points, the ellipse is mapped on to the points and the connecting arc is drawn. The order of the points determines the direction the arc is swept. The arc starts at the first point, passes through the second point, and ends at the third point. Figure 3.11 shows several examples of how this mapping works.

Listing 3.10 shows how three-point arcs (such as those in Figure 3.11) can be created.

```
HPS hps;
ARCPARAMS arcParams;
POINTL arc1[3] = { {10,27}, {-35,-25}, {10,-10} };
POINTL arc2[3] = { {37,2}, {35,20}, {10,5} };
...
/* Define the shape of the arc using arc parameters */
arcParams.lR = -1;  /* Arc params can be to any scale */
```

LISTING 3.10 Using GpiPointArc.

```
arcParams.lQ =  1;
arcParams.lP =  2;
arcParams.lS =  2;
GpiSetArcParams(hps, &arcParams);
/* Draw the first arc around the origin */
GpiMove(hps, &origin);
GpiPointArc(hps, &arc1);
/* Draw the next arc around a different point but using the same shape */
GpiMove(hps, &origin);
GpiPointArc(hps, &arc2);
```

LISTING 3.10 (Continued).

Multi-Arc Functions

The GPI provides several other functions for drawing more complex curves. These functions do not depend on the arc parameters used by the simple arc functions; rather, these functions rely on a series of points and mathematical curve-fitting formulas.

The function called *GpiPolySpline* creates curves as a sequence of *Bezier splines*. Each spline in the sequence is defined by four points. The first spline is defined by points [cp,0,1,2] where cp is current position. The second spline is defined by points [2,3,4,5], the third by points [5,6,7,8], and so on. The spline passes through the two endpoints of the spline with the middle two points acting as control points for defining the shape of the curve. Figure 3.12 shows an example of output from a GpiSpline function.

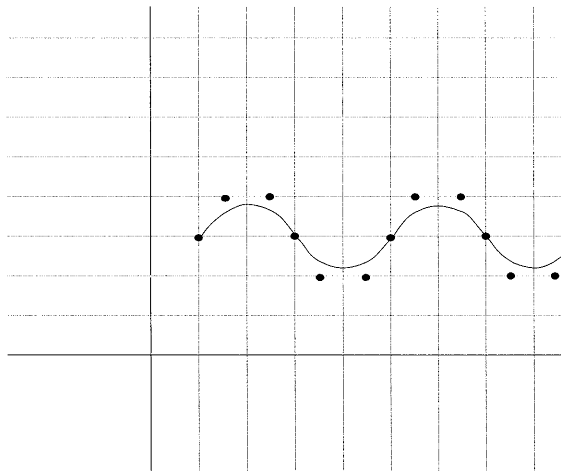


FIGURE 3.12 GpiPolySpline output.

Listing 3.11 is a code fragment that will produce Figure 3.12. Note that the number of points passed on this call must be a multiple of three.

```
HPS hps;
POINTL points[13] = {{10,30}, {15,40}, {25,40}, {30,30}, {35,20}, {45,20},
{50,30}, {55,40}, {65,40}, {70,30}, {75,20}, {85,20}, {90,30}};
/* Move to the first point and draw the spline */
GpiMove(hps, &points[0]);
GpiPolySpline(hps, 12L, &points[1]);
```

LISTING 3.11 Using GpiPolySpline.

Another useful function for generating curves is called *GpiPolyFillet*. This function draws a series of fillets. Due to the way these fillets are constructed, there is generally a smooth transition from one fillet to another resulting in a smooth curve. A single fillet is based on three points. An imaginary line is drawn from the current position to point1, and another from point1 to point2. The fillet is a curve that begins at the current position and is initially tangent to the first imaginary line. The curve then travels toward point1 and curves toward point2 such that it ends at point2 and is tangent to the second imaginary line. Figure 3.13 shows an example of a fillet.

When specifying more than three points, a sequence of fillets is constructed. This takes place by initially drawing a set of imaginary lines connecting all the given points. Then each line (except the first and last line) is split in half by a new imaginary point. The fillets are then drawn through all the points with each subsequent fillet beginning where the last fillet left off. Figure 3.14 shows an example of a series of fillets strung together.

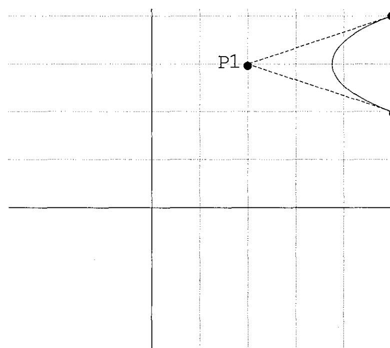


FIGURE 3.13 Single fillet.

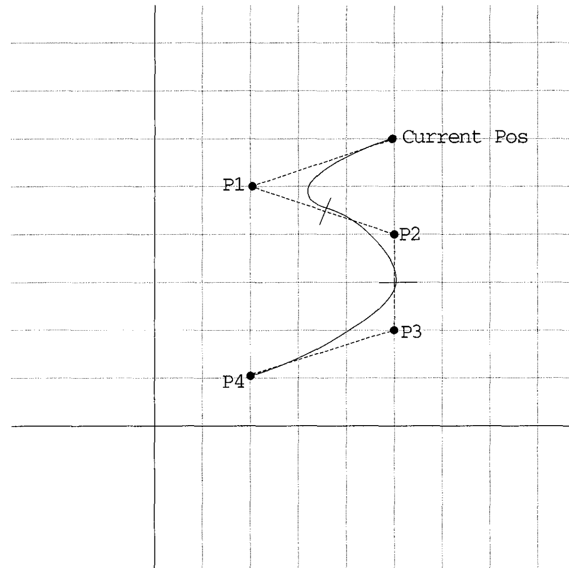


FIGURE 3.14 Series of fillets.

Calling GpiPolyFillet is the same as calling GpiPolySpline but the number of points does not have to be a multiple of three. The graphic editor makes use of this function to implement the fillet tool as shown in Listing 3.12.

```
voidGoPolyfilletDrawDetails(GOBJ self)
{
    GpiMove(hps, &((*self->Points)[0]));
    GpiPolyFillet(hps, self->PointCount-1, &((*self->Points)[1]));
}
```

LISTING 3.12 GpiPolyFillet as used in the graphic editor.

The function called *GpiPolyFilletSharp* is similar to the GpiPolyFillet function in that it is drawn as a series of fillets. However, with this function, the fillets do not necessarily have a continuous gradient from one fillet to another. In other words, the result is not necessarily a smooth curve unless the points are chosen just so. In addition, this function lets you control the sharpness of each fillet curve. The sharpness of a fillet is a ratio that is best explained by looking at Figure 3.15.

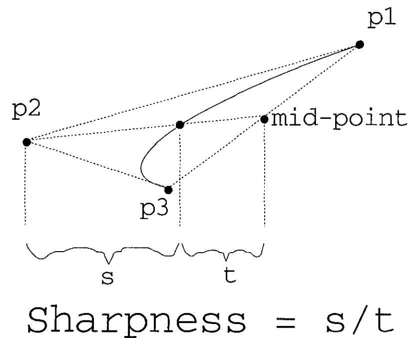


FIGURE 3.15 Fillet sharpness ratio.

The sharpness of a fillet is defined using three points and a sharpness ratio. Given three points p1, p2, and p3, the fillet will always begin at p1 and end at p3. The sharpness ratio defines where the fillet will cross the imaginary line segment from point p2 to the midpoint of line segment p1-p3. This is shown in Figure 3.15.

With the GpiPolyFilletSharp function, you pass in an array of values that define the sharpness of each fillet drawn. Unlike the GpiPolyFillet function, this function does not insert any mid-points for determining fillet connections. With this function, a fillet is defined by three points in the array, and each fillet begins where the previous one left off. Therefore, every two points begins the start of a new fillet. As a result, the number of points passed in must be a power of two, and one sharpness value is required for each fillet.

The code segment in Listing 3.13 shows how to produce a simple polyfillet using sharpness values to produce an interesting diminishing wave pattern. Figure 3.16 shows what the resulting output would look like.

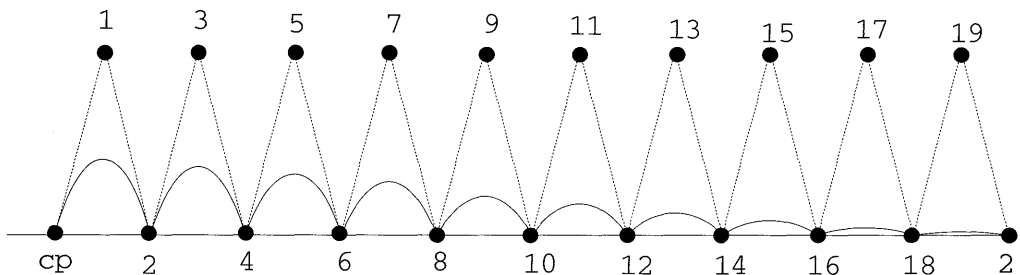


FIGURE 3.16 GpiPolyFilletSharp output.

```
HPS hps;
POINTL points[21] = {
   { 0, 0}, {10, 20}, {20, 0}, {30, 20}, {40, 0}, {50, 20}, {60, 0},
   {70, 20},{80, 0}, {90, 20}, {100,0}, {110,20}, {120,0}, {130,20},
   {140,0}, {150,20},{150,0}, {170,20}, {180,0}, {190,20}, {200,0}
};
FIXED sharpness[10];
int i;
float j;
 ...
/* Compute deminishing sharpnesses ratios */
j=2;
for(i=1; i<=10; i++) {
   sharpness[i] = FLOAT2FIX(j);
   j = j/(1.5);
}
/* Move to the first point and draw the polyfillet */
GpiMove(hps, &points[0]);
GpiPolyFilletSharp(hps, 20L, &points[1], sharpness);
  ...
```

LISTING 3.13 Using GpiPolyFilletSharp.

AREA PRIMITIVES

Drawing lines and arcs is fine for many parts of a drawing. But sometimes you may want to fill in the interior of some of the figures you are drawing. This section describes what functions can be used to create enclosed areas and what special attributes exist for determining how those areas will look when displayed. Table 3.8 lists the functions in this category and gives a brief description of the operations they perform.

The attribute bundle for areas is specific for this category of drawing primitives. Setting of attributes in this bundle will affect the display of figures drawn using these operations. Table 3.9 describes each attribute in this bundle.

The *Area Color* attribute defines what color is used to draw the interior foreground of any areas to be filled. For example, if an ellipse is to be filled with a diagonal line pattern, the lines in the diagonal pattern will be drawn in the Area Color.

TABLE 3.8 Area functions

Function	Description
GpiBeginArea	Begins definition of an area.
GpiEndArea	Completes definition of an area.
GpiBeginPath	Begins definition of a path.
GpiEndPath	Completes definition of a path.
GpiFillPath	Fills the contents of a path.
GpiStrokePath	Strokes geometric lines in a path.
GpiModifyPath	Converts path into new path containing only geometric lines.
GpiCloseFigure	Completes definition of a figure in a path.
GpiPolygons	Draws a series of polygons using area fill attributes.
GpiBox	Draws a rectangle using area fill attributes.
GpiFullArc	Draws an ellipse using area fill attributes.

TABLE 3.9 Area attribute bundle

Attribute Description	AREABUNDLE Field Name	flAttrMask Value	Default Value	Helper Function
Area color	lColor	ABB_COLOR	CLR_BLACK	GpiSetColor*
Area background color	lBackColor	ABB_BACK_COLOR		GpiSetBackColor*
Area mix	usMixMode	ABB_MIX_MODE	FM_OVERPAINT	GpiSetMix*
Area backgound mix	usBackMixMode	ABB_BACK_MIX_MODE	FM_LEAVEALONE	GpiSetBackMix*
Pattern set	usSet	ABB_SET	LCID_DEFAULT	GpiSetPatternSet GpiQueryPatternSet
Pattern symbol	usSymbol	ABB_SYMBOL	Solid	GpiSetPattern GpiQueryPattern
Pattern reference pattern	ptlRefPoint	ABB_REF_POINT	(0,0)	GpiSetPatternRefPoint GpiQueryPatternRefPoint

*** These helper functions set the color and mix attributes in all bundles.**

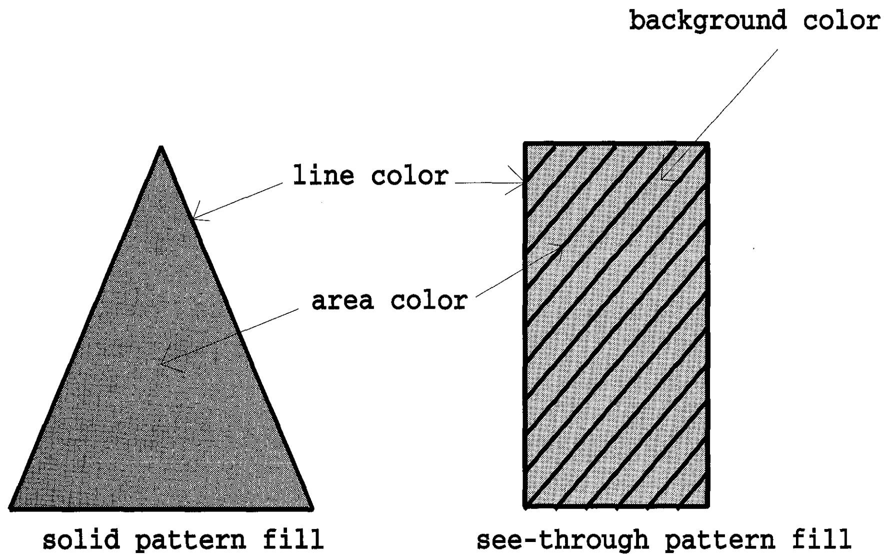


FIGURE 3.17 Colors in a pattern-filled area.

The *Area Background Color* attribute defines what color is used to draw the interior background of any areas to be filled. In the ellipse example given above, the area between the diagonal fill lines would be drawn in the Area Background Color. Figure 3.17 shows how area, color, and background color are used to define the colors in a pattern-filled area.

The *Area Mix* attribute defines how the area foreground color is mixed with the color of other figures on the drawing surface. Typically when the specified foreground color is desired, this attribute is set to FM_OVERPAINT.

Likewise, the *Area Background Mix* attribute defines how the area background color is mixed with the color of other figures on the drawing surface. One typical setting for this attribute is BM_OVERPAINT. This causes the background to be drawn in the specified background color. Another common setting is BM_LEAVEALONE. This causes the background to be invisible (you can see through the area if the fill pattern has exposed background areas). We use the BM_LEAVEALONE attribute in the graphical editor when we use a fill style other than solid. This produces the see-through effect that is apparent when objects overlap each other.

The *Pattern Set* attribute selects which set of pattern symbols are to be used for drawing filled areas. The GPI provides a default set of pattern symbols specified by a value of LCID_DEFAULT. You can create custom pattern sets with pattern symbols created using bitmaps or characters from a raster font.

If creating from a bitmap there is only one pattern symbol in the set and it is automatically chosen when the Pattern Set attribute is set. If creating from a font, each character in the font is available as a symbol in the pattern set.

TABLE 3.10 Default pattern set symbols

Symbol Value	Description
PATSYM_DENSE1	Shaded fill pattern (darkest)
PATSYM_DENSE2	
PATSYM_DENSE3	
PATSYM_DENSE4	
PATSYM_DENSE5	
PATSYM_DENSE6	
PATSYM_DENSE7	
PATSYM_DENSE8	Shaded fill pattern (lightest)
PATSYM_VERT	Vertical lines
PATSYM_HORIZ	Horizontal lines
PATSYM_DIAG1	Lower left to upper right diagonal lines
PATSYM_DIAG2	Lower left to upper right diagonal lines (closely spaced)
PATSYM_DIAG3	Upper left to lower right diagonal lines
PATSYM_DIAG4	Upper left to lower right diagonal lines (closely spaced)
PATSYM_NOSHADE	Completely empty
PATSYM_SOLID	Completely solid
PATSYM_HALFTONE	Every other pel is set
PATSYM_HATCH	Grid
PATSYM_DIAGHATCH	Diagonal Grid
PATSYM_BLANK	Blank (same as PATSYM_NOSHADE)
PATSYM_DEFAULT	Default pattern symbol (initially set to PATSYM_SOLID)

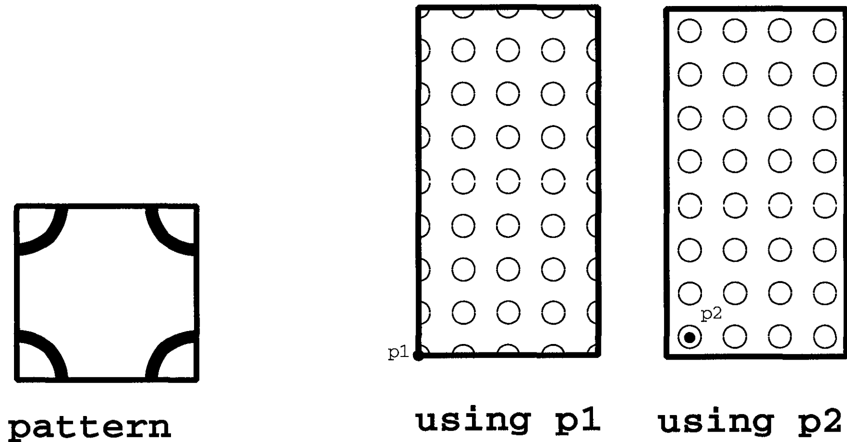


FIGURE 3.18 Pattern reference point attribute effects.

The *Pattern Symbol* attribute selects which symbol in the pattern set is to be used to fill the area. Table 3.10 lists the symbols of the default pattern set.

The *Pattern Reference Point* attribute allows you to shift the alignment of the pattern symbol that is used to fill the area. The value of this attribute defines with what the lower-left corner of the pattern symbol is to align. The entire fill pattern will then shift to match this alignment. The reference point is defined in world coordinates and defaults to point (0,0). Figure 3.18 shows how the Pattern Reference Point affects alignment of the fill pattern.

Listing 3.14 shows an example of how to use the Pattern Set, Pattern Symbol, and Pattern Reference Point attributes.

```
HPS hps;
LONG myFontId;
POINTL midPoint;
LONG charWidth, charHeight;
/* Choose a character out of a font set as the fill pattern */
GpiSetPatternSet(hps, myFontId);
GpiSetPattern(hps, '$');
/* set the reference point to the middle of the character */
determine width and height of character in world coordinates
midPoint.x = -charWidth/2;
midPoint.y = -charHeight/2;
GpiSetPatternRefPoint(hps, &midPoint);
/* Draw objects that are to be filled */
```

LISTING 3.14 Using pattern attributes.

AREA FUNCTIONS

There are several ways to fill the interior of the objects. First, some of the GPI drawing functions allow you to specify directly on the call whether they are to be filled or not. For example, as shown in Listing 3.15, the second parameter of the GpiBox() function call lets you specify whether the box should be filled, outlined, or both. If the box is to be filled, this is done using the current area bundle attributes.

```
void GoRectDrawDetails(GOBJ self)
{
GpiMove(hps, &((*self->Points)[0]));
GpiBox(hps, DRO_FILL, &((*self->Points)[1]), 0L, 0L);
}
```

LISTING 3.15 Filling areas using GpiBox.

A second method of filling an area is by defining the area as a set of one or more closed figures. This is done by enclosing a series of GPI drawing primitives between the area bracket functions called *GpiBeginArea* and *GpiEndArea*. When the GpiEndArea function is called, the figures in the area are drawn and filled using the area bundle attributes that were in effect when the GpiBeginArea function was called. Note that areas cannot be nested within other areas. Listing 3.16 shows an example of using area bracketing functions.

```
GpiBeginArea(hps, BA_BOUNDARY | BA_ALTERNATE);
/* Various GPI calls to be included in the area */
GpiEndArea(hps);
```

LISTING 3.16 Basic form of area bracket functions.

The GpiBeginArea function has an *flOptions* parameter that lets you control how the area is constructed. This parameter is really a combination of two flags and you specify its value by ORing the desired flag values together. The first flag determines whether or not to draw the boundary lines of primitives drawn within the area brackets. The available values for this flag are:

- `BA_BOUNDARY` – Draw outlines of the area primitives (default).
- `BA_NOBOUNDARY` – Do not draw the outlines.

If `BA_BOUNDARY` is chosen, the outlines of all area primitives within the area bracket will be drawn. These outlines will be drawn using the cur-

rent line bundle attributes, while the interiors will be drawn using the current area bundle attributes. If BA_NOBOUNDARY is chosen, the outlines will not be drawn, only the interior.

If you recall, the GpiBox function had a *control* parameter that let you specify if the outline, the interior, or both parts of the box should be drawn (DRO_FILL, DRO_OUTLINE, and DRO_OUTLINEFILL, respectively). When you choose anything other than DRO_OUTLINE, the GPI automatically places an implicit area bracket around the GpiBox call and sets the appropriate boundary flag. Since areas cannot be nested, the GpiBox cannot be used within an area bracket if DRO_FILL or DRO_OUTLINEFILL are used.

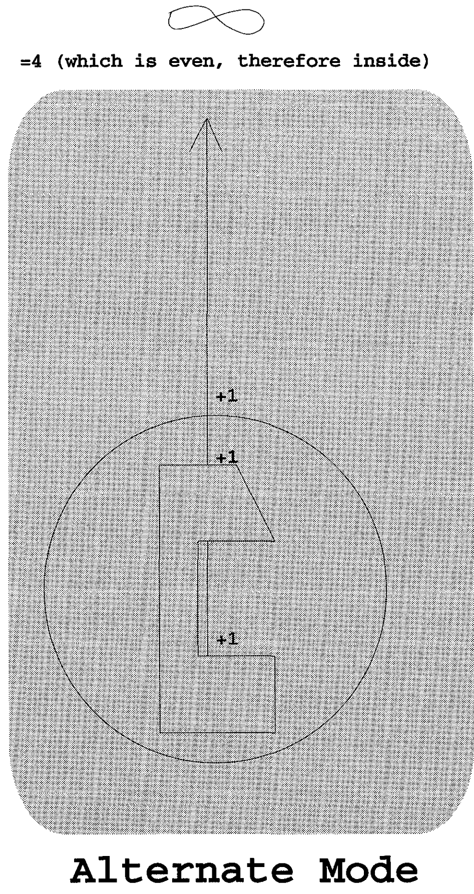


FIGURE 3.19 Fill area determination using BA_ALTERNATE.

The second flag determines how the interior of the area will be constructed. The available choices for this flag are:

- BA_ALTERNATE (default)
- BA_WINDING

Given any particular point in the area figure, this flag determines which algorithm will be used to determine if that point is inside an area that needs to be filled or not. If the BA_ALTERNATE flag is chosen, the algorithm goes as follows: first, draw a line from the current point towards infinity (in any direction); next, count how many boundary lines are crossed as the line travels toward inifinity. If the number of crossings is odd, then it is in the interior of an area that needs to be filled. If the number is even, then the point is not inside a fill area. Figure 3.19 shows graphically how this works.

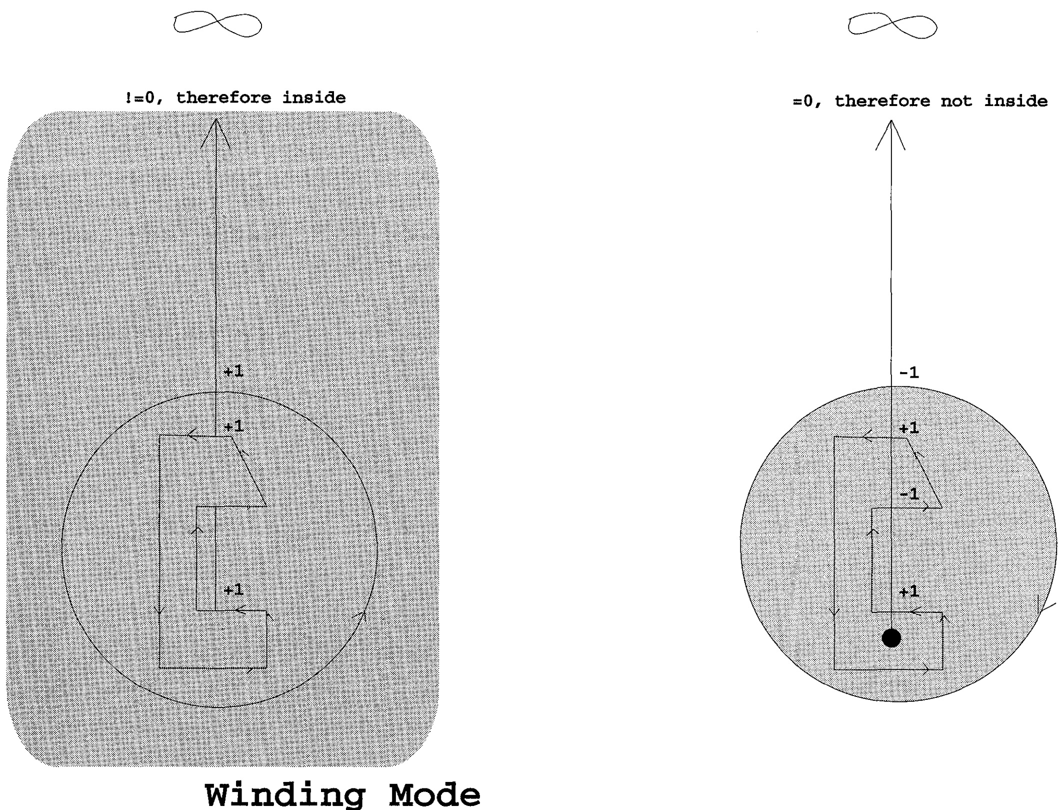


FIGURE 3.20 Fill area determination using BA_WINDING.

If the BA_WINDING flag is chosen, the algorithm changes slightly. A line is still drawn from the current point towards inifinity, but now as the line crosses boundary lines, the crossing count can be either incremented or decremented depending upon the direction of the line that was crossed over. If the line being crossed was drawn in a counterclockwise direction (with respect to the initial point in question) the count is incremented. If it was drawn in a clockwise direction, the count is decremented. After all crossings are counted, if the final count is not zero, the point is within an area to be filled. If the final count is zero, the point is not in a filled area. Figure 3.20 shows graphically how the fill area is determined.

The direction of open figures is simply the direction from the starting point towards the endpoint (following the intermediate points if a multi-point figure). The direction of lines in closed figures varies. The GpiBox figure can be drawn in either direction. If the imaginary line drawn between the endpoints of the box has a positive slope, the box is drawn counterclockwise, if it has a negative slope, the box is drawn clockwise. The direction of arcs can be controlled by the arc parameters (p, q, r, and s) and the sweep angle. Listing 3.17 draws the rightmost area shown if Figure 3.20.

```
ARCPARAMS arcParams;
POINTL circle = {50,50};
POINTL polygon[9] = {{40,15}, {70,15}, {70,35}, {50,35}, {50,75}, {70,75},
{60,95}, {40,95}, {40,15}};
/* Draw the Area in winding mode */
GpiBeginArea(hps, BA_BOUNDARY | BA_WINDING);
arcParams.lP = 1L;  // Define arc parameters for a circle arcParams.
lQ =-1L; // drawn in clockwise direction arcParams.lR = 0L;
arcParams.lS = 0L;
GpiSetArcParams(hps, &arcParams);
GpiMove(hps, &circle);
GpiFullArc(hps, DRO_OUTLINE, MAKEFIXED(60,0));
GpiMove(hps, &polygon[0]);
GpiPolyLine(hps, 8L, &polygon[1]);
GpiEndArea(hps);
```

LISTING 3.17 Filling areas using area bracket functions.

A third way to fill figures is by using *Paths*. Paths work similarly to areas in that they bracket a series of GPI calls. The path definition is started with a function called *GpiBeginPath* and completed with a function called *GpiEndPath*. Once the path is complete, a function called *GpiFillPath* can be used to fill the path according to the current area bundle attributes. Paths, however, also provide a different capability known as *geometric lines*. Basically, geometric lines are lines whose widths can vary. Using the function

called *GpiModifyPath*, you can convert the path into a new path containing only geometric lines. After using GpiModifyPath, you can then use GpiFillPath to draw the cosmetic lines. The GPI provides a function that combines these two steps into one. This function is called *GpiStrokePath* and using it causes all the lines in the path to be drawn using the current geometric line width and the current fill attributes (just like calling GpiModifyPath followed by GpiFillPath).

The function called *GpiCloseFigure* can be used within a Path definition to close off the current figure with a line drawn from the current position to the beginning of the figure. However, this function can only be used inside a path definition, not within an area definition.

The graphic editor uses paths both to fill the interiors of figures and to draw geometric lines. Listing 3.18 shows how paths are used to draw the interior of the object.

```
// Draw the interior of the object
if (self->Fillable) {
    GpiBeginPath(hps, 1L);
    self->Do->DrawDetails(self);
    GpiEndPath(hps);
    GpiFillPath(hps, 1L, FPATH_ALTERNATE);
}
```

LISTING 3.18 Filling areas using paths in the graphic editor.

Listing 3.19 shows how paths are used to draw geometric lines.

```
if (self->Attribs.LineWidth==0) {
    // if width is 0 draw cosmetic lines
    self->Do->DrawDetails(self);
} else {
    // otherwise draw geometric lines using a path
    GpiBeginPath(hps, drawPath);
    self->Do->DrawDetails(self);
    GpiEndPath(hps);
    areaAttr.lColor = self->Attribs.LineColor;
    areaAttr.usSymbol = PATSYM_SOLID;
    GpiSetAttrs(hps, PRIM_AREA, ABB_COLOR | ABB_SYMBOL,0L, &areaAttr);
    GpiStrokePath(hps, drawPath, strokeOptions);
} /* endif */
```

LISTING 3.19 Drawing cosmetic lines in the graphic editor.

This example illustrates using paths both for area fills and for geometric lines. It also updates the area bundles color and symbol attributes. These attributes will be used when stroking the path. A more in-depth discussion on paths and their use in clipping is given in Chapter 7, "Paths, Regions, Clipping, Boundary Accumulation, and Correlation."

Finally, the function called *GpiPolygons* allows you to draw a large number of polygons in a single call. This function accepts an array of polygon structures, each of which defines a single polygon. Also specified in the call is a drawing options parameter. This parameter works just like the options flag on the GpiBeginArea function. The option flags for this call are:

- POLYGON_BOUNDARY (default)
- POLYGON_NOBOUNDARY

and

- BA_ALTERNATE (default)
- BA_WINDING

In addition, a *model* parameter is provided to define whether the bottom-right edges of the polygon are included in the fill area. Listing 3.20 shows an example of how this function can be used. The resulting output is shown in Figure 3.21.

```
LINEBUNDLE lineAttr;
AREABUNDLE areaAttr;
POINTL p1[5] = {{10,0}, {70,0}, {70,50}, {10,50}, {10,0}};
POINTL p2[5] = {{0,90}, {40,70}, {70,70}, {30,90}, {0,90}};
POINTL p3[7] = {{100,0}, {170,0}, {230,40}, {230,110}, {180,130},
          {100,110}, {100,0}};
POINTL p4[9] = {{130,10}, {160,10}, {190,30}, {190,50}, {180,70},
         {170,50}, {170,30}, {130,30}, {130,10}};
POINTL p5[5] = {{120,70}, {140,50}, {160,80}, {140,100}, {120,70}};
POLYGON polygons[5] = { /* Polygon definitions */
    {5, p1},
    {5, p2},
    {7, p3},
    {9, p4},
    {5, p5}
};
/* Set line and area attributes */
lineAttr.lColor = CLR_BLACK;
lineAttr.usType = LINETYPE_SOLID;
GpiSetAttrs(hps, PRIM_LINE, LBB_COLOR | LBB_TYPE, 0L, &lineAttr);
areaAttr.lColor = CLR_PALEGRAY;
areaAttr.usSymbol = PATSYM_SOLID;
GpiSetAttrs(hps, PRIM_AREA, ABB_COLOR | ABB_SYMBOL, 0L, &areaAttr);
/* now draw the polygons */
GpiPolygons(hps, 5L, polygons, POLYGON_BOUNDARY |
POLYGON_ALTERNATE, POLYGON_INCL);
```

LISTING 3.20 Using GpiPolygons.

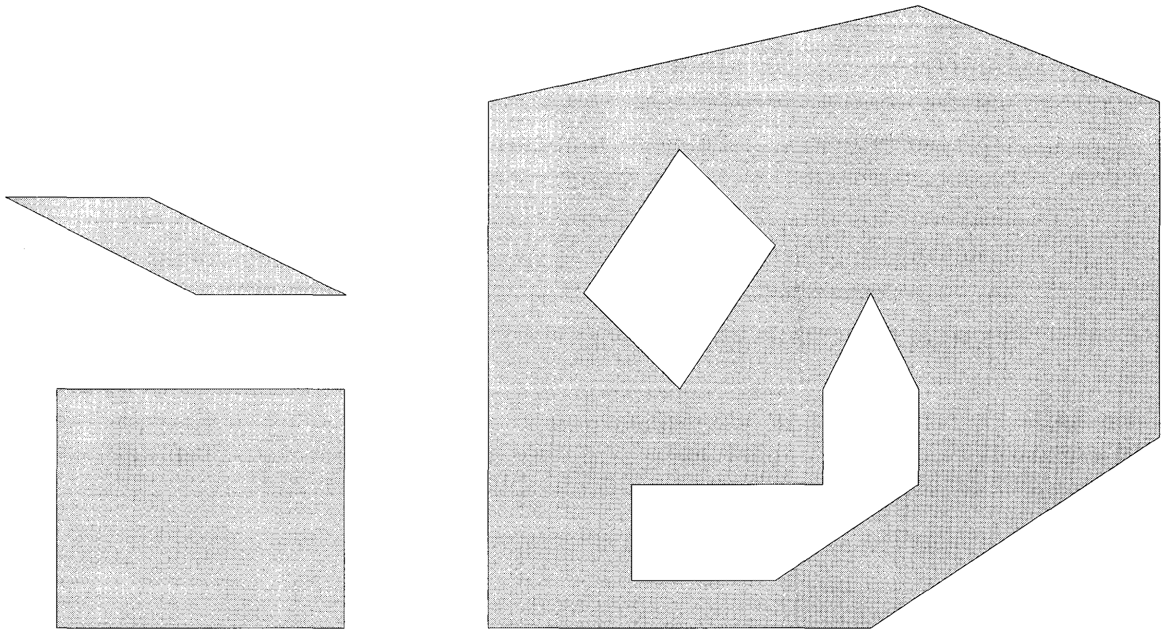


FIGURE 3.21 GpiPolygons output.

As with primitives drawn within area brackets, polygons drawn with this call are outlined using the line bundle attributes and filled using the area bundle attributes.

TEXT PRIMITIVES

We've discussed many calls for drawing figures using various line, arc, and area calls. Now, how about satisfying the basic call for any C programmer... *"Just tell me how to write 'Hello World' to the screen!"* Well, you don't use *print* but it's not that much different. Besides, the GPI provides enough attributes in the text bundle to let you write 'Hello World' with real style! As a starting point, Table 3.11 gives a quick overview of the functions in this category and the operations they perform.

The attribute bundle for text is specific for this category of drawing primitives. Setting of attributes in this bundle affects how text drawn using these operations is displayed. Table 3.12 describes each attribute in this bundle.

TABLE 3.11 Text functions

Function	Description
GpiCharString	Draws text at current position.
GpiCharStringAt	Draws text at specified position.
GpiCharStringPos	Draws text at current position using widths table.
GpiCharStringPosAt	Draws text at specified position using widths table.

TABLE 3.12 Text attribute bundle

Attribute Description	TEXTBUNDLE Field Name	flAttrMask Value	Default Value	Helper Function
Character color	lColor	CBB_COLOR	CLR_BLACK	GpiSetColor*
Character background color	lBackColor	CBB_BACK_COLOR	clear	GpiSetBackColor*
Character mix	usMixMode	CBB_MIX_MODE	FM_OVERPAINT	GpiSetMix*
Character background mix	usBackMixMode	CBB_BACK_MIX_MODE	FM_LEAVEALONE	GpiSetBackMix*
Character set	usSet	CBB_SET	LCID_DEFAULT	GpiSetCharSet GpiQueryCharSet
Character mode	usPrecision	CBB_MODE	CM_MODE1	GpiSetCharMode GpiQueryCharMode
Character box	sizfxCell	CBB_BOX	outline font – by dev raster font – by font	GpiSetCharBox GpiQueryCharBox
Character angle	ptlAngle	CBB_ANGLE	(1,0)	GpiSetCharAngle GpiQueryCharAngle
Character shear	ptlShear	CBB_SHEAR	(0,1)	GpiSetCharShear GpiQueryCharShear
Character direction	usDirection	CBB_DIRECTION	left to right	GpiSetCharDirection GpiQueryCharDirection
Character text alignment	usTextAlign	CBB_TEXT_ALIGN	left	GpiSetTextAlignment GpiQueryTextAlignment
Character extra	fxExtra	CBB_EXTRA	0	GpiSetCharExtra GpiQueryCharExtra
Character break extra	fxBreakExtra	CBB_BREAK_EXTRA	0	GpiSetCharBreakExtra GpiQueryCharBreakExtra

The attributes in this table are covered briefly below. For a more detailed discussion on these attributes, refer to Chapter 4, “Fonts.”

The attribute called *Character Color* defines in which color the characters symbols are drawn. The *Character Background Color*,meanwhile, determines the color of the box behind the character symbol.

The attributes *Character Mix* and *CharacterBackground Mix* determine how the character color and character background color are combined with the colors of what was previously drawn.

The current font is chosen by setting the *Character Set* attribute. This attribute is the local identifier (*lcid*) that was associated with a font using the function *GpiCreateLogicalFont*.

The GPI allows two types of fonts to be used: raster fonts and outline fonts. The attribute called *Character Mode* influences how raster font text is drawn. There are three possible values for this attribute (`CM_MODE1`, `CM_MODE2`, and `CM_MODE3`). When `CM_MODE1` is used, raster font text strings ignore all character attributes except character direction. When `CM_MODE2` is used, raster font characters are positioned properly according to character shear, box, angle, and direction attributes but the characters themselves are not affected by these attributes. `CM_MODE3` is only valid for drawing outline fonts. Outline fonts will always follow to the current character attributes for position and display. The value of the character mode attribute has no effect how outline fonts are drawn.

The *Character Box* attribute specifies the width and height of the character cell in world coordinates. The cell size is mainly used for positioning characters when the text is drawn. Depending on the type of font and the

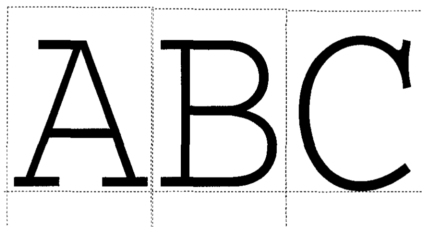


FIGURE 3.22 Character box effect on outline font character.

current character mode, various results can occur. When using an outline font, the character will always scale up or down to fit the size of the character cell. This happens regardless of the character mode. Figure 3.22 shows the effect of character box on outline font characters.

When using a raster font, however, the characters cannot be scaled to fit the character box. If the character mode is CM_MODE1, the character cell size is completely ignored. If the character mode is CM_MODE2, the character cell size is used to position the text characters that are drawn; that is, a large cell size will cause the characters to be more widely separated, while a smaller cell size will cause them to be more closely placed. Figure 3.23 shows the effect of character box on raster font characters.

A negative value for character box width causes the text to be written backward (right to left). Likewise, a negative character box height causes the text to be reflected along the X axis to appear upside down. Figure 3.24 shows this *mirroring* effect of a negative character box when applied to outline font text.

The direction in which the text is written can be controlled using the attribute called *Character Angle*. This attribute determines the direction of the baseline that subsequent text strings will be drawn along. This direction is specified in degrees of rotation from the X axis. Figure 3.25 shows the effect of the character angle attribute on outline font text.

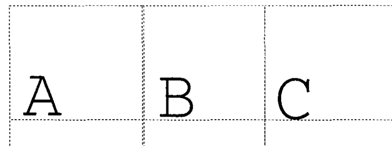


FIGURE 3.23 Character box effect on raster font character.

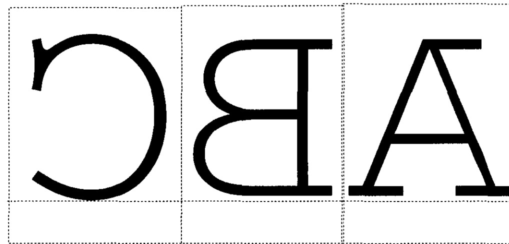


FIGURE 3.24 Mirroring outline font text using negative char box width.

How raster font text strings are drawn depends on the character mode. When character mode `CM_MODE1` is in effect, raster font text strings will ignore the character angle completely and simply draw as usual. When using character mode `CM_MODE2`, however, raster font text strings will be drawn along the rotated baseline. The characters themselves will not be rotated, but they will be individually positioned such that they travel at the specified angle. Figure 3.26 shows the effect of character angle on raster font text.

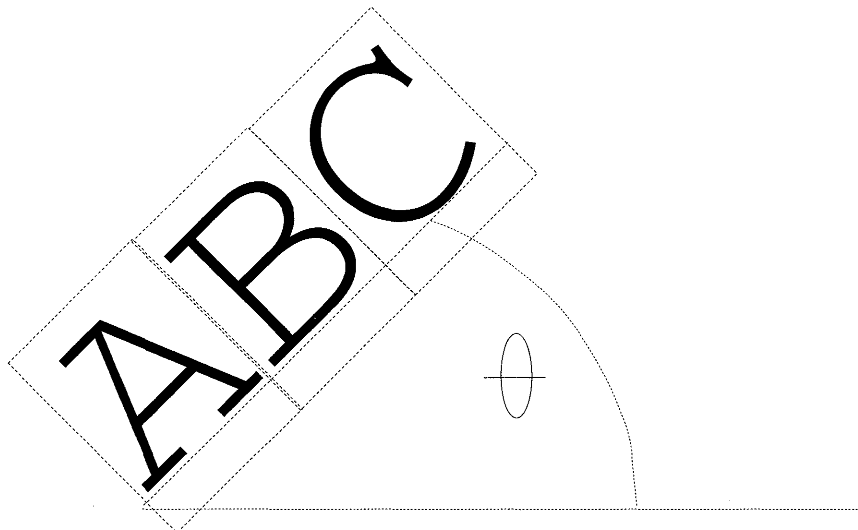


FIGURE 3.25 Character angle effect on outline fonts.

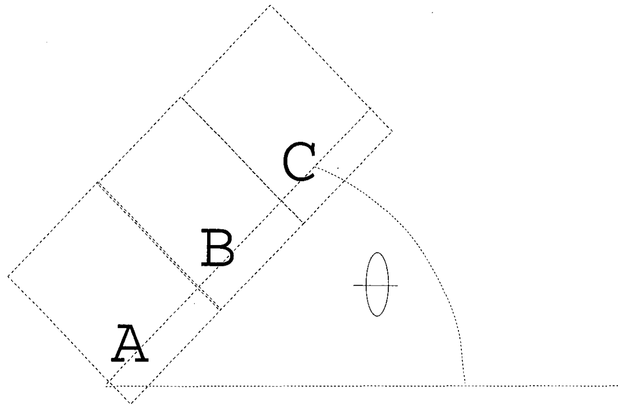


FIGURE 3.26 Character angle effect on raster fonts.

The *Character Shear* attribute is used to specify the amount to horizontally shear the character cell box by. Although shear is typically specified as an angle, the GPI uses an (x,y) pair to define the shear. The shear angle is defined by a line that begins at point (0,0) and ends at the specified point. Figure 3.27 shows the effect of character shear on outline font text.

Outline font text strings are sheared as shown in Figure 3.27. Raster font text strings, however, are drawn differently depending upon the character mode. When character mode is set to `CM_MODE1`, the character shear is completely ignored by raster font text strings. When character mode is set to `CM_MODE2`, raster font strings use the character shear attribute to position the characters in the string. Figure 3.28 shows the effect of character shear on raster font text.

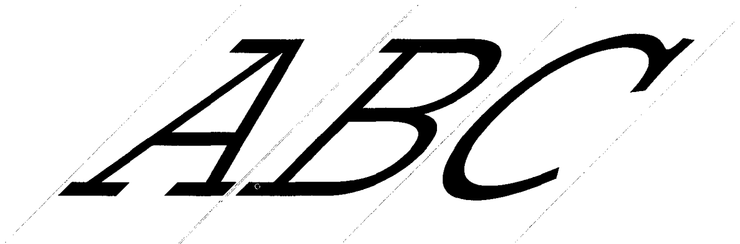


FIGURE 3.27 Character shear effect on outline fonts.

FIGURE 3.28 Character shear effect on raster fonts.

The direction that text strings are drawn can be controlled using the *Character Direction* attribute. This attribute can be one of four possible values:

- CHDIRN_LEFTRIGHT
- CHDIRN_RIGHTLEFT
- CHDIRN_TOPBOTTOM
- CHDIRN_BOTTOMTOP

Normally text strings are drawn left to right; but, by changing this attribute you can force them to be drawn in alternate directions. Figure 3.29 shows the effect of character direction on outline or raster font text.

The *Character Text Alignment* attribute is used to specify how the text string is to be positioned relative to the current position. Horizontally, the text string can be positioned using the one of the following choices:

- TA_LEFT – Leftmost edge of string is aligned with current position(default).
- TA_RIGHT – Rightmost edge of string is aligned with current position.
- TA_CENTER – Center of string is aligned with current position.

Vertically, the text string can be positioned using one of the following:

- TA_TOP – Top edge of string is aligned with current position.
- TA_HALF – Middle vertical position of string is aligned with current position.

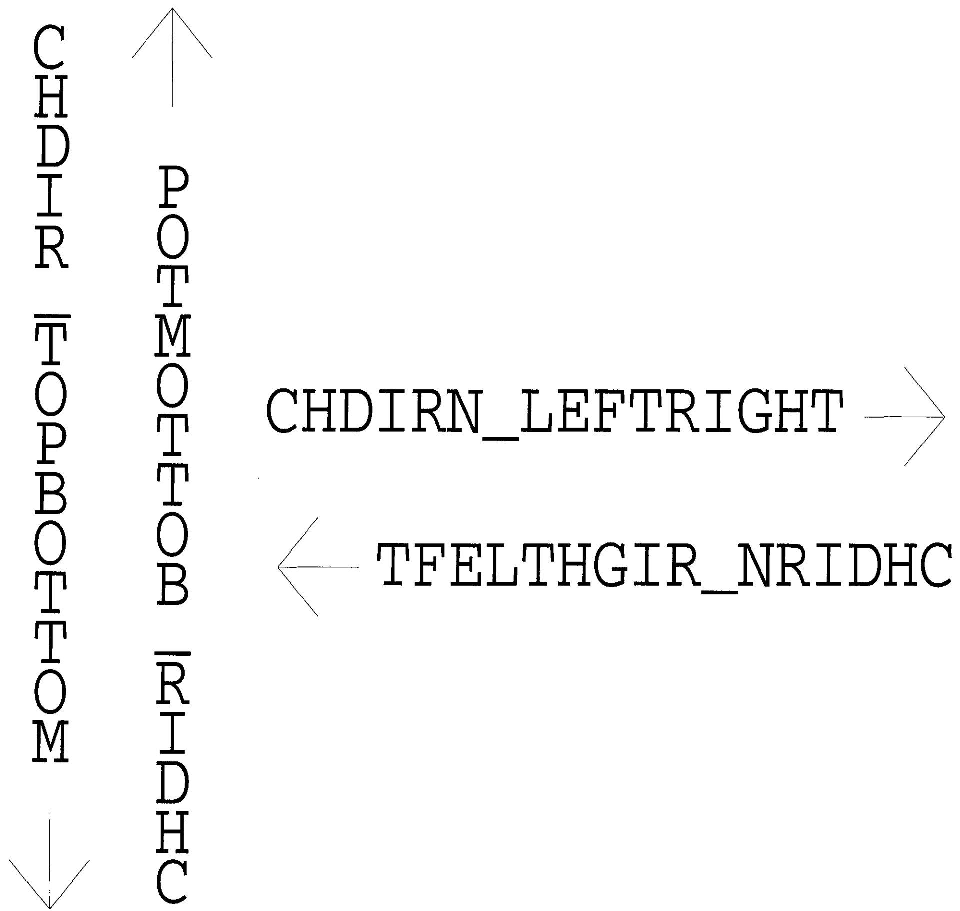


FIGURE 3.29 Character direction effect on text.

- `TA_BASE` – Baseline of string is aligned with current position.
- `TA_BOTTOM` – Bottom edge of string is aligned with current position.

Note that when setting the text alignment attribute, you must set it to the sum of the horizontal attribute and the vertical attribute. For example, to center the text around the specified point, set the attribute as follows:

charAttrs.usTextAlign = `TA_CENTER + TA_HALF`

Figure 3.30 shows the effects of various character text alignments on outline or raster font text.

FIGURE 3.30 Character text alignment effects.

The attributes *Char Extra* and *Char Break Extra* provide additional spacing control on devices that support them. The char extra attribute specifies additional space to be inserted between each character cell when the text string is drawn. The char break extra attribute specifies additional space to be inserted when the *break character* is encountered when drawing the text string. The break character is defined by the font and is usually the space character. If the value of either of these attributes is zero, no additional space is added. If the values are negative, space is subtracted and the character cells may actually overlap. Figure 3.31 shows the effect of char extra and char break extra on outline and raster font text.

Text Functions

There are really only a few functions in the GPI for generating text, but the variety of attributes allows you to create many different looks for your text. The simplest text function is called *GpiCharString*. This function draws a

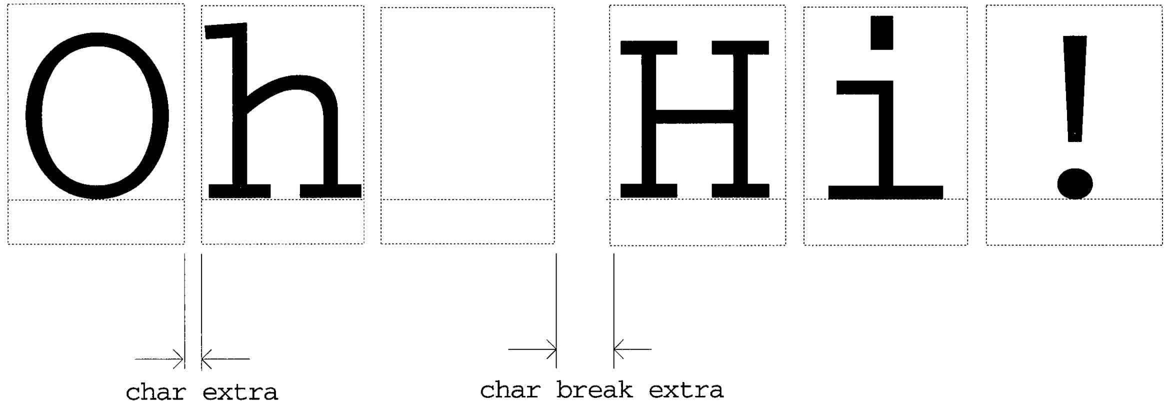


FIGURE 3.31 Char break and char break extra effects on text.

string to the current position on the device. The text is drawn using the current values of the attribute bundle. The current position is the beginning of the lower-left corner of the initial character.

The function called *GpiCharStringAt* allows you to specify the location of the text as a parameter to the call. Essentially this is the same as calling the GpiMove function followed by the GpiCharString function. Listing 3.21 shows an example of these functions.

```
CHARBUNDLE charAttr;
POINTL point1 = {100,10};
POINTL point2 = {20,20};
/**** First draw plain old "Hello World" using GpiCharString  *****/
GpiMove(hps, &point1);
GpiCharString(hps, 11L, "Hello World");
/**** Now draw "Hello World" with flair using GpiCharStringAt *****/
/* First set up any special attributes - in this example we use   */
/* the char bundle to set the attributes. You could also use the  */
/* GPI helper functions to get the same effect.                   */
charAttr.lColor = CLR_GREEN;
/* Text color will be green */
charAttr.usSet = myOutlineFontId;
/* use a previously loaded outline font */
charAttr.usPrecision = CM_MODE3;
/* outline font draw mode   */
charAttr.ptlAngle.x = 1;
 /* draw at 45 deg angle     */
charAttr.ptlAngle.y = 1;
charAttr.ptlShear.x = 3;
```

LISTING 3.21 Using GpiCharString and GpiCharStringAt functions.

```
/* Shear by 33 degrees      */
charAttr.ptlShear.y = 1;
GpiSetAttrs(hps, PRIM_CHAR, CBB_COLOR | CBB_SET | CBB_MODE |CBB_ANGLE |
CBB_SHEAR, 0L, &charAttr);
/* all set up, now draw it */
GpiCharStringAt(hps, &point2, 12L, "Hello World!");
```

LISTING 3.21 (Continued).

The function called *GpiCharStringPos* is a more powerful version of the GpiCharString function. It lets you specify additional formatting options to be used when the text is drawn. With these options you can control things such as spacing the text, clipping the text, backdropping the text, underscoring the text, and overstriking the text. In addition, you can control whether the current position is updated to the end of the text or left in its original position. These options are chosen by logically ORing the values of the desired formatting flags together and then passing the result in for the *flOptions* parameter.

To control spacing of the text you must pass in an array of character increment values. These values are in world coordinates and specify the distance from the reference position of the current character to the reference position of the next character. If the text is drawn left-to-right or right-to-left, the distance is along the baseline; if it is drawn top-to-bottom or bottom-to-top, the distance is along the shearline. Figure 3.32 shows how character increment values are used to control spacing of text.

By determining the widths for each character in the text string, an array increment table can+* be constructed for the string. At this point, the array can be used to draw the text using the spacings appropriate for that font. With just a little more work, however, the text can also be kerned. By searching

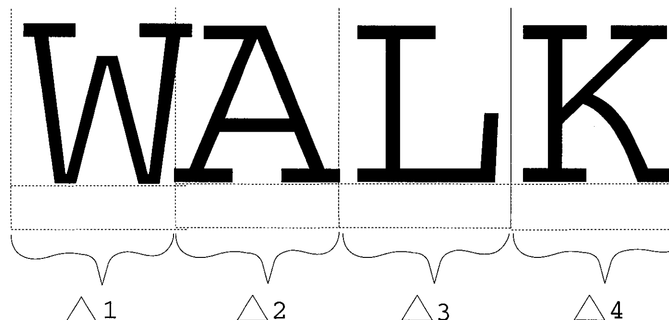


FIGURE 3.32 Controlling spacing using character increment values.

the text for kerning pairs, the increment values for those pairs can be adjusted to produce the overlapped effect. Kerning will be discussed in more detail in Chapter 4, "Fonts."

The function called *GpiCharStringPosAt* allows you to specify the location at which the formatted text is to be drawn. It provides the same formatting options as the GpiCharStringPos function. Listing 3.22 shows how these two functions are used.

```
POINTL start;
char String[132];
HPS hps;
LONG goTextWidths[132];
...
/*****************************************************/
/* Create the character spacing array for this string */
/*****************************************************/
SetWidthsTable(hps, goTextWidths, String);
/********************************************************/
/* Draw the string to the display using GpiCharStringPos */
/********************************************************/
GpiMove(hps, &start);
GpiCharStringPos(hps,NULL,CHS_VEC-
TOR,(LONG)strlen(String),String,goTextWidths);
/********************************************************/
/* Draw the same string using GpiCharStringPosAt instead */
/********************************************************/
GpiCharStringPosAt(hps,&start,NULL,CHS_VECTOR,(LONG)strlen(String),String,
                   goTextWidths);
```

LISTING 3.22 Using GpiCharStringPos and GpiCharStringPosAt functions.

See Chapter 4, "Fonts" for more detail on the SetWidthsTable function. Basically, this function gets the proper widths for each character in the string.

MARKER PRIMITIVES

Often when figures are drawn you will want to highlight or draw attention to particular items or points in the figure. The GPI has provided a set of graphic objects called *Marker Primitives* for this purpose. A marker is usually a symbol such as a dot, a cross, a diamond, a star, or other shape. These symbols are often used for charts (such as line or scatter charts) to highlight data points. The GPI comes with a predefined set of markers to choose from. In addition, you can define your own customized marker sets for more creative markers. Table 3.13 gives an overview of the marker functions and the operations they perform.

TABLE 3.13 Marker functions

Function	Description
GpiMarker	Draws a marker at a specified position.
GpiPolyMarker	Draws markers at positions specified by an array of points.

The attribute bundle for markers is specific for this category of drawing primitives. Setting of attributes in this bundle will affect how markers drawn using these operations are displayed. Table 3.14 describes each attribute in this bundle.

TABLE 3.14 Marker attribute bundle

Attribute Description	MARKER BUNDLE Field Name	flAttrMask Value	Default Value	Helper Function
Marker color	lColor	MBB_COLOR	CLR_BLACK	GpiSetColor*
Marker background color	lBackColor	MBB_BACK_COLOR	clear	GpiSetBackColor*
Marker mix	usMixMode	MBB_MIX_MODE	FM_OVERPAINT	GpiSetMix*
Marker background mix	usBackMixMode	MBB_BACK_MIX_MODE	FM_LEAVEALONE	GpiSetBackMix*
Marker set	usSet	MBB_SET	LCID_DEFAULT	GpiSetMarkerSet GpiQueryMarkerSet
Marker symbol	usSymbol	MBB_SYMBOL	MARKSYM_CROSS	GpiSetMarker GpiQueryMarker
Marker box	sizfxCell	MBB_BOX	device dependent; equal to size of 1 char	GpiSetMarkerBox GpiQueryMarkerBox

The *Marker Color* and *Marker Background Color* attributes control the foreground and background colors of the markers that are subsequently drawn.

The *Marker Mix* and *Marker Background Mix* attributes control how the foreground and background marker colors will be combined with the current contents of the drawing. Use the attribute called *Marker Set* to specify what set of marker symbols you wish to use. The default set of symbols can be

overidden by setting this attribute to the logical font identifier (lcid) of the font from which you wish to choose symbols.

The desired symbol within the current marker set can be chosen by setting the *Marker Symbol* attribute. Table 3.15 shows the symbols in the default marker set.

TABLE 3.15 Symbols in default marker set

Symbol Value	Definition
MARKSYM_CROSS	X sign
MARKSYM_PLUS	Plus sign
MARKSYM_DIAMOND	Diamond
MARKSYM_SQUARE	Square
MARKSYM_SIXPOINTSTAR	Six-point star
MARKSYM_EIGHTPOINTSTAR	Eight-point star
MARKSYM_SOLIDDIAMOND	Filled diamond
MARKSYM_SOLIDSQUARE	Filled square
MARKSYM_DOT	Filled circle
MARKSYM_SMALLCIRCLE	Hollow circle
MARKSYM_BLANK	Nothing
MARKSYM_DEFAULT	Blank

If the marker set has been overidden by a logical font lcid, you can specify the symbol in the font by setting the marker symbol attribute to the desired character. Listing 3.23 shows an example of this.

```
LONG myFontId;
POINTL markPt1 = {10,10};
POINTL markPt2 = {20,20};
/**** First an example of drawing a marker from the default marker set****/
/* In this example, we use the helper functions to choose the marker set */
/* and marker symbol attributes. You can also use the marker bundle to   */
/* set these same attributes.                                            */
GpiSetMarkerSet(hps, LCID_DEFAULT);
GpiSetMarker(hps, MARKSYM_EIGHTPOINTSTAR);
GpiMarker(hps, &markPt1);
/**** Next, choose the letter M as a marker from a font set ****/
GpiSetMarkerSet(hps, myFontId);
GpiSetMarker(hps, 'M');
GpiMarker(hps, &markPt2);
```

LISTING 3.23 Using a logical font character as a marker symbol.

The *Marker Box* attribute specifies the width and height of the marker box in world coordinates. If the marker is an outline font character, it will be scaled to fill the marker box. If it is a raster font character, the marker box is ignored.

Marker Functions

There are only two functions for actually drawing marker primitives. The function called *GpiMarker* is used to draw a single marker at a specified location. The position is specified in world coordinates and the marker will be centered around it. The editor tool uses the GpiMarker function to draw the *edit handles* of objects you can edit.

The second function is called *GpiPolyMarker*. This function draws a series of markers whose positions are specified by an array that is passed in to the function. If you were drawing a line chart, you would likely use the same array of points for both the GpiPolyLine function call and the GpiPolyMarker function call. Listing 3.24 shows an example of using the GpiPolyMarker function.

```
POINTL chartPts[5] = {{10,10}, {20,15}, {30,35}, {40,60}, {50,45} };
POINTL origin = {0,0};
/* Draw the line chart with a polyline and then hightlight the points   */
/* with markers.                                                         */
GpiMove(hps, &origin);
GpiPolyLine(hps, 5L, chartPts);
GpiSetMarkerSet(hps, LCID_DEFAULT);
GpiSetMarker(hps, MARKSYM_DIAMOND);
GpiPolyMarker(hps, 5L, chartPts);
```

LISTING 3.24 Using the GpiPolyMarker function.

Both of these functions will draw the markers according to the current attributes in the MARKERBUNDLE.

IMAGE PRIMITIVES

Some applications will want to display images. These images can come from a variety of sources such as a picture that has been electronically scanned and captured. The OS/2 GPI provides functions that allow applications to manipulate images that are stored in one of several *bitmap* formats. These bitmap images can be created and displayed on devices that support raster

operations. They cannot be displayed on plotters or other vector devices. Bitmaps are device dependent and therefore will not always look the same when displayed on different devices.

Images aside, bitmaps also have several other uses. Some of these uses include:

- Program icons (inside and outside the application).
- Program pointers.
- Rapid movement of a fixed picture across the screen.
- Animated pictures.

Table 3.16 gives an overview of the image functions and the operations they perform.

TABLE 3.16 Image functions

Function	Description
GpiCreateBitmap	Creates a bitmap resource.
GpiDeleteBitmap	Deletes a bitmap resource.
GpiSetBitmap	Sets a bitmap as current in a device context.
GpiSetBitmapDimensions	Sets the width and height of a bitmap.
GpiSetBitmapBits	Copies raw bitmap data into the bitmap resource.
GpiImage	Draws monochrome image from raw bitmap data.
GpiDrawBits	Draws color image from raw bitmap data.
GpiBitBlt	Draws color image from bitmap resource using device coordinates.
GpiWCBitBlt	Draws color image from bitmap resource using world coordinates.

The attribute bundle for images is specific for this catagory of drawing primitives. Setting of attributes in this bundle will affect how markers drawn using these operations are displayed. Table 3.17 describes each attribute in this bundle.

TABLE 3.17 Image attribute bundle

Attribute Description	IMAGE BUNDLE Field Name	flAttrMask Value	Default Value	Helper Function
Image color	lColor	IBB_COLOR	CLR_BLACK	GpiSetColor*
Image background color	lBackColor	IBB_BACK_COLOR	clear	GpiSetBackColor*
Image mix	usMixMode	IBB_MIX_MODE	FM_OVERPAINT	GpiSetMix*
Image background mix	usBackMixMode	IBB_BACK_MIX_MODE	FM_LEAVEALONE	GpiSetBackMix*

The *Image Color* and *Image Background Color* attributes control the foreground and background colors of the images that are subsequently drawn.

The *Image Mix* and *Image Background Mix* attributes control how the foreground and background colors of the images will be combined with the current contents of the drawing.

Depending on the image primitive function, color and mix attributes may be ignored. See the discussion on the function of interest for details.

Image Basics

A bitmap is basically a two-dimensional grid. Each cell in the grid represents a single point on the output device. These points are called picture elements or pels (sometimes called *pixels*). Each cell in the bitmap has a numeric value. A cell's value determines how the pel it represents will be displayed (i.e., what color it will have). Bitmaps can be either monochrome or color. Monochrome bitmaps require less storage than color bitmaps.

Monochrome bitmaps require only a single bit to represent each pel in the grid. If the bit value is one, then the pel is turned *on*. If the bit value is zero, the pel is turned *off*. Therefore, by simply turning on and off the appropriate cell bits in the bitmap, a picture can be constructed. Figure 3.33 shows an example of this.

Where:

1 – Black (on)
0 – White(off)

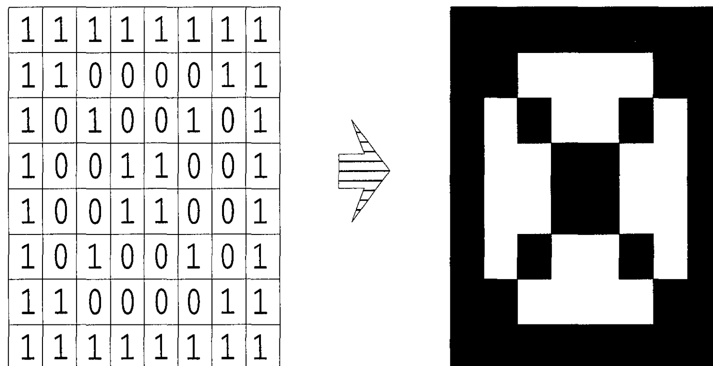


FIGURE 3.33 Mapping monochrome bitmap data to output pels.

Color bitmaps require multiple bits of storage to represent each pel. The number of colors that can be simultaneously used in a bitmap is 2^*n* where *n* is the number of bits used per pel. Therefore, if the bitmap has four bits per pel, you can display up to 2^4 or 16 simultaneous colors in the bitmap. Each 4-bit pel value specifies which of the 16 colors are to be used for that pel.

The value of each cell in a bitmap can be determined in two different ways. By organizing the bitmap into a single 'plane' of data, one or more consecutive bits are used to determine the color of the cell. For example, in a four-color bitmap, every cell would require two consecutive bits to determine the color of the cell. If it were an 8x8 bitmap, two bytes of information would be required for each row in the bitmap, since each byte has 8 bits and can thus hold 4 cells worth of color information. Actually, each row may require some additional padding, which we will discuss shortly.

The second way to represent color bitmap data is by using multiple planes of data. A cell has data in each plane that is combined to produce the overall color value for that cell. For example, the 8x8 four-color bitmap described would require two color planes. A row in each plane would require one byte of storage. The most common multiplane format uses three planes: one plane specifies the red value, one specifies the blue value, and one specifies the green value. These three values combine to produce the final color of the bitmap cell.

Theoretically, both formats would require the same amount of storage. The single plane format requires 2 bytes per scan line (row) and 8 rows for a total of 16 bytes. And the two plane format would require 1 byte per scan line (row) and 8 rows for a total of 16 bytes. The GPI, however, requires that the data for each scan line be aligned on a double word (i.e., 32-bit ULONG) boundary, inserting padding if necessary. This means that the single plane format will have an extra 2 bytes padding for each scan line which brings the total bitmap size to 32 bytes. The multiplane format gets hit even worse since it requires an extra 6 bytes padding for each scan line (3 bytes per scan line in each plane) which brings its total bitmap size to 64 bytes.

The single plane representation is the standard format used in OS/2 applications. In addition, as we will soon discuss, bitmaps have several standard bits/pel counts that must be supported by all devices. These counts are 1, 4, 8, and 24 bits/pel. Certain devices may support other bits/pel counts (like the 2 bits/pel we discussed above) but those are device dependent.

Figure 3.34 shows an example of a single plane color bitmap and how data is mapped to cells in the bitmap. This bitmap is a sixteen color bitmap and hence requires four bits to represent each cell's color value. Note, as was discussed above, each row in the bitmap data will actually be padded by an additional two bytes to get the proper alignment required by the GPI.

Since each cell is represented by four bits, it can have one of sixteen values. A color table is used to map each of the sixteen values into its real color. The number of entries in the color table will generally be 2^*n*, where *n* is the number of bits per bitmap cell.

4-Bit Values	Cell Values
0000 0000 0000 0000 0000 0000 0000 0000	0 0 0 0 0 0 0 0
0000 0001 0101 0101 0101 0101 0010 0000	0 1 5 5 5 5 2 0
0000 0100 0001 0101 0101 0010 0110 0000	0 4 1 5 5 2 6 0
0000 0100 0100 0001 0010 0110 0110 0000	0 4 4 1 2 6 6 0
0000 0100 0100 0010 0001 0110 0110 0000	0 4 4 2 1 6 6 0
0000 0100 0010 0011 0011 0001 0110 0000	0 4 2 3 3 1 6 0
0000 0010 0011 0011 0011 0011 0001 0000	0 2 3 3 3 3 1 0
0000 0000 0000 0000 0000 0000 0000 0000	0 0 0 0 0 0 0 0

FIGURE 3.34 Sixteen color bitmap data mapped into bitmap cell values.

Each entry in the color table defines the color for that entry as a 24-bit RGB value. An RGB value defines color as a combination of three colors (Red, Green, and Blue). The first 8 bits of an RGB value specify the red component, the next 8 bits specify the green component, and the last 8 bits specify the blue component. Each 8-bit value designates the intensity of the component (0-least, 255-most). By mixing various levels of the three colors, up to 16.7 million different color combinations can be produced. For example, suppose we wanted to map the bitmap cell values shown in Figure 3.34 into our own set of various colors. Table 3.18 shows some example RGB values for each entry in the color table.

TABLE 3.18 Color Table RGB values

Cell Value	RGB Value	Color Description
0	0xFF0000	Red
1	0x00FF00	Green
2	0x0000FF	Blue
3	0x000000	Black
4	0xFFFFFF	White
5	0xFFFF00	Yellow
6	0xFF00FF	Pink
7	0x00FFFF	Cyan

As mentioned earlier, different devices support various bits/cell bitmap formats. All devices must, however, support the standard set of bitmap formats shown Table 3.19.

TABLE 3.19 Standard bitmap formats

# Bits per Cell	# Colors Available
1	2
4	16
8	256
24	16.7 million

You can determine what other types of bitmap formats are supported by an output device by using the function called *GpiQueryDeviceBitmapFormats*.

For bitmaps that require less than 24 bits of color, the cell values are used to index into a color table to determine the 24-bit RGB value associated with that cell. Bitmaps of this nature use a structure called BITMAPINFO2 to describe the color table and other general bitmap information such as dimensions of the bitmap, the number of bits per cell, the number of color planes used, and so on. When using this structure, you need to declare your variable as a pointer to this structure and then allocate enough storage to hold both the structure and any color table entries that are required.

Bitmaps that require a full 24 bits of color are treated a little differently. Since each cell provides 24 bits, there is no need for a color table. Instead, each cell value can be used directly as an RGB value. Because of this, 24-bit bitmaps do not supply the color table portion of the BITMAPINFO2 structure.

Bitmaps are constructed using the function called *GpiCreateBitmap*. As input, this function accepts BITMAPINFOHEADER2 structure that will define the bitmap to be created. This structure is the same as the BITMAPINFO2 structure but does not contain a color table. As an option, the GpiCreateBitmap function will let you specify the initial bitmap data. If you choose to do this, you must supply the initial data and, since the initial data may be in a different form than the bitmap you are creating, you must also supply a BITMAPINFO2 structure that defines the initial data.

Before creating a bitmap, you must first create a memory device context and an associated presentation space. The memory device context lets you store and manipulate bitmaps that are destined for an output device. As such, you should create the device context to be compatible with the device that the bitmap will be eventually displayed on. For example, if you will be putting the bitmaps to the display, create a memory device context (using DevOpenDC) that is compatible with the display. If no device is specified, the display is assumed.

Once the memory device context and memory presentation space are available, you can create bitmaps into the memory presentation space. Before manipulating the bitmap or copying it to another presentation space, you must set it as the current bitmap in the memory presentation space. This is done using the function called *GpiSetBitmap*.

Listing 3.25 shows how a simple four-color bitmap is defined and created.

```
HDC hdcMemory;                  // Window device context handle.
HPS hps, hpsMemory;          // Presentation space handle for client area.
HBITMAP newBmp;
BITMAPINFOHEADER2 newBmpFormat;
BITMAPINFO2 *initDataFormat;
BYTE initData[] = {
   0x00, 0x00, 0x00, 0x00,
   0x01, 0x55, 0x55, 0x20,
   0x04, 0x15, 0x52, 0x60,
   0x04, 0x41, 0x26, 0x60,
   0x04, 0x42, 0x16, 0x60,
   0x04, 0x23, 0x31, 0x60,
   0x02, 0x33, 0x33, 0x10,
   0x00, 0x00, 0x00, 0x00
};
int fmtSize;
POINTL boundaryInfo[4];
 ...
/* Create a memory device context and presentation space that      */
/* we can create the bitmap in. Later we can copy it to the        */
/* normal display presentation space where it will become visible. */
hdcMemory = DevOpenDC(hab, OD_MEMORY, "*", 4, (PDEVOPENDATA) pszData,   NUL
L);
sizl.cx=PAGEXSIZE;          // Create Normal-PS.  Keep as global.
sizl.cy=PAGEYSIZE;
hpsMemory=GpiCreatePS(hab, hdcMemory, &sizl, PU_PELS | GPIT_MICRO |
   GPIA_ASSOC);
...
/* Define the target bitmap format */
memset(&newBmpFormat, 0, sizeof(newBmpFormat));
newBmpFormat.cbFix = sizeof(newBmpFormat);
newBmpFormat.cx = 8;
newBmpFormat.cy = 8;
newBmpFormat.cPlanes = 1;
newBmpFormat.cBitCount = 4;
newBmpFormat.cclrUsed = 16;

/* Allocate the BITMAPINFO2 structure large enough to hold 16 color table
entries */
fmtSize = sizeof(BITMAPINFO2) + (16*sizeof(RGB2));
initDataFormat = malloc(fmtSize);
memset(initDataFormat, 0, fmtSize);
/* define the initial data format and the color table
 */
initDataFormat->cbFix = sizeof(BITMAPINFOHEADER2);
initDataFormat->cx = 8;             /* Bitmap is 8x8 pels             */
initDataFormat->cy = 8;
initDataFormat->cPlanes = 1;        /* one plane of data              */
initDataFormat->cBitCount = 4;      /* Two bits per pel in each plane   */
```

LISTING 3.25 Creation of sixteen-color bitmap.

```
initDataFormat->cbImage = sizeof(initData); /* Total size of bitmap data */
initDataFormat->cclrUsed = 16;        /* Number of color indexes used     */
initDataFormat->argbColor[0].bRed   = 0xFF;  /* Index 0 is Red              */
initDataFormat->argbColor[0].bGreen = 0x00;
initDataFormat->argbColor[0].bBlue  = 0x00;
initDataFormat->argbColor[1].bRed   = 0x00;     /* Index 1 is Green      */
initDataFormat->argbColor[1].bGreen = 0xFF;
initDataFormat->argbColor[1].bBlue  = 0x00;
initDataFormat->argbColor[2].bRed   = 0x00;     /* Index 2 is Blue       */
initDataFormat->argbColor[2].bGreen = 0x00;
initDataFormat->argbColor[2].bBlue  = 0xFF;
initDataFormat->argbColor[3].bRed   = 0x00;     /* Index 3 is Black      */
initDataFormat->argbColor[3].bGreen = 0x00;
initDataFormat->argbColor[3].bBlue  = 0x00;
initDataFormat->argbColor[4].bRed   = 0xFF;  /* Index 4 is White         */
initDataFormat->argbColor[4].bGreen = 0xFF;
initDataFormat->argbColor[4].bBlue  = 0xFF;
initDataFormat->argbColor[5].bRed   = 0xFF;  /* Index 5 is Yellow        */
initDataFormat->argbColor[5].bGreen = 0xFF;
initDataFormat->argbColor[5].bBlue  = 0x00;
initDataFormat->argbColor[6].bRed   = 0xFF;  /* Index 6 is Pink          */
initDataFormat->argbColor[6].bGreen = 0x00;
initDataFormat->argbColor[6].bBlue  = 0xFF;
initDataFormat->argbColor[7].bRed   = 0x00;     /* Index 7 is Cyan       */
initDataFormat->argbColor[7].bGreen = 0xFF;
initDataFormat->argbColor[7].bBlue  = 0xFF; /* All others are Black      */
/* Create the bitmap using the initial data & make it the current bitmap in
the ps */
newBmp = GpiCreateBitmap(hpsMemory, &newBmpFormat, CBM_INIT, initData,
   initDataFormat);
GpiSetBitmap(hpsMemory,newBmp);
// Alternatively, we could have created the bitmap and initialized it using
// the GpiSetBitmapBits function as follows:
// newBmp = GpiCreateBitmap(hpsMemory, &newBmpFormat, 0L, NULL, NULL);
// GpiSetBitmap(hpsMemory, newBmp);
// GpiSetBitmapBits(hpsMemory, 0L, 8L, initData, initDataFormat);
/* Copy the loaded bitmap into the display */
boundaryInfo[0].x = 0; /* target destination (in device coords) */
boundaryInfo[0].y = 100;
boundaryInfo[1].x = 50;
boundaryInfo[1].y = 150;
boundaryInfo[2].x = 0; /* source rectangle (pels in src color bitmap) */
boundaryInfo[2].y = 0;
boundaryInfo[3].x = 8;
boundaryInfo[3].y = 8;
GpiBitBlt(hps, hpsMemory, 4L, boundaryInfo, ROP_SRCCOPY, BBO_IGNORE);
```

LISTING 3.25 (Continued).

The 24-bit format is special in that no color table is required; instead, the 24-bit value represents an RGB value (8 bits for Red, 8 bits for Green, and 8 bits for Blue). The RGB value is translated directly into a color. In this case, the bitmap is defined using the `BITMAPINFOHEADER2` structure instead of the `BITMAPINFO2` structure. The `BITMAPINFOHEADER2` structure is ba-

sically the same as the BITMAPINFO2 structure except that it does not have the color table in it. The color table is not needed since the bitmap data will contain the RGB values (there is no need to go through an intermediate table).

GpiCreateBitmap can also be used to create bitmaps using the BITMAPINFOHEADER2 structure. If using this structure, you cannot specify initial bitmap data on the GpiCreateBitmap call.

You can determine what types of bitmap formats are supported by your output device by using the function called *GpiQueryDeviceBitmapFormats*. Bitmaps can be destroyed using the function called *GpiDeleteBitmap*.

Image Functions

As you've seen, bitmaps can be created and initialized using the function GpiCreateBitmap. There are several other ways of initializing the state of a bitmap that can be quite useful. These include drawing into a bitmap, loading a bitmap from a file, or copying from one bitmap to another. There are a variety of GPI functions available for these types of bitmap operations and more.

Another way to alter a bitmap after it has been created is by using the functions called *GpiSetBitmapDimension* and *GpiSetBitmapBits*. As the name implies, GpiSetBitmapDimension lets you set the width and height of the bitmap (specified in .1 millimeter increments). These dimensions are *not* used by the GPI and have no impact on bitmap operations. They can, however, be retrieved by the application using the function *GpiGetBitmapDimensions*. GpiSetBitmapBits is used to copy all or part of the defining bitmap data into a bitmap.

Bitmaps can be loaded from a file using the function *GpiLoadBitmap*. You might use this function if you've created some bitmaps using the OS/2 Icon Editor, for example. This function is similar to GpiCreateBitmap except the data is initialized from a bitmap file rather than passed in as an array in memory. The bitmap can be stretched or compressed, if desired, when it is loaded. Listing 3.26 shows an example of loading a bitmap resource from the applications EXE file.

To create the contents of a bitmap by drawing into it, you must first set the bitmap as the current bitmap in a memory device context. This can be done using the function called *GpiSetBitmap*. Once this is done, the application can perform normal drawing operations and the results will be drawn

```
.RC file contains:
   ...
   BITMAP   IDB_MYPIC
mypic.bmp.C source file contains:
   HDC hdcMemory;            // Window device context handle.
   HPS hps, hpsMemory;       // Presentation space handle for client area.
   HBITMAP myBmp;
   HPS hps;
   LONG myBmpId=1;
   ...
   /* Create a memory device context and presentation space that         */
   /* we can create the bitmap in. Later we can copy it to the           */
   /* normal display presentation space where it will become visible.    */
   hdcMemory = DevOpenDC(hab, OD_MEMORY, "*", 4, (PDEVOPENDATA)
      pszData, NULL);
   sizl.cx=PAGEXSIZE;          // Create Normal-PS.  Keep as global.
   sizl.cy=PAGEYSIZE;
   hpsMemory=GpiCreatePS(hab, hdcMemory, &sizl, PU_PELS | GPIT_MICRO |
      GPIA_ASSOC);
   ...
   /* Create the bitmap, loading it out of the .exe file */
   myBmp = GpiLoadBitmap(hps, NULL, IDB_MYPIC, 50L, 50L);
/* Assign a local identifier to the bitmap so it can be used for fills */
   GpiSetBitmapId(hps, myBmp, myBmpId);
   /* Select the new bitmap as current for the memory ps */
   GpiSetBitmap(hpsMemory,myBmp);
   /* Copy the loaded bitmap into the display */
   boundaryInfo[0].x = 0; /* target destination (in device coords) */
   boundaryInfo[0].y = 200;
   boundaryInfo[1].x = 50;
   boundaryInfo[1].y = 250;
   boundaryInfo[2].x = 0; /* source rectangle (pels in src color bitmap) */
   boundaryInfo[2].y = 0;
   boundaryInfo[3].x = 50;
   boundaryInfo[3].y = 50;
   GpiBitBlt(hps, hpsMemory, 4L, boundaryInfo, ROP_SRCCOPY, BBO_IGNORE);
```

LISTING 3.26 Loading a bitmap resource from a file.

into the bitmap (providing they fall within the bitmap boundaries). Listing 3.27 shows an example of drawing into a bitmap to define its contents.

```
HBITMAP newBmp, prevBmp;
BITMAPINFOHEADER2 newBmpFormat;
POINTL points[9] = {
   {40,15}, {70,15}, {70,35}, {50,35}, {50,75},
   {70,75}, {60,95}, {40,95}, {40,15}
};
POINTL boundaryInfo[4];
...
/* Create a memory device context and presentation space that      */
/* we can create the bitmap in. Later we can copy it to the        */
```

LISTING 3.27 Defining contents of a bitmap resource by drawing into it.

```
/* normal display presentation space where it will become visible. */
hdcMemory = DevOpenDC(hab, OD_MEMORY, "*", 4, (PDEVOPENDATA) pszData,
            NULL);
sizl.cx=PAGEXSIZE;          // Create Normal-PS.  Keep as global.
sizl.cy=PAGEYSIZE;
hpsMemory=GpiCreatePS(hab, hdcMemory, &sizl, PU_PELS | GPIT_MICRO |
            GPIA_ASSOC);
...
/* Define the target bitmap format */
memset(&newBmpFormat, 0, sizeof(newBmpFormat));
newBmpFormat.cbFix = sizeof(newBmpFormat);
newBmpFormat.cx = 100;                  /* size of the bitmap */
newBmpFormat.cy = 100;
newBmpFormat.cPlanes = 1;
newBmpFormat.cBitCount = 4;             /* std 16 color format */
newBmpFormat.cclrUsed = 16;
/* Create the bitmap using no initial data */
newBmp = GpiCreateBitmap(hpsMemory, &newBmpFormat, 0L, NULL, NULL);
/* make it the currently selected bitmap in the memory device context */
GpiSetBitmap(hpsMemory, newBmp);
/* Now draw into the bitmap */
GpiSetColor(hpsMemory, CLR_WHITE);
GpiMove(hpsMemory, &points[0]);
GpiPolyLine(hpsMemory, 8L, &points[1]); /* Copy the loaded bitmap into the
display*/
boundaryInfo[0].x = 0; /* target destination (in device coords) */
boundaryInfo[0].y = 300;
boundaryInfo[1].x = 50;
boundaryInfo[1].y = 350;
boundaryInfo[2].x = 0; /* source rectangle (pels in src color bitmap) */
boundaryInfo[2].y = 0;
boundaryInfo[3].x = 100;
boundaryInfo[3].y = 100;
GpiBitBlt(hps, hpsMemory, 4L, boundaryInfo, ROP_SRCCOPY, BBO_IGNORE);
```

LISTING 3.27 (Continued).

Now that we've talked about creating bitmaps, let's see how you can display them. The GPI again comes to the rescue with a variety of functions for doing this. Some functions perform very basic draw capabilities while others are more flexible.

The function called *GpiImage* is useful only for drawing monochrome images. It accepts raw bitmap data and therefore does not require you to create a bitmap resource as discussed above. The bitmap will be drawn using the current image foreground and background colors. A value of 1 uses the foreground color while a value of 0 uses the background color. The current image foreground and background mix attributes are used as well. The current position will be the top-left corner of the image. The image size is specified in pels and is therefore device dependent. Output from this function can not be directed into retained segment storage. While most bitmap drawing

functions interpret scanline data from bottom to top, GpiImage interprets it from top to bottom.

Three GPI functions exist that work with color bitmaps (*GpiDrawBits*, *GpiBitBlt*, and *GpiWCBitBlt*). These functions all operate as though there is a *source* bitmap that you are copying from and a *target* bitmap that you are copying to. The source is usually the bitmap that you wish to draw, and the target is either pels that are output to the device or another bitmap that you want to copy into.

Because there are both source and target bitmaps, these three functions must, at times, be able to convert from monochrome bitmaps to color bitmaps, and vice versa. The conversion is necessary, for example, when copying a monochrome bitmap into a color bitmap or drawing a color bitmap to a monochrome device. When that conversion is needed, the image foreground and background colors are used. When converting from monochrome to color, bits in the bitmap that are set to 1 are mapped to the foreground color, while bits in the bitmap that are set to 0 are mapped to the background color. When converting from a color bitmap to a monochrome bitmap, those bits whose color matches the background color are mapped to the value 0, while all other bits are mapped to the value 1.

When copying the source bitmap to the target bitmap, a mix option is also specified. As with color mixing, this mix determines how the source bitmap is to be combined with the existing target bitmap. A normal copy operation, for example, would specify the option ROP_SRCCOPY. This mix option specifies that the source bits are blindly copied directly over the top of the current target bitmap (ignoring its current value). The option ROP_SRCAND, however, would perform a logical AND operation of the source bits with the target bits and then update the target bits with the resulting values. Table 3.20 shows the constants defined for some of the more common mix operations.

Note that mix operations can also include the current pattern set and pattern symbol. This gives you even more flexibility for mixing bitmaps. The constants shown in Table 3.20 are only a subset of the potential mix combinations. In reality, you can produce just about any combination of source, target, and pattern symbol that you want. Typical uses for mix include straight copy of source over target (ROP_SRCCOPY), XOR dynamic drawing of source on top of target(ROP_SRCINVERT), drawing source behind

target(ROP_SRCERASE), and drawing source over target with a pattern effect (ROP_MERGECOPY).

TABLE 3.20 Common bitmap mix options

Mix Value	Logical Description
ROP_SRCCOPY	target=source
ROP_SRCPAINT	target=source OR target
ROP_SRCAND	target=source AND target
ROP_SRCINVERT	target=source XOR target
ROP_SRCERASE	target=source AND (NOT target)
ROP_NOTSRCCOPY	target=NOT source
ROP_NOTSRCERASE	target=(NOT source) AND (NOT target)
ROP_MERGECOPY	target=source AND pattern
ROP_MERGEPAINT	target=(NOT source) OR target
ROP_PATCOPY	target=pattern
ROP_PATPAINT	target=(NOT source) OR pattern OR target
ROP_PATINVERT	target=target XOR pattern
ROP_DSTINVERT	target=NOT target
ROP_ZERO	target=0
ROP_ONE	target=1

Before getting into the color bitmap functions, one last item to address concerns mapping between source and target bitmaps when the sizes do not match. In this situation, the GPI compresses or stretches the source bitmap to fit into the target bitmap space. When stretching a bitmap, the GPI inserts duplicate rows or columns at regular intervals to produce the expansion effect. When compressing the source bitmap, the GPI combines adjacent rows or columns at regular intervals to produce the shrinking effect. When combining the adjacent rows, the GPI allows you to specify one of three combination methods, as follows:

1. BBO_OR
2. BBO_AND
3. BBO_IGNORE

The option BBO_OR combines adjacent rows (or columns) by logically ORing the values of the adjacent rows together. The option BBO_AND com-

bines adjacent rows by logically ANDing the values of the adjacent rows together. These two options are most useful for compressing monochrome or two color bitmaps (`BBO_OR` when foreground pels are 1 and `BBO_AND` when foreground pels are 0). The option `BBO_IGNORE` compresses by simply removing rows (or columns) at regular intervals. No merging with adjacent rows is performed. This is most useful for color bitmaps since pel colors would be corrupted by merging them with adjacent pels.

Okay, now that we've talked about the factors that go into displaying and converting color bitmaps, let's examine the functions available for doing these things.

The function called *GpiDrawBits* is similar to GpiImage. It, too, lets you specify raw bitmap data, but this data is to be interpreted as color bitmap data. You need to supply a `BITMAPINFO2` structure to describe the color bitmap layout. The *aptlPoints* parameter specifies the bounding rectangles of the source and target bitmaps. The source rectangle specifies the portion of the source bitmap to draw (specified in device coordinates—pels), while the target rectangle specifies the area of the current device context that the source is to be drawn into (specified in world coordinates and mapped to device coordinates using current transforms). GpiDrawBits performs any conversions required for mismatches in color or size. This function will also route the output to a retain segment if the draw mode is set appropriately.

About now you may be asking yourself "why did we go to the trouble of creating a bitmap resource if these drawing routines can accept the raw data directly?" Well, as you'll soon see, the last two drawing functions we will talk about make use of the bitmap resources we described earlier.

You can draw a bitmap resource directly to device coordinates using the function called *GpiBitBlt*. This function accepts a target rectangle specified in device coordinates and copies the data from the source bitmap resource into the target rectangle. This function will stretch or compress the data as needed to make it fit the target rectangle. In addition, colors will be converted from the source bitmap to the output device as previously described. Output of this function will not go to retained segment storage.

You can draw a bitmap resource into world coordinates using the function called *GpiWCBitBlt*. Using this function, the bitmap will be subject to all the normal viewing transformations and will be routed to retained storage. The GpiWCBitBlt function provides the same compression and mixing options as the GpiBitBlt function.

Listing 3.28 shows some examples of using the various bitmap functions discussed so far.

```
/* Monochrome image of the letter T in non-standard form. */
BYTE monoImageData[] = {
   0xFF, 0xFF,     /* 1111111111111111 */
   0xC1, 0x83,     /* 1100000110000011 */
   0x01, 0x80,     /* 0000000110000000 */
   0x01, 0x80,     /* 0000000110000000 */
   0x01, 0x80,     /* 0000000110000000 */
   0x01, 0x80,     /* 0000000110000000 */
   0x01, 0x80,     /* 0000000110000000 */
   0x01, 0x80,     /* 0000000110000000 */
   0x07, 0xE0      /* 0000011111100000 */
};
/* Monochrome image of the same letter T only this time the data */
/* is in std bitmap form. Notice that image data is reversed and */
/* that it is padded to 32 bits per scan line.                   */
BYTE monoImageData2[] = {
   0x07, 0xE0, 0x00, 0x00,     /* 0000011111100000 */
   0x01, 0x80, 0x00, 0x00,     /* 0000000110000000 */
   0x01, 0x80, 0x00, 0x00,     /* 0000000110000000 */
   0x01, 0x80, 0x00, 0x00,     /* 0000000110000000 */
   0x01, 0x80, 0x00, 0x00,     /* 0000000110000000 */
   0x01, 0x80, 0x00, 0x00,     /* 0000000110000000 */
   0x01, 0x80, 0x00, 0x00,     /* 0000000110000000 */
   0xC1, 0x83, 0x00, 0x00,     /* 1100000110000011 */
   0xFF, 0xFF, 0x00, 0x00,     /* 1111111111111111 */
};
SIZEL size;
POINTL monoImagePos = {50,50};
BITMAPINFO2 monoFormat;
BITMAPINFOHEADER2 newBmpFormat;
POINTL boundaryInfo[4];
HBITMAP myBmp, anotherBmp;
PSZ pszData[4] = {"Display", NULL, NULL, NULL };
...
/* Create a memory device context and presentation space that      */
/* we can create the bitmap in. Later we can copy it to the        */
/* normal display presentation space where it will become visible. */
hdcMemory = DevOpenDC(hab, OD_MEMORY, "*", 4, (PDEVOPENDATA) pszData, NULL);
sizl.cx=PAGEXSIZE;          // Create Normal-PS.  Keep as global.
sizl.cy=PAGEYSIZE;
hpsMemory=GpiCreatePS(hab, hdcMemory, &sizl, PU_PELS | GPIT_MICRO |
   GPIA_ASSOC);
...
/***************************************************************/
/* Draw a monochrome bitmap to the output device using GpiImage */
/***************************************************************/
GpiSetColor(hps, CLR_BLUE);
GpiSetBackColor(hps, CLR_YELLOW);
```

LISTING 3.28 Using GpiImage, GpiDrawBits, GpiBitBlt, and GpiWCBitBlt functions.

```
GpiMove(hps, &monoImagePos);
size.cx = 16;
size.cy = 9;
GpiImage(hps, 0L, &size, sizeof(monoImageData), monoImageData);
...
/*************************************************************/
/* Draw the std version of the mono image using GpiDrawBits.    */
/* Stretch the bits during the draw to make it taller.          */
/*************************************************************/
/* Define the format of the mono image data */
/* By setting the size of the Fixed area to 16 bytes, you can tell the   */

/* GPI to default the remaining entries in the BITMAPINFO2 structure.    */

/* Otherwise you can define a full size BITMAPINFO2 header complete with */
/* color table as we did in listing 3.24.                                */
memset(&monoFormat, 0, sizeof(monoFormat));/* Set image format parameters
*/
monoFormat.cbFix = 16L /*sizeof(monoFormat)*/;
monoFormat.cx = 16;                          /* size of the bitmap
*/
monoFormat.cy = 9;
monoFormat.cPlanes = 1;
monoFormat.cBitCount = 1;                    /* 2-color format
*/
/* Define the old/new sizes, make narrower and taller */
boundaryInfo[0].x = 1; /* target destination (in world coords)
*/
boundaryInfo[0].y = 1;
boundaryInfo[1].x = 25;
boundaryInfo[1].y = 50;
boundaryInfo[2].x = 0;/* source rectangle (pels in the monochrome bitmap)
*/
boundaryInfo[2].y = 0;
boundaryInfo[3].x = 16;
boundaryInfo[3].y = 9;
GpiDrawBits(hps, monoImageData2, &monoFormat, 4L, boundaryInfo,
ROP_SRCCOPY,
           BBO_IGNORE);
...
/********************************************************************/
/* Load a color bitmap from a file, then, copy it onto the screen      */
/* using GpiBitBlt. Draw it over the top of the current screen contents */
/* by using ROP_SRCINVERT. Compress it using BBO_IGNORE since it is a   */
/* color bitmap.                                                        */
/********************************************************************/
/* Create the source bitmap, loading it out of the .exe file */
myBmp = GpiLoadBitmap(hpsMemory, 0L, IDB_MYPIC, 50L, 50L);
/* make it the currently selected bitmap in the memory device context */
GpiSetBitmap(hpsMemory, myBmp);
/* Copy the loaded bitmap into the display */
boundaryInfo[0].x = 200; /* target destination (in device coords) */
```

LISTING 3.28 (Continued).

```
boundaryInfo[0].y = 1;
boundaryInfo[1].x = 250;
boundaryInfo[1].y = 25;
boundaryInfo[2].x = 1; /* source rectangle (pels in src color bitmap) */
boundaryInfo[2].y = 1;
boundaryInfo[3].x = 50;
boundaryInfo[3].y = 50;
GpiBitBlt(hps, hpsMemory, 4L, boundaryInfo, ROP_SRCINVERT, BBO_IGNORE);
...
/*****************************************************************/
/* Finally, load another bitmap into the display device context      */
/* and copy it into world coordinates using the GpiWCBitBlt function. */
/*****************************************************************/
/* Create the source bitmap, loading it out of the .exe file         */
/* Note the bitmap must be loaded into the same device context as it  */
/* will be BitBlt'ed to using GpiWCBitBlt.                            */
anotherBmp = GpiLoadBitmap(hps, 0L, IDB_MYPIC2, 100L, 100L);
/* Copy the loaded bitmap into the display */
boundaryInfo[0].x = 250; /* target destination (in world coords) */
boundaryInfo[0].y = 0;
boundaryInfo[1].x = 300;
boundaryInfo[1].y = 50;
boundaryInfo[2].x = 1; /* source rectangle (pels in src color bitmap) */
boundaryInfo[2].y = 1;
boundaryInfo[3].x = 100;
boundaryInfo[3].y = 100;
GpiWCBitBlt(hps, anotherBmp, 4L, boundaryInfo, ROP_SRCCOPY, BBO_IGNORE);
```

LISTING 3.28 (Continued).

Another interesting way to use bitmaps is as fill patterns for closed areas. This can be done using the function called *GpiSetBitmapId.* This function creates a local identifier that is associated with a bitmap resource you created using GpiCreateBitmap or GpiLoadBitmap. This local identifier can then be used as a pattern set for filling closed areas. The pattern set will contain a single member. Listing 3.29 shows how GpiSetBitmapId is used to set up a fill pattern of geometric shapes.

```
BYTE monoImageData2[] = {
   0xFF, 0x00, 0x00, 0x00,    /* 1 1 1 1 1 1 1 1 */
   0xC0, 0x00, 0x00, 0x00,    /* 1 1 0 0 0 0 0 0 */
   0x60, 0x00, 0x00, 0x00,    /* 0 1 1 0 0 0 0 0 */
   0x30, 0x00, 0x00, 0x00,    /* 0 0 1 1 0 0 0 0 */
   0x18, 0x00, 0x00, 0x00,    /* 0 0 0 1 1 0 0 0 */
   0x0C, 0x00, 0x00, 0x00,    /* 0 0 0 0 1 1 0 0 */
   0x06, 0x00, 0x00, 0x00,    /* 0 0 0 0 0 1 1 0 */
   0x03, 0x00, 0x00, 0x00,    /* 0 0 0 0 0 0 1 1 */
   0xFF, 0x00, 0x00, 0x00,    /* 1 1 1 1 1 1 1 1 */
};
BITMAPINFO2 monoFormat;
```

LISTING 3.29 Using GpiSetBitmapId to create fill patterns.

```
BITMAPINFOHEADER2 newBmpFormat;
HBITMAP myBmp;
LONG myBmpId=10;
POINTL point1 = {300, 0};
POINTL point2 = {500, 200};
...
/****************************************************************/
/* Create a bitmap that contains the desired pattern.          */
/****************************************************************/
/* Define the target bitmap format */
memset(&newBmpFormat, 0, sizeof(newBmpFormat));
newBmpFormat.cbFix = sizeof(newBmpFormat);
newBmpFormat.cx = 8;
newBmpFormat.cy = 8;
newBmpFormat.cPlanes = 1;
newBmpFormat.cBitCount = 1;
newBmpFormat.cclrUsed = 2;
/* Define the format of the mono image data */
/* By setting the size of the Fixed area to 16 bytes, you can tell     */
/* the GPI to default the remaining entries in the BITMAPINFO2         */
/* structure.  Otherwise you can define a full size BITMAPINFO2 header */
/* complete with color table as we did in listing 3.24.                */
memset(&monoFormat, 0, sizeof(monoFormat));/* Set image format parameters
*/
monoFormat.cbFix = 16L /*sizeof(monoFormat)*/;
monoFormat.cx = 8;                         /* size of the bitmap
*/
monoFormat.cy = 8;
monoFormat.cPlanes = 1;
monoFormat.cBitCount = 1;                  /* 2-color format
*/
/* Create the bitmap using the initial data */
myBmp = GpiCreateBitmap(hps, &newBmpFormat, CBM_INIT, monoImageData2,
          &monoFormat);
/* Assign a local identifier to the bitmap */
/* so it can be used for a pattern fill    */
GpiSetBitmapId(hps, myBmp, myBmpId);
/* Set the pattern set to use the bitmap */
GpiSetPatternSet(hps, myBmpId);
/* Now draw a rectangle and it will be filled */
/* with the bitmap fill pattern               */
GpiMove(hps, &point1);
GpiBox(hps, DRO_OUTLINEFILL, &point2, 0L, 0L);
```

LISTING 3.29 (Continued).

In this section we've talked mainly about creating and drawing entire images; however, the GPI does provide a function for setting just a single pel. This function, called *GpiSetPel*, sets a pel (at the specified point in world coordinates) using the current line color and mix attributes. This function is subject to normal transformation and clipping operations. The function called *GpiQueryPel* can be used to examine the color of a single pel (again specified as a point in world coordinates).

As you can see, the GPI provides a large set of graphics primitives. Using them, you can create a wide variety of drawings and special effects. In this chapter we discussed drawing attributes such as color and that there are bundles of attributes for each category of drawing primitives.

In addition we have examined the attributes and drawing primitives in each of the following categories:

- Line and Arc primitives
- Area primitives
- Text primitives
- Marker primitives
- Image primitives

These drawing primitives give you quite a rich set of functions to choose from in your application programming. When combining them with other GPI features such as segments and transformations, you can draw just about anything! In the next chapter, you will learn more details about fonts, including how to design and use your own.

CHAPTER 4
Fonts

It wasn't that long ago that most programmers didn't really care that much about fonts. This was because most programmers weren't developing applications in a graphical environment like OS/2's Presentation Manager! Instead, many were writing applications in an environment like DOS, and the font used was whatever the display adapter hardware produced in text mode. Formatting text was pretty straightforward, too. This was because all the characters for the display had the same spacing; hence, simple row/column calculations were all that was needed. Now, because graphical user interfaces are much more common, it becomes important for software developers to understand and take advantage of the variety of fonts available. By doing this, the applications they develop can become much more appealing and usable, and with application software becoming increasingly compe-tive, all aspects of the application (such as font selection) should be carefully considered to help insure the success of your product.

As you will soon see, working with fonts in OS/2 is much different than in an environment like DOS, and there is little doubt that the complexity of the code is greater. The results, however, are also much more spectacular and rewarding than in a non-graphical environment.

Before we talk about the functions that the OS/2 GPI has to help you work with fonts, some basic background information is necessary. Fonts are applied to a table called a *code page*. If you were to look at code page defini-

tions, you would notice that it has several entries which contain different symbols. These symbols usually include letters, numbers, and special graphic characters. Each of these symbols or images in a code page is called a *glyph*. Each entry in the code page is called a *code point*. Hence, a code point is an index into a code page to identify a glyph. So, for instance, if you were to look at an ASCII code page, you would notice code point X'31' would have a glyph for the character "1" defined in it.

Code pages are identified by a number and are published so the symbol sets can be known to all hardware and software developers. For instance, the standard United States code page number for ASCII is 437, while for EBCDIC the code page number is 37. Now if you were to look at these two different code pages, you would see the same basic glyphs are found in both of them, however, the glyphs are found at different code points. For instance, the glyph for the character "1" which was located at code point X'31' for the ASCII code page is at code point X"F1" for the EBCDIC code page. You might think that if you know the ASCII 437 code page and EBCDIC 37 code page that you have code pages licked. In some parts of the world, this may be true, but as you look at the code pages available, you will notice that there are several ASCII and EBCDIC code pages used by different countries and these different code pages have some different glyphs defined in them. Furthermore, some code pages may not have any alphanumeric data in them at all! Instead, these code pages could be other symbol sets that you may use for some other reason. Table 4.1 shows the code pages currently supported by OS/2.

Fonts are placed in groups that share the same basic characteristics of the font design. These groups are called *font family names*. Examples of font family names are Courier™, Helvetica™, and Times New Roman™. Within a font family, a specific font will have a *facename*. For example, within the Courier font family there are specific fonts such as Courier, Courier Bold, Courier Bold Italic, Courier Italic, and so on. The primary unit of measure for specifying a font's size is called a *point*. A point is 1/72 of an inch and refers to the visible height of the font.

So what's so difficult about fonts? Well, fonts really aren't that hard to deal with, but the metrics associated with a font can be much more complex than just point size. When you look at all the metrics that can be assoicated with a font (as we will shortly), you may even begin to wonder how point size is calculated! After all, the characters within a font can all have different

visible heights. And characters like g, j, p, q, and y can even drop below the imaginary horizontal line used to determine the character's vertical position. This imaginary line is called the font's *baseline*. In fact, the more you look at fonts and all the metrics associated with them, you may determine that designing and developing a font is more of an art than a science!

TABLE 4.1 OS/2 supported code pages

Code Page Number	Name
37	EBCDIC US ENGLISH/PORTUGUESE
273	EBCDIC AUSTRIAN/GERMAN
277	EBCDIC DANISH/NORWEGIAN
278	EBCDIC FINNISH/SWEDISH
280	EBCDIC ITALIAN
284	EBCDIC SPANISH
285	EBCDIC UK ENGLISH
297	EBCDIC FRENCH
437	ASCII US ENGLISH
500	EBCDIC BELGIAN/INTERNATIONAL
850	ASCII LATIN 1 MULTILINGUAL
851	ASCII LATIN 2 MULTILINGUAL
857	ASCII TURKEY
860	ASCII PORTUGUESE
861	ASCII ICELAND
863	ASCII CANADIAN-FRENCH
865	ASCII NORWEGIAN
1004	ASCII DESKTOP PUBLISHING
1026	EBCDIC TURKEY

So what other metrics do you need to be concerned with when it comes to fonts? Well, another obvious metric is the width of each character within a font. As you probably already know, fonts are either *proportional* or *non-proportional*. If a font is non-proportional (also called monospaced), then all the characters within the font have the same width. Hence, calculating the horizontal starting location for each character in a string or the total width of a string is fairly straightforward with monospaced fonts. If the font is proportional, however, the characters within the font can all have different

widths. Hence, to format text correctly for a proportional font can take more calculation and seem more tedious. As you will soon see, however, the GPI provides functions that make the chore of positioning characters within a string fairly easy.

With this brief discussion of font widths you might think you have a fairly good understanding of the kinds of things you will need to consider when generating text. And you probably do! But there is one more topic that needs to be discussed when it comes to the positioning adjacent characters in a string. This topic is called *kerning*. Some fonts are designed so some of their character cells can actually overlap and still produce a very pleasing appearence. For example, in a character string where the characters "V" and "A" are adjacent, the top of the "V" can hang over the bottom of the "A" and everything will still look great. When a font is designed so adjacent characters can overlap, the font is called *kernable*. Figure 4.1 illustrates this point more clearly by showing adjacent characters where kerning is used. Note that when a font is designed to be kerned, however, only particular pairs of characters within the font are kernable. For instance, the character pair "AV" can be kerned but the character pair "AA" can not because "AA" can not overlap. Hence, when a font is kernable, all character pairs in a string must be interrogated so the appropriate adjustments in character placement can be made! Again, this may seem tedious but you will see that the GPI also helps with the kerning of the font is designed to be kerned!

Another key point about fonts in OS/2 is how they are physically generated during drawing. There are two basic ways in which fonts can be generated and the method used depends on the type of font. If the font is a *raster* or

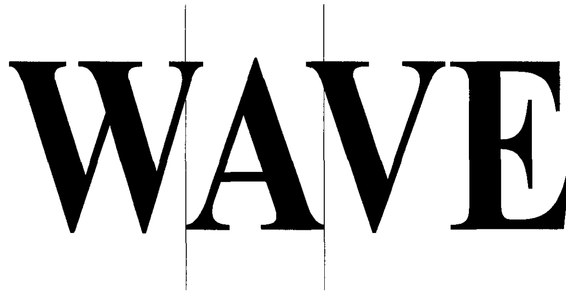


FIGURE 4.1 Kerned font.

image font, its picture elements for each character are predefined and are copied directly to the drawing. Because a raster font is a predefined image, the actual draw performance of image fonts is excellent. On the other hand, because using raster fonts is a matter of an image copy, raster fonts are device dependent and can not be scaled or rotated. Of course, given a good selection of raster fonts, an application that needs simple text operations or excellent draw performance will find raster fonts to be ideal. (And OS/2 has a good selection of image fonts.)

The other type of font is called an *outline font* (sometimes called a *vector font*). An outline font is not a set of images for the character set, but rather a series of draw orders such as lines and curves that generate the characters in the font. Hence, each character in a font is actually drawn, not copied. Because the characters are drawn, they can be scaled, rotated, sheared, and so on, just like any other object in our graphic editor. (By the way, our graphic editor only uses outline fonts for this reason! All transforms for text objects work just like all other objects in the graphic editor!) Of course, because each character of an outline font needs to be drawn, the performance of an outline font is not as good as that of an image font.

One last thing to discuss before we get in to the details of working with fonts is where fonts are located. In OS/2, a font is either a *public font*, a *private font*, or a *device font*. Public fonts are available to all applications in the system and can be used on multiple devices. When you install OS/2, you are prompted to select the fonts you want loaded on your system. These fonts are the public fonts that come with OS/2. Currently, IBM provides the public outline fonts shown in Table 4.2. You can, however, receive additional fonts from other sources and install them on your system and make them public fonts too.

In addition to the public outline fonts listed in Table 4.2, OS/2 also provides several raster fonts. These raster fonts come in a variety of point sizes that are designed for a variety of device resolutions. The face names that are available for these raster fonts are Courier, System Proportional, System Monospaced, Helv, and Tms Rmn.

An application can also have fonts loaded dynamically as it needs them. These fonts are not implicitly known to the other applications on the system and are called private fonts. Private fonts may be fonts that you developed

TABLE 4.2 OS/2 Public Outline Fonts

Font Family	Font Face Name
Courier	Courier
	Courier Bold
	Courier Bold Italic
	Courier Italic
Helvetica	Helvetica
	Helvetica Bold
	Helvetica Bold Italic
	Helvetica Italic
Times New Roman	Times New Roman
	Times New Roman Bold
	Times New Roman Bold Italic
	Times New Roman Italic
Symbol	Symbol

just for your application or fonts that you received from some other source and do not want to share with other applications.

Finally, devices like printers often have fonts designed especially for their use as part of their device support. These fonts are called device fonts and are available only to applications that are using the device. Device fonts are valid only while you have an associated device context with the device. Therefore, device fonts have a more restrictive use than public fonts. In particular, you can not use the system spooler for printing jobs if device fonts are involved. Instead, you must print using raw mode. To understand the difference between printing in raw mode and printing via the OS/2 spooler, see Chapter 8, "Printing."

Well, now that you've been introduced to the basic concepts of fonts in OS/2, let's look at fonts in a little closer detail and see the functions that the GPI has to help you work with them!

FONT METRICS

As stated earlier, a presentation space saves non-device-specific and environmental drawing information. One such piece of information saved in the presentation space is the current font. Hence, when you create a presentation space, the presentation space will automatically have a default font assigned to it. This default font is typically the system monospace font. When your application wants a different font with which to display information, it may want to query the system for all (or some) the fonts that are available so an intelligent selection can be made. To query the system for fonts, your application can use a function called *GpiQueryFonts*. Following are the parameters for the GpiQueryFont function:

hps – This is the handle for your presentation space.

flOptions – This is an unsigned long value that specifies the type of fonts for which the query will be performed. This parameter has some predefined constants that can be ORed together. These predefined constants and their meanings are as follows:

`QF_PUBLIC` – query public fonts.
`QF_PRIVATE` – query private fonts.
`QF_NO_DEVICE` – do not list device fonts.
`QF_NO_GENERIC` – do not list generic fonts (non-device fonts).

pszFacename – This is a pointer to an ASCIIZ string that has the facename of the font you want to query. Note that several fonts can have the same facename but different metrics. If NULL is used for this parameter, then all facenames will be queried.

plReqFonts – This is a pointer to a long value and is both an input and output parameter. As an input parameter, you can specify the maximum number of fonts for which you want font metrics returned. As an output parameter, this specifies how many fonts had their font metrics returned.

lMetricsLength – This is a long value that specifies the amount of space reserved for each font metric on return. Note that the font metric structure is relatively large but constructed in such a way that the most popular metrics are listed toward the beginning of the structure. This structure will be shown shortly.

afmMetrics – This is a pointer to the space where the array of font metics is to be returned. This space should be at least as large as (plReq-Fonts x lMetricsLength) bytes. The structure of the metrics returned will be shown shortly.

On return from the GpiQueryFonts function, a long value is returned which indicates the number of fonts that matched the search criteria but did not return font metric data. As you can see, the GpiQueryFonts function will let you query for a particular facename and font type, but before you limit your search or allocate space for the font metric data, you may like to know how many of the different fonts for a particular type are even available. To do this, set up your search criteria (flOptions and pszFacename) and specify 0 for plReqFonts. The value returned by the GpiQueryFonts function is the number of fonts that the system found that met the search criteria.

But, what is all of this potential font metric data that can be returned by the GpiQueryFonts function, and what is it good for? Well, as we look at the structure used to return font metric data, it becomes more obvious what some of the information is useful for. However, much of the information is not so obviously useful and you may choose to ignore it. The font metric data is returned in a structure called FONTMETRICS. Listing 4.1 shows the definition of the FONTMETRICS structure followed by a definition of each field within the structure. As you read these metric definitions, note that definitions will occasionally refer to field definitions within the same FONTMETRICS structure. As you read the definitions of the fields for the FONTMETRICS structure, look at Figure 4.2. Figure 4.2 will show you many of these font metrics in relationship to some actual sample characters within a font.

As you can see, there is more than just a little bit of information stored in the FONTMETRICS structure. But don't be intimidated by its size! When you think about the information that is being communicated to you and how you might choose to use it, it's really pretty cool. Read on and think of the possibilities!

```
typedef struct _FONTMETRICS {
CHAR  szFamilyname[32];
CHAR  szFacename[32];
USHORT   usRegistry;
USHORT   usCodePage;
```

LISTING 4.1 Font metrics definition.

```
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   usType;
USHORT   usDefn;
USHORT   usSelection;
USHORT   usCapabilities;
LONG  lSubscriptXSize;
LONG  lSubscriptYSize;
LONG  lSubscriptXOffset;
LONG  lSubscriptYOffset;
LONG  lSuperscriptXSize;
LONG  lSuperscriptYSize;
LONG  lSuperscriptXOffset;
LONG  lSuperscriptYOffset;
LONG  lUnderscoreSize;
LONG  lUnderscorePosition;
LONG  lStrikeoutSize;
LONG  lStikeoutPosition;
SHORT sKerningPairs;
SHORT sFamilyClass;
LONG  lMatch;
ATOM  FamilyNameAtom;
ATOM  FaceNameAtom;
PANOSE   panPanose;
} FONTMETRICS;
```

LISTING 4.1 (Continued).

szFamilyname – This field is a null-terminated character string which is the family name for the font. The family name helps identify the general appear-

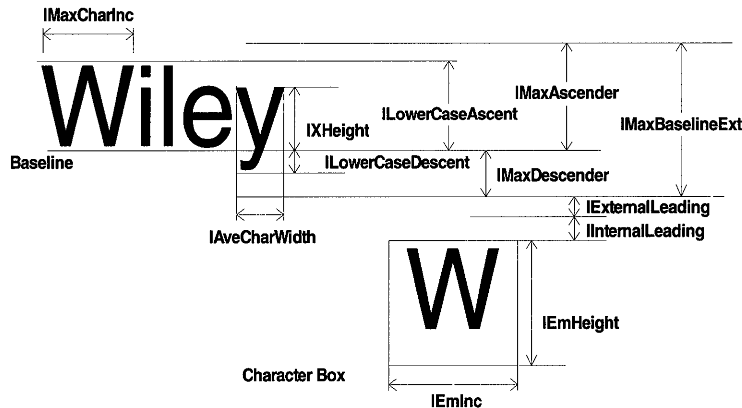


FIGURE 4.2 Font metrics.

ance of the font. For example, Courier is a family name. This string is limited to 31 characters plus a 0 to terminate the string. If the family name is larger than 31 characters, you can use the FamilyNameAtom value to retrieve the entire family name string.

szFacename – This field is a null-terminated character string, which is the facename for the font. The facename identifies a particular font within a font family. For example, Courier Bold Italic is a facename in the Courier family. This string is limited to 31 characters plus a 0 to terminate the string. If the facename is larger than 31 characters, you can use the FaceNameAtom value to retrieve the entire facename string.

usRegistry – This field is the registry identifier for the font.

usCodePage – This field is the registered code page for which the font was designed. Often, this field is 0, which means the font can be used with any of the OS/2 supported code pages. Hence, if this field is 0, you can specify the code page you want when you create a logical font. (Logical font creation is discussed later.) When you create a logical font, the font character to code point mappings will be made for you. If the font contains special symbols which have no register code page, a value of 65400 is returned in this field. In this case, you must use the returned code page value during logical font creation.

lEmHeight – This field is the height of the largest character in the font. Historically, this was considered to be the height of the character "M", hence the name EmHeight. But, as it turns out, this is not always true! This metric is returned in world coordinates and is also used for calculating point size.

lXHeight – This field is the nominal height of lowercase characters above the baseline. This metric ignores ascenders and is returned in world coordinates.

lMaxAscender – This field is the maximum height above the baseline for any symbol found in the font. This metric is returned in world coordinates.

lMaxDescender – This field is the maximum depth below the baseline for any symbol found in the font. This metric is returned in world coordinates.

lLowerCaseAscent – This field is the maximum height above the baseline for any lowercase character (a–z) symbol found in the font. This metric is returned in world coordinates.

lLowerCaseDescent – This field is the maximum depth below the baseline for any lowercase character (a–z) symbol found in the font. This metric is returned in world coordinates.

lInternalLeading – This field is the difference between the lEmHeight and the lMaxBaseExt. Therefore, this metric is a measure of distance between the highest symbol in the font and the highest alphabetic character in the font. This metric is returned in world coordinates.

lExternalLeading – This field is the maximum recommended space added between rows of text so the text will have a pleasing appearance. This metric is only the font designers recommended white space between rows and is often 0. This metric is returned in world coordinates.

lAveCharWidth – This field represents the average character width for the font. The value is determined by weighing each of the lowercase characters found in the font by its expected frequency of use before the average is calculated. This metric is returned in world coordinates.

lMaxCharInc – This field represents the maximum width of a character for the font. This metric is returned in world coordinates.

lEmInc – This field represents the width of an imaginary box called the *Em Square*. The Em Square is also the *Character Box* attribute for outline fonts.

If the vertical and horizontal device resolutions for a device are equal, then lEmInc is equal to lEmHeight. Therefore, because outline fonts are device independent, lEmInc and lEmHeight are equal for outline fonts. This metric is returned in world coordinates.

lMaxBaseExt – This field represents the maximum vertical space required for the font. This is the sum of the maximum ascender and the maximum descender. This metric is returned in world coordinates.

sCharSlope – This field represents nominal slope for the font. This metric is the number of degrees and minutes clockwise from vertical that the font may slant. For example, an italic font may have a small positive slope while a normal (non-italic) font would have a value of 0. The way degrees and minutes are represented in a SHORT value is by the high order seven bits of the SHORT representing minutes and the low order nine bits of the SHORT representing degrees. For example, X'3F04' represents 260 degrees, 30 minutes.

sInLineDir – This field represents the direction in which the glyphs in the font were designed to be viewed. The unit of measure for this metric is in degrees and minutes and stored in the same way as described for sCharSlope. This measurement is taken clockwise from horizontal. When a code point is used to reference a glyph for the font, it is added to the line of text in the inline direction.

sCharRot – This field represents the rotation of the character glyphs in the font with respect to the baseline. The unit of measure for this metric is in degrees and minutes and stored in the same way as described for sCharSlope. This measurement increases in the counterclockwise direction.

usWeightClass – This field represents the thickness of the stroke used for generating the glyphs of the font. Table 4.3 shows the values that can be found in this field and its meaning.

usWidthClass – This field represents the aspect ratio for the characters in the font compared to a normal aspect ratio for a font of this type. Table 4.4 shows the values that can be found in this field and its meaning.

sXDeviceRes – This field represents the horizontal device resolution. For bitmap fonts this is measured in pels per inch for the intended output device. For outline fonts this is measured in notional units for the width of the Em Square. (Note that outline fonts are defined in a space above world space

called the notional font definition space. The notional font definition space dimensions are typically 1000x1000.)

TABLE 4.3 Fonts weight class values

Value	Meaning
1	Ultra-light
2	Extra-light
3	Light
4	Semi-light
5	Medium
6	Semi-bold
7	Bold
8	Extra-bold
9	Ultra-bold

TABLE 4.4 Fonts width class values

Value	Meaning
1	50% of normal width
2	62.5% of normal width
3	75% of normal width
4	87.5% of normal width
5	normal width
6	112.5% of normal width
7	125% of normal width
8	150% of normal width
9	200% of normal width

sYDeviceRes – This field represents the vertical device resolution. For bit-map fonts this is measured in pels per inch for the intended output device. For outline fonts this is measured in notional units for the height of the Em Square. (Note that outline fonts are defined in a space above world space called the *notional font definition space*. The notional font definition space dimensions are typically 1000x1000.)

sFirstChar – This field represents the code point for the first character in the font.

sLastChar – This field represents the code point for the last character in the font. This number, however, is an offset from the sFirstChar value. For example, if sFirstChar were 20 and sLastChar were 80, then the last code point value is 100. All code points in the range from sFirstChar to sLastChar are supported by the font.

sDefaultChar – This field represents the code point within the font that is used if a code point outside of the supported font range is needed. Like sLastChar, this value is an offset of the sFirstChar value.

sBreakChar – This field represents the code point that represents the "break" character or "space" character for the font. Like sLastChar, this value is an offset of the sFirstChar value.

sNominalPointSize – This field represents the nominal point size for the font. For raster fonts, this value is the height of the font. For outline fonts, however, this value is the intended height of the font as determined by the font designer. The unit of measure for this metric is decipoints or 1/720 of an inch.

sMinimumPointSize – This field represents the minimum intended height of the font as determined by the designer of the outline font. Because raster fonts are static, this field has no value to them. Like sNominalPointSize, the unit of measure for this metric is decipoints or 1/720 of an inch.

sMaximumPointSize – This field represents the maximum intended height of the font as determined by the designer of the outline font. Because bitmap fonts are static, this field has no value to them. Like sNominalPointSize, the unit of measure for this metric is decipoints or 1/720 of an inch.

usType – This field contains a series of bit indicators that communicate type information about the font. The OS/2 Toolkit provides definitions for the different type indicators. Following is a list of these definitions and their meanings:

`FM_TYPE_FIXED` – All characters in the font have the same width.
`FM_TYPE_LICENSED` – The font is licensed or protected.
`FM_TYPE_64K` – The font is larger than 64KB in size.
`FM_TYPE_DBCS` – The font is for double-byte code pages.

FM_TYPE_MBCS – The font is for mixed single/double-byte code pages.
FM_TYPE_FACETRUNC – The font szFaceName field is truncated. Therefore, you may want to use the System Atom Table to retrieve the entire name.
FM_TYPE_FAMTRUNC – The font szFamilyName field is truncated. Therefore, you may want to use the System Atom Table to retrieve the entire name.
FM_TYPE_ATOMS – The font atom values for szFamilyName and szFaceName are valid.

usDefn – This field contains a series of bit indicators that communicate definition information about the font. The OS/2 Toolkit provides definitions for the different definition indicators. Following is a list of these definitions and their meanings:

FM_DEFN_OUTLINE – The font is an outline font. If this is not true, then the font is a font.
FM_DEFN_GENERIC – The font is in a format that can be used by the GPI. If this is not true, then the font is a device font.

usSelection – This field contains a series of bit indicators that communicate style information about the font. Note that the GPI can simulate styles but the styles indicated here are part of the physical font design. The OS/2 Toolkit provides definitions for the different selection indicators. Following is a list of these definitions and their meanings:

FM_SEL_ITALIC – The font is italic.
FM_SEL_UNDERSCORE – The font has underscore for each character.
FM_SEL_NEGATIVE – The font has its background and foreground reversed.
FM_SEL_OUTLINE – The font characters are hollow.
FM_SEL_STRIKEOUT – The font has an overstrike in each character.
FM_SEL_BOLD – The font is bold.

usCapabilities – This field contains a bit indicator that communicates font capabilities. Note that this field only applies to device fonts and the indicator is located in the low order byte of this USHORT field. The OS/2 Toolkit provides a definition for this indicator named

FM_CAP_NOMIX. If this indicator is true, then font characters cannot be mixed with graphics for the device. The high order byte of this USHORT may also contain a value that indicates the font's print quality. Following is a list of the quality values and their meanings:

0 – Undefined
1 – DP quality
2 – DP draft
3 – Near letter quality
4 – Letter quality

lSubscriptXSize – This field contains the recommended horizontal size for subscripts for this font. This metric is returned in world coordinate units.

lSubscriptYSize – This field contains the recommended vertical size for subscripts for this font. This metric is returned in world coordinate units.

lSubscriptXOffset – This field contains the recommended horizontal offset for subscripts for this font. This metric is returned in world coordinate units.

lSubscriptYOffset – This field contains the recommended vertical offset from the baseline for subscripts for this font. In this case, a positive value means below the baseline. This metric is returned in world coordinate units.

lSuperscriptXSize – This field contains the recommended horizontal size for subscripts for this font. This metric is returned in world coordinate units.

lSuperscriptYSize – This field contains the recommended vertical size for subscripts for this font. This metric is returned in world coordinate units.

lSuperscriptXOffset – This field contains the recommended horizontal offset for subscripts for this font. This metric is returned in world coordinate units.

lSuperscriptYOffset – This field contains the recommended vertical offset from the baseline for subscripts for this font. This metric is returned in world coordinate units.

lUnderscoreSize – This field contains the thickness of the underscore for this font. In the case of a font that has FM_SEL_UNDERSCORE set to true, this is the actual thickness of the underscore. If FM_SEL_UNDERSCORE is false, then this is the simulated underscore thickness. This metric is returned in world coordinate units.

lUnderscorePosition – This field contains the position of the underscore from the baseline for this font. A positive value for this metric means below the baseline. In the case of a font that has FM_SEL_UNDERSCORE set to true, this is the actual position of the underscore. If FM_SEL_UNDERSCORE is false, then this is the simulated underscore position. This metric is returned in world coordinate units.

lStrikeoutSize – This field contains the thickness of the strikeout stroke for this font. In the case of a font that has FM_SEL_STRIKEOUT set to true, this is the actual thickness of the strikeout stroke. If FM_SEL_STRIKEOUT is false, then this is the simulated strikeout stroke thickness. This metric is returned in world coordinate units.

lStrikeoutPosition – This field contains the position of the strikeout stroke from the baseline for this font. In the case of a font that has FM_SEL_STRIKEOUT set to true, this is the actual position of the strikeout stroke. If FM_SEL_STRIKEOUT is false, then this is the simulated strikeout stroke position. This metric is returned in world coordinate units.

sKerningPairs – This field contains the number of kerning pair values for this font.

sFamilyClass – This field contains a font family design classification.

lMatch – This field contains a value that uniquely identifies the font for a device driver to device combination. If this value is less than 0, the font is a device font. If this value is greater than 0, the font is generic and can be used by the GPI. Note that this value may vary from system to system and should not be used to identify fonts across system boundaries.

FamilyNameAtom – This field contains the atom identifier for retrieving the full font family name character string from the System Atom Table. This identifier is valid if FM_TYPE_ATOMS is true.

FaceNameAtom – This field contains the atom identifier for retrieving the full font face name character string from the System Atom Table. This identifier is valid if FM_TYPE_ATOMS is true.

panPanose – This field contains the Panose descriptor which identifies the visual characteristics of the font.

Once you have font metric data for some or all of the existing fonts on the system, you may then want to select one of these fonts and associate it with

your presentation space. To associate a font to a presentation space, you must first use a function called *GpiCreateLogFont* to provide a logical definition for the font you desire. The input parameters to the GpiCreateLogFont function are as follows:

hps – This parameter specifies the presentation space handle for which this logical font description is being provided.

pName – This parameter specifies a pointer to an eight-character string. This string can be used to help identify the required font.

lLcid – This parameter specifies a local identifier to which the logical font description will be assigned. Once assigned, this local identifier can be used to identify the logical font description for this presentation space until either the local identifier has been reassigned or deleted. (Your presentation space can have up to 256 local identifiers that can relate meaningful information to it at any time.)

pAttr – This parameter specifies a pointer to a font attributes structure which defines the requirements of the logical font.

As you may have guessed, the key parameter for identifying a font via the GpiCreateLogFont function is the pointer to a font attributes structure which you must provide. When you look at Listing 4.2, which shows the definition of the font attributes structure, you should recognize many of the field names. This is because many of these field names are the same or similar to those found in the font metrics data structure. Therefore, filling in the font attributes structure is fairly simple. When you want to select a particular font for which you have font metrics data, you can just copy some of the

```
typedef struct _FATTRS {
USHORT   usRecordLength;
USHORT   fsSelection
LONG  lMatch
CHAR  szFacename[32];
USHORT   idRegistry;
USHORT   usCodePage;
LONG  lMaxBaselineExt
LONG  lAveCharWidth;
USHORT   fsType;
USHORT   fsFontUse;
} FATTRS;
```

LISTING 4.2 Font attributes structure.

required information from the font metrics structure to the font attributes structure and then query and set similar values for the other fields that have similar but not equal meanings.

Note that the fsSelection, fsType, and fsFontUse fields in the font attributes structure have similar meanings to the usSelection, fsType, and usDefn fields of the font metrics structure. The font attribute fields, however, do have different defined values than those for the similar font metrics fields. Following is a list of these particular font attribute field names and their meanings:

fsSelection – This field contains a series of bit indicators that communicate style information that is to be simulated by the GPI if possible. The OS/2 Toolkit provides definitions for the different selection indicators. Following is a list of these definitions and their meanings:

FATTR_SEL_ITALIC – The GPI will simulate an italic font.

FATTR_SEL_UNDERSCORE – The GPI will simulate an underscore for each character.

FATTR_SEL_OUTLINE – The GPI will use an outline font with hollow characters.

FATTR_SEL_STRIKEOUT – The GPI will simulate an overstrike in each character.

FATTR_SEL_BOLD – The GPI will simulate a bold font.

fsType – This field contains a series of bit indicators that communicate type information about the desired font. The OS/2 Toolkit provides definitions for the different type indicators. Following is a list of these definitions and their meanings:

FATTR_TYPE_KERNING – The font supports kerning.

FATTR_TYPE_DBCS – The font is for double-byte code pages.

FATTR_TYPE_MBCS – The font is for mixed single/double-byte code pages.

FATTR_TYPE_ANTIALIASED – The font supports antialiasing. (This is a device-specific type of font.)

fsFontUse – This field contains a series of bit indicators that communicate how the font will be used. The OS/2 Toolkit provides definitions for the different definition indicators. Following is a list of these definitions and their meanings:

FATTR_FONTUSE_OUTLINE – The font to be used is an outline font.

FATTR_FONTUSE_NOMIX – The font will not be mixed with graphic objects.
FATTR_FONTUSE_TRANSFORMABLE – The font may be scaled, sheared, or rotated.

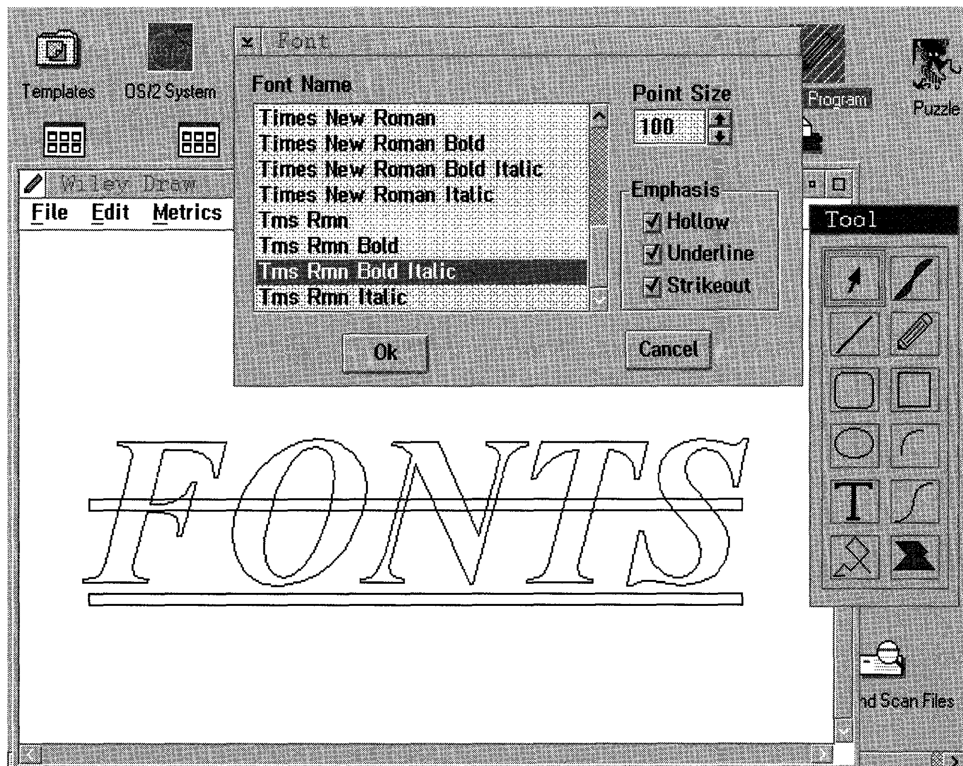
As it turns out, you don't have to fill in very many fields of the font attributes structure to have the GpiCreateLogFont function assign a logical font description to a local identifier. This is because the GpiCreateLogFont function will attempt to find a best fit font description from the information provided! And in the case of a local workstation for which you already have a unique lMatch identifier, all you need to do is copy the lMatch identifier. Using only an lMatch identifier in the font attributes structure to select a specific font, however, is not recommended. This is because lMatch identifiers are limited in scope. The lMatch identifier can only be assured to be valid on the machine on which it is defined for the duration of the *intial program load* (IPL). Hence, a lMatch identifier may not be meaningful or give you the same font description on other remote machines. If you fill in all the fields of the font attributes structure, however, the GPI will have a much better chance of providing an excellent match for a logical font description based on the additional font attribute information (search criteria). Therefore, another alternative to querying all the font metric data and then filling in the font attributes for a particular font, is to fill in the basic font attributes you desire, set lMatch to 0, and let the GpiCreateLogFont function find the best match for you!

After you have a local identifier for your logical font description, you can then use it to change the current font for the presentation space. This done with a function called *GpiSetCharSet*, which has the presentation space handle and the local identifier as input parameters. Hence, to change the current font of a presentation space to a font description you already have an local identifier assigned to, you just use GpiSetCharSet. Once you have issued the GpiSetCharSet function successfully, any draw text operations will use the new font. (Note that if you specify a local indentifier of 0 for the GpiSetCharSet function, the system default font will be used.)

If your new font is an outline font, you may wish to change its default size. Recall from the font metric data that a font has an imaginary box called the Em Square which has a height of lEmHeight and a width of lEmInc. The Em Square dimensions are also the dimensions of the character box which is one of the text attributes for the presentation space. Also recall from the font

metric data that the sXDeviceRes and sYDeviceRes values for outline fonts are the dimensions of the notional font definition space. The transformation from the notional font definition space to world coordinate space is defined by the ratio of the sXDeviceRes,sYDeviceRes rectangle and the Em Square. Hence, for outline fonts, changing the character box attribute will change the size of the font as it moves from notional font definition space to world coordinate space. The GPI function that allows you to change the character box attribute for the presentation space is called *GpiSetCharBox.*

To determine how to change the character box attribute to scale an outline font's size, you must first query the device characteristics for effective font height in device unit-per-unit distance. The OS/2 function called DevQueryCaps can be used to find this effective font height and is returned as pels per inch. Once you have this effective font height metric, you can use it to calculate the height of the character box to fit your need in device units,



SCREEN 4.1 Graphic editor font selection dialog.

then convert it to world space units, and finally convert the numbers to FIXED where they can be applied to the GpiSetCharBox function. Time for an example to see how all this font information can be used!

The graphic editor for this book provides a dialog that will allow the user to select a font for use during the edit session. Screen 4.1 shows an example of the types of things you can do with fonts with the graphic editor, and the dialog box we use to allow you to pick fonts. As mentioned earlier, the graphic editor only uses outline fonts; hence, the only fonts that will show up in the font selection dialog box list box are outline fonts known to the system. You can also see from Screen 4.1 that we allow the user to pick the font point size (which can range from 1 to 300) and to specify special font attributes like underline, strike out, and hollow.

By this time you should almost be able to guess how we wrote this dialog and selected the font, but let's look at Listing 4.3 to see how we actually did it.

```
/*********************************/
/* Select font dialog procedure. */
/*********************************/
MRESULT EXPENTRY SelectFontDlgProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2){
   extern HWND hwndClient;
   PFONTMETRICS fontMetrics=NULL;
   LONG numFonts,reqFonts,i,remFonts=GPI_ALTERROR;
   HDC hdc=0L;
   FATTRS fatAttrs;
   CHAR outlineFont[40];
   extern CHAR currentFont[40];
   extern HPS hps;
   USHORT item, fontIndex;
   MRESULT mReturn;
   ULONG pointSize;
   USHORT emp;
   extern USHORT emphasis;
   extern SIZEF charBox;
   extern ATTRIBS currentAttribs;
   LONG lcid;
   USHORT keyFlags;
   switch(msg){
      /*********************/
      /* Initialize dialog. */
      /*********************/
      case WM_IITDLG:
         /*********************************************************/
         /* Query number of public fonts known by PM and allocate */
         /* memory in which to read their metrics information.    */
         /*********************************************************/
```

LISTING 4.3 Select font routines.

```
reqFonts=0;
numFonts=GpiQueryFonts(hps,QF_PUBLIC,NULL,&reqFonts,
   0L, NULL);
/*********************************************/
/* Get memory for font metrics if any exist. */
/*********************************************/
if((numFonts!=GPI_ALTERROR) && (numFonts!=0L))
   fontMetrics=malloc((SHORT)(numFonts*sizeof(FONTMETRICS)));
/***************************************/
/* Query font information into array. */
/***************************************/
remFonts=GpiQueryFonts(hps,QF_PUBLIC,NULL,
   &numFonts,(LONG)sizeof(FONTMETRICS),fontMetrics);
/**********************************************************/
/* Search for outline font and put facename in list box. */
/**********************************************************/
 if(remFonts!=GPI_ALTERROR){
   for(i=0;i<(INT)numFonts;i++){
       if((fontMetrics[i].fsDefn&FM_DEFN_OUTLINE) &&
         (fontMetrics[i].usCodePage!=65400) ){
         WinSendDlgItemMsg(hwnd,ID_FONTNAME,LM_INSERTITEM,
            MPFROM2SHORT(LIT_SORTASCENDING,0),
            (MPARAM)fontMetrics[i].szFacename);
      }
   }
}
if(fontMetrics!=NULL)free(fontMetrics); // Free memory from array.
/********************************/
/* Find index of current font. */
/********************************/
fontIndex=(USHORT)(WinSendDlgItemMsg
   (hwnd,ID_FONTNAME,LM_SEARCHSTRING,
   MPFROM2SHORT(LSS_CASESENSITIVE,LIT_FIRST),
      (MPARAM)currentFont));
if((fontIndex==LIT_ERROR) || (fontIndex==LIT_NONE)){
   /*************************************/
   /* Highlight first facename in list. */
   /*************************************/
   WinSendDlgItemMsg(hwnd,ID_FONTNAME,LM_SELECTITEM,
      MPFROMSHORT(0),(MPARAM)TRUE);
}
else{
   /***************************************/
   /* Highlight current facename in list. */
   /***************************************/
   WinSendDlgItemMsg(hwnd,ID_FONTNAME,LM_SELECTITEM,
      MPFROMSHORT(fontIndex),(MPARAM)TRUE);
}
/*********************************************/
/* Set spin button limits and initial value. */
/*********************************************/
pointSize=*((ULONG *)AttGet(currentAttribs,ATT_FONTSIZE));
WinSendMsg(WinWindowFromID(hwnd,ID_POINTSIZE),
```

LISTING 4.3 (Continued).

```
            SPBM_SETLIMITS,(MPARAM)300,(MPARAM)1);
        WinSendMsg(WinWindowFromID(hwnd,ID_POINTSIZE),
            SPBM_SETCURRENTVALUE,(MPARAM)pointSize,(MPARAM)NULL);
        /************************/
        /* Set emphasis buttons. */
        /************************/
        if(emphasis & FATTR_SEL_OUTLINE)
            WinSendMsg(WinWindowFromID(hwnd,ID_OUTLINE),
                BM_SETCHECK,MPFROMSHORT(1),(MPARAM)NULL);
        if(emphasis & FATTR_SEL_STRIKEOUT)
            WinSendMsg(WinWindowFromID(hwnd,ID_STRIKEOUT),
                BM_SETCHECK,MPFROMSHORT(1),(MPARAM)NULL);
        if(emphasis & FATTR_SEL_UNDERSCORE)
            WinSendMsg(WinWindowFromID(hwnd,ID_UNDERSCORE),
                BM_SETCHECK,MPFROMSHORT(1),(MPARAM)NULL);
        /*******************************/
        /* Set input focus to list box. */
        /*******************************/
        WinSetFocus(HWND_DESKTOP,WinWindowFromID(hwnd,ID_FONTNAME));
        return(MRESULT)1L;
    /*****************************/
    /* Process keystroke message. */
    /*****************************/
    case WM_CHAR:
        keyFlags=(USHORT) SHORT1FROMMP(mp1);
        if(keyFlags & KC_VIRTUALKEY){
            switch(SHORT2FROMMP(mp2)){
                case VK_ENTER:     // Check for enter key.
                case VK_NEWLINE:   // Check for newline key.
                    mReturn=WinSendDlgItemMsg(hwnd,ID_FONTNAME,
                        LM_QUERYSELECTION, MPFROMSHORT(LIT_FIRST),
                        (MPARAM)NULL);
                    item=(SHORT)mReturn;
                    WinSendDlgItemMsg(hwnd,ID_FONTNAME,LM_QUERYITEMTEXT,
                        MPFROM2SHORT(item,32),(MPARAM)outlineFont);
                    WinSendMsg(WinWindowFromID(hwnd,ID_POINTSIZE),
                        SPBM_QUERYVALUE,(MPARAM)&pointSize,
                         MPFROM2SHORT(0,SPBQ_DONOTUPDATE));
                    AttSet(&currentAttribs,ATT_FONTSIZE,&pointSize);
                    emp=0;
                    if(WinSendMsg(WinWindowFromID(hwnd,ID_OUTLINE),
                        BM_QUERYCHECK,(MPARAM)NULL,(MPARAM)NULL))
                        emp=emp | FATTR_SEL_OUTLINE;
                    if(WinSendMsg(WinWindowFromID(hwnd,ID_STRIKEOUT),
                         BM_QUERYCHECK,(MPARAM)NULL,(MPARAM)NULL))
                        emp=emp | FATTR_SEL_STRIKEOUT;
                    if(WinSendMsg(WinWindowFromID(hwnd,ID_UNDERSCORE),
                         BM_QUERYCHECK,(MPARAM)NULL,(MPARAM)NULL))
                     emp=emp | FATTR_SEL_UNDERSCORE;
                lcid = AddFont(outlineFont,emp);
                WinSendMsg(hwndClient,WM_FONT,(MPARAM)lcid,(MPARAM)pointSize);
                WinDismissDlg(hwnd,TRUE);
```

LISTING 4.3 (Continued).

```
            return(MRESULT)TRUE;
         }
      }
      break;
   /*************************/
   /* Process notifications. */
   /*************************/
   case WM_COMMAND:
      switch(COMMANDMSG(&msg)->cmd){
         case DID_OK:
            mReturn=WinSendDlgItemMsg(hwnd,ID_FONTNAME,
               LM_QUERYSELECTION,
               MPFROMSHORT(LIT_FIRST),(MPARAM)NULL);
            item=(SHORT)mReturn;
            WinSendDlgItemMsg(hwnd,ID_FONTNAME,LM_QUERYITEMTEXT,
               MPFROM2SHORT(item,32),(MPARAM)outlineFont);
            WinSendMsg(WinWindowFromID(hwnd,ID_POINTSIZE),
               SPBM_QUERYVALUE,(MPARAM)&pointSize,
                MPFROM2SHORT(0,SPBQ_DONOTUPDATE));
            AttSet(&currentAttribs,ATT_FONTSIZE,&pointSize);
            emp=0;
            if(WinSendMsg(WinWindowFromID(hwnd,ID_OUTLINE),
               BM_QUERYCHECK,(MPARAM)NULL,(MPARAM)NULL))
               emp=emp | FATTR_SEL_OUTLINE;
            if(WinSendMsg(WinWindowFromID(hwnd,ID_STRIKEOUT),
                BM_QUERYCHECK,(MPARAM)NULL,(MPARAM)NULL))
               emp=emp | FATTR_SEL_STRIKEOUT;
            if(WinSendMsg(WinWindowFromID(hwnd,ID_UNDERSCORE),
                BM_QUERYCHECK,(MPARAM)NULL,(MPARAM)NULL))
               emp=emp | FATTR_SEL_UNDERSCORE;
            lcid=AddFont(outlineFont,emp);
            WinSendMsg(hwndClient,WM_FONT,(MPARAM)lcid,(MPARAM)pointSize);
            WinDismissDlg(hwnd,TRUE);
            return(MRESULT)TRUE;
         case DID_CANCEL:
            WinDismissDlg(hwnd,FALSE);
            return(MRESULT)TRUE;
         }
         break;
      }
      return WinDefDlgProc(hwnd,msg,mp1,mp2);
   }
   /**********************************************************/
   /* Add a font to our font array if not already in array. */
   /**********************************************************/
   LONG AddFont(CHAR *outlineFont, USHORT emp){
      BOOL retCode=FALSE;
      HDC hdc=0L;
      USHORT codePage=GPI_ERROR;
      FATTRS fatAttrs;
      extern HPS hps;
      LONG lcid;
```

LISTING 4.3 (Continued).

```
    extern FONTINFO fonts[AFONTSIZE];
    extern CHAR currentFont[40];
    extern USHORT emphasis;
    extern ATTRIBS currentAttribs;
    BOOL found=FALSE;
    int i=0;
    while((i<AFONTSIZE) && !found){
       if((strcmp(outlineFont,fonts[i].name)==0) &&
          (fonts[i].emphasis==emp)) found=TRUE;
       i++;
    }
    if(!found){
       /***********************/
       /* Get a new local ID. */
       /***********************/
       lcid=GetSetID(hps);
       /**********************************/
       /* Set the attribute of the font. */
       /**********************************/
       codePage=GpiQueryCp(hps);
       fatAttrs.usRecordLength=sizeof(FATTRS);
       fatAttrs.fsSelection=emp; fatAttrs.lMatch=0L;
        strcpy(fatAttrs.szFacename,outlineFont);
       fatAttrs.idRegistry=0; fatAttrs.usCodePage=codePage;
       fatAttrs.lMaxBaselineExt=0L;
       fatAttrs.lAveCharWidth=0L;
        fatAttrs.fsType=0;
       fatAttrs.fsFontUse=(FATTR_FONTUSE_OUTLINE |
          FATTR_FONTUSE_TRANSFORMABLE);
       /************************/
       /* Create logical font. */
       /************************/
       retCode=GpiCreateLogFont(hps,NULL,lcid,&fatAttrs);
       for(i=0;i<AFONTSIZE;i++ ){
          if(fonts[i].name[0]==0)break;
       }
       strcpy(fonts[i].name,outlineFont);   // Put copy of font in draw array.
       fonts[i].id=lcid;     // Put the current local set ID in the array.
       fonts[i].emphasis=emp;  // Save the font emphasis in the draw array.
       }
    else {
       i--;
       lcid=fonts[i].id; // Get the local set ID for the font we aready have.
       }
    emphasis=emp;        // Save the emphasis in the current master.
    strcpy(currentFont,outlineFont); // Put copy of font in current master.
    GpiSetCharSet(hps,lcid);
    AttSet(&currentAttribs,ATT_FONT,&lcid);
    return lcid;
 }
 /******************************/
 /* Get a set ID for our font. */
 /******************************/
```

LISTING 4.3 (Continued).

```
LONG FAR GetSetID(HPS hps){
   #define MAXSETID 254L
   INT i;
   LONG lcid=GPI_ERROR;
   LONG count;
   BOOL retCode=FALSE;
   PLONG lcids=NULL;
   PLONG types;
   PSTR8 names;
   /**********************************************/
   /* See if any local set IDs have been used yet. */
   /**********************************************/
   count=GpiQueryNumberSetIds(hps);
   if(count==0)
      return(1L);
   /*********************************/
   /* Find first unused local set ID. */
   /*********************************/
   if(count!=GPI_ALTERROR){
      lcids=malloc((SHORT)(16*count));
      types=(PLONG)(lcids+count);
      names=(PSTR8)(types+count);
      if(lcids!=NULL)
         retCode=GpiQuerySetIds(hps,count,types,names,lcids);
      if(retCode){
         for(lcid=1;lcid<(MAXSETID+1);lcid++){
            for(i=0;(i<(INT)count) && (lcids[i]!=lcid);i++);
            if(i==(INT)count)break;
         }
         if(lcid==MAXSETID+1)lcid=GPI_ERROR;
      }
      free(lcids);
   }
   return(lcid);
}
/*********************************/
/* Set point size for outline font. */
/*********************************/
BOOL FAR SetPtSize(HPS hps,LONG lcid,LONG pointSize){
   #define POINTSPERINCH 72L
   HDC hdc;
   BOOL retCode=FALSE;
   LONG yDevResFont;
   POINTL points[2];
   LONG ySizeInPels;
   LONG ySizeInWC;
   SIZEF charBox;
   extern POINTL ptlTranslate;    // Origin for scroll translation.
   extern POINTL ptlDefault;      // Origin for printing.
   extern ULONG zoomFactor;
   retCode=GpiSetCharSet(hps,lcid);
   /****************************************/
   /* Query current font vertical resolution. */
   /****************************************/
```

LISTING 4.3 (Continued).

```
    hdc=GpiQueryDevice(hps);
    DevQueryCaps(hdc,CAPS_VERTICAL_FONT_RES,1L,&yDevResFont);
    /*********************************************************/
    /* Calculate point size and convert to world coordinates. */
    /*********************************************************/
    ySizeInPels=((yDevResFont*pointSize)/POINTSPERINCH);
    points[0].x=0L;
    points[0].y=0L;
    points[1].x=0L;
    points[1].y=ySizeInPels;
    SetDefaultView(hps,1,ptlDefault);
    GpiConvert(hps,CVTC_DEVICE,CVTC_WORLD,2L,points) ;
    SetDefaultView(hps,zoomFactor,ptlTranslate);
    ySizeInWC=points[1].y-points[0].y;
    /*******************************************************/
    /* Set the font for the presentation space and make the  */
    /* width and height value type fixed. Then set char box. */
    /*******************************************************/
    charBox.cx=ySizeInWC*0x10000;
    charBox.cy=ySizeInWC*0x10000;
    retCode=GpiSetCharBox(hps,&charBox);
    return(retCode);
}
```

LISTING 4.3 (Continued).

Listing 4.3 actually has several routines which can all be found in FUNCS.C. To start with, however, let's look at the SelectFontDlgProc routine. The SelectFontDlgProc routine is the routine that gets control and works with our font dialog box when the user selects the Fonts pull-down menu item. As you can see, the first thing this routine does when it needs to intialize the dialog box is to find out how many public fonts the system knows about. It does this by using the GpiQueryFonts function and specifying 0 for the number of fonts for which it wants metrics returned. After it knows how many public fonts are available, this routine allocates enough memory to contain all the metrics for all the public fonts and then queries again specifying the number returned. Once all font metrics are returned, this routine interrogates each font in the array. If the font is an outline font and does not have the special code page number 65400, the font facename is inserted in the dialog box list box. (Recall that a code page number of 65400 indicates that the font has special symbols in it that don't map to other code pages.) After the font facename list box has been filled in, the list box is searched to find the current font. If the current font is found in the list box, then that facename is highlighted. If the current font is not found in the list box, then the first facename in the list box is highlighted. Note that the vari-

able currentFont is a global variable which is used throughout the graphic editor edit session.

Once this dialog box font facename list box has been intialized, the font point size spin buttons and emphasis check boxes are intialized. These controls are initialized by other global variables named *pointSize* and *emphasis,* which also exist throughout the graphic editor edit session. Finally, the dialog initialization process is finished by setting the focus to the dialog box font facename list box.

Processing user actions within the font selection dialog box is very straightforward. In general, we look for the "newline" and "enter" keystrokes and treat them the same way we would process the "OK" button. The way we process the "OK" button is by first querying the different controls in the dialog box (font facename, point size, and emphasis) and setting the appropriate variables with the results. Once the control values are known, another function called *addFont* is used to add the desired font facename and the requested emphasis attributes to an array of font descriptions. This array is maintained by the graphic editor program during the entire edit session. Notice that the addFont function also returns a local identifier. This local identifier is the one that has been assigned to the requested font description. After the addFont function returns, the local identifier and the requested point size are sent back to the main window procedure via a user-defined message named `WM_FONT`. As it turns out, when the main window procedure receives the `WM_FONT` message, it updates any of the currently selected text objects with this new font information and redraws them. Also, this font information is now the current font information so any new text objects will use this font description and point size. After the `WM_FONT` message returns, the font selection dialog box is dismissed and this routine is done. If the user would have pushed the cancel button, the font selection dialog would have simply been dismissed and no global variable would have been updated.

Now let's look at the addFont function also found in Listing 4.3. As stated previously, the addFont function maintains an array of font facenames with a particular emphasis attributes for the graphic editor. Therefore, the first thing this function does when called is search the graphics editor font array to see if the requested font and emphasis already exist in the array. If the requested combination does exist in the graphic editor font array, the local identifier assigned to the font description (which is also saved in the ar-

ray) is fetched. If the requested font combination does not exist in the graphic editor font array, a function called getSetID (also shown in Listing 4.3) is used to find an available local identifier for the graphic editor presentation space and the font information passed to the addFont function is used to set up the font attributes structure. The GpiCreateLogFont function is then used to assign a logical font description to the newly found local identifier. After the GpiCreateLogFont function returns, the font array is searched for the first empty record and the new font information is stored in it. Once a local identifer for the font and the font array has been updated as needed, the global variables named currentFont and emphasis are updated. Finally, the current font for the graphic editor presentation space is updated with the GpiSetCharSet function and the addFont function returns the local identifier for the new current font to the caller.

The last function shown in Listing 4.3 that we haven't discussed yet is called setPtSize. The setPtSize function is used by the graphic editor when a text object is about to be drawn. This function insures that the request size of the font for a specified local identifier is set correctly. The first thing this function does is to insure the request font is also the current font by using the GpiSetCharSet function. The device's effective vertical font resolution is then found by using the DevQueryCaps function. Once the effective vertical font resolution is retrieved, the requested point size is calculated in device units. The GpiConvert function is then used to change the point size calculation from device to world coordinate units. Finally, the point size is converted to FIXED by multiplying it by X'10000' and this number is used as input to the GpiSetCharBox function. Notice that the character box width field is set to the same value as the height field. This is because the character box which represents the device's vertical and horizontal resolution are equal for outline fonts. After the GpiSetCharBox function returns, the characters displayed by the graphic editor will have the requested point size.

Now, as it turns out, OS/2 gives you yet another way to find fonts! The way it does this is with a function called *WinFontDlg* that actually displays a standard font selection dialog box which, on return, has a font attribute stucture all set up for you. Hence, all you need to do is use the GpiCreateLogFont function with the font attribute structure returned. We chose not to use the WinFontDlg function in the graphic editor for two reasons. First, we wanted to have an example of how to query font metrics and then use some of this metric information to find a font. Second, we wanted more control

over the point size than what appeared to be available with the WinFontDlg function. We did, however, use the WinFontDlg function in the generic Browse utility. A small code fragment where we use this function is shown in Listing 4.4. This code fragment came from `BROWSE.C` and is the processing for the Font pull-down menu item in the Browse utility.

As you can see by looking at Listing 4.4, the WinFontDlg function requires an address of a font dialog structure (FONTDLG) as an input parameter. This font dialog structure is a way for you to customize the way the font dialog will work, as well as the area where the font attribute data is returned. This structure is static in the Browse utility and is initialized only once on its first use. After the font dialog stucture's first use, the strucure is reused with whatever was left in it from the previous WinFontDlg call. On return from the WinFontDlg function, a test to see if the function completed successfully and if the user pushed the "OK" button is done. If the user did press "OK", a copy of the font attributes from the font dialog structure is made. If the "OK" button wasn't pressed, the font attributes from the last successful font selection are used. Right before the GpiCreateLogFont is used, however, the current local identifier used for font selection is deleted and a new local identifier is found. (This is different from the graphic editor, which keeps an array of font information including local identifiers used for font descriptions.) The new font is then made the current font for the browse presentation space by using the GpiSetCharSet function. Finally, if the new font is an outline font, the character box attribute is set so the size of the font will be what was selected by the user via the font dialog structure. The remainder of the pull-down menu item process sets variables used by other parts of the Browse utility to perform string and cursor placement in the client window. The WinInvalidateRect function is then issued so the utility will redraw the client window area with the new font information.

```
switch (msg){
   /**********************************/
   /* Process pull-down menu items. */
   /**********************************/
   case WM_COMMAND:
      switch (COMMANDMSG(&msg)->cmd){
         /***********************************/
         /* Process font pull-down option. */
         /***********************************/
         case IDM_BFONT:
         if(first){
```

LISTING 4.4 Browse utility font selection processing.

```
            memset(&fntDialog,0,sizeof(FONTDLG));
            fntDialog.cbSize=sizeof(FONTDLG);
            fntDialog.fl=FNTS_HELPBUTTON | FNTS_CENTER;
            fntDialog.clrFore=CLR_BLACK;
            fntDialog.clrBack=SYSCLR_WINDOW;
             fntDialog.fxPointSize=MAKEFIXED(10,0);
            fntDialog.hpsScreen=hpsBrowse;
            fntDialog.usWeight=5;
            fntDialog.usWidth=5;
            familyName[0]=0;
            fntDialog.pszFamilyname=familyName;
            fntDialog.usFamilyBufLen=FACESIZE;
            first=FALSE;
         }
          hwndFontDlg=WinFontDlg(HWND_DESKTOP,hwnd,&fntDialog);
          if((hwndFontDlg) && (fntDialog.lReturn==DID_OK)){
            memcpy(&fAttrs,&fntDialog.fAttrs,sizeof(FATTRS));
         }
         if(lcid<256)GpiDeleteSetId(hpsBrowse,lcid);
         lcid=GetSetID(hpsBrowse);
         GpiCreateLogFont(hpsBrowse,NULL,lcid,&fAttrs);
         GpiSetCharSet(hpsBrowse,lcid);
         if(fntDialog.fAttrs.fsFontUse==FATTR_FONTUSE_OUTLINE) {
            /**************************/
            /* Set the character box. */
            /**************************/
            sizefxCharBox.cx=fntDialog.lEmHeight*0x10000;
            sizefxCharBox.cy=fntDialog.lEmHeight*0x10000;
            GpiSetCharBox(hpsBrowse,&sizefxCharBox);
         }
          GpiQueryFontMetrics(hpsBrowse,(LONG)sizeof(fm),&fm);
         xChar=(SHORT)fm.lMaxCharInc;     // Set new font width.
          yChar=(SHORT)fm.lMaxBaselineExt; // Set new font height.
         WinSendMsg(hwnd,WM_SIZE,
            MPFROM2SHORT(cxnew,cynew),MPFROM2SHORT(cxnew,cynew));
         WinInvalidateRect(hwnd,NULL,FALSE);
         return(MRESULT)TRUE;
      /**************************************/
      /* Default processing for WM_COMMAND. */
      /**************************************/
      default:
         return WinDefWindowProc(hwnd,msg,mp1,mp2);
   }
```

LISTING 4.4 (Continued).

We now know how to select fonts. We also know that there is a lot of data available about fonts that is supposed to help with character and string placement on the presentation page. So what GPI functions are available to help us manage the formatting of text output? As it turns out, there are only a few GPI functions that actually output text. But, because of all the text attributes, fonts, and font metric data, you can really do about anything imaginable with text. As you may recall from Chapter 3, text attributes al-

low you to do things like choose the direction in which text will be written, choose how the text will be aligned, choose shear lines, and so on. Hence, all of these text attributes will affect the way text is drawn with your chosen font when you use the GPI text output function. These GPI text output functions are called *GpiCharString*, *GpiCharStringAt*, *GpiCharStringPos*, and *GpiCharStringPosAt*. All of these GPI functions take a pointer to a character string that you want to output. The difference between the functions that end with *At* and those that don't is that the functions that end with *At* also pass the starting point for the string. The functions that don't end with *At* use the current position as their starting point. GpiCharString and GpiCharStringAt are fairly simple in the function they perform. These two functions simply write the text to the presentation page using the positioning and width information available for the current font and update the current position to the end of the string. GpiCharStringPos and GpiCharStingPosAt, however, can specify much more information about how the text string is to be drawn. In particular, the GpiCharStringPos and GpiCharStringPosAt functions also pass rectangle coordinate information, options information, and an array of character width data. How the options parameter is set will determine how the rectangle and width data is used. Following is a list of defined values available (which can be ORed together) to be used with the options parameter of the GpiCharStringPos and GpiCharStringPosAt functions:

CHS_OPAQUE – This option specifies that the area defined by the rectangle coordinates is to be filled wih the background color before text is drawn.

CHS_VECTOR – This option specifies that width data supplied as a parameter is to be used. (By supplying width data, you have control over the character position within a text string.)

CHS_LEAVEPOS – This option specifies that the current position is not altered by this function. If not specified, the current position will be updated to the end of the string.

CHS_CLIP – This option specifies that the text is to be clipped to the rectangle.

CHS_UNDERSCORE – This option specifies that the text is to be underscored.

CHS_STRIKEOUT – This option specifies that the text is to be overstruck.

As you can see from the options available, you can do a lot more with these functions than you can with the GpiCharStringPos and GpiChar-

StringPosAt functions. The graphic editor only uses the GpiCharStringPos function to display its text objects. However, the graphic editor doesn't do this so it can use the rectangle features of the function; rather, the graphic editor uses the GpiCharStringPos function so it can provide kerning for fonts that can be kerned! Of course this begs the question, "How do you get width data for kerning?" Great question! The obvious answer, "The GPI!" To be much more specific, you use a function called *GpiQueryKerningPairs*. When you use the GpiQueryKerningPairs function, you supply the function with a pointer to a space where it is to place the kerning pair data. Of course, you have to insure the space is large enough to hold all the data, so you should query the font metrics beforehand to see how many kerning pairs exist. The kerning pair data is returned as an array of structures named KERNINGPAIRS. The KERNINGPAIRS structure has three fields named sFirstChar, sSecondChar, and lKerningAmount. As you may guess, sFirstChar is the first character of the pair, sSecondChar is the second character of the pair, and lKerningAmount is the value added to the width data of the first character to adjust for kerning. And to find the width data for the current font, use the function called *GpiQueryWidthsTable*. The GpiQueryWidthsTable function is fairly straightforward. It will return an array of width values for the current font based on a starting code point and a length.

To understand how you might use all of this width data and kerning pair information, look at Listing 4.5. Listing 4.5 shows the routine that is used by the graphic editor to retrieve width data for a given string. This routine can be found in the source part named `FUNCS.C`. As you can see, the first thing this routine does is query all the widths for the first 256 code points for the current code page. After the current font's widths have been retrieved, a function called *GpiQueryFontMetrics* is used to gather the font metrics for the current font. The current font is then tested to see if it has kerning pair data. If it does, this kerning pair data is also retreived. Now that all the width data has been collected, a widths array that reflects the widths of the character in the input string is built. If kerning is not available, this widths array is ready to be used by the GpiCharStringPos function. If kerning is available, however, each character in the string and its following adjacent character are tested to see if they match a kerning pair in the KERNINGPAIRS array. If matches are found, adjustments are made to the appropriate widths in the

widths array and, as before, the string's widths array is now ready to be used by the GpiCharStringPos function.

```
/**********************************************/
/* Set width table and calculate string length. */
/**********************************************/
void FAR SetWidthsTable(HPS hps,LONG *widths,CHAR *str){
   FONTMETRICS fm;
   PKERNINGPAIRS pairs=NULL;
   LONG i,j;
   LONG widthTable[256];
   /*****************************************/
   /* Get width table for all 256 code points. */
   /*****************************************/
   GpiQueryWidthTable(hps,0L,256L,widthTable);
   /***********************************************/
   /* Check for kerning font and get pair if needed. */
   /***********************************************/
   GpiQueryFontMetrics(hps,(LONG)sizeof(FONTMETRICS),&fm);
    if(fm.sKerningPairs){
   pairs=malloc(fm.sKerningPairs*sizeof(KERNINGPAIRS));
   GpiQueryKerningPairs(hps,fm.sKerningPairs,pairs);
}
/******************************************************/
/* Set widths for each character in string before kerning. */
/******************************************************/
for(i=0;i<(LONG)strlen(str);i++)
   widths[i]=widthTable[str[i]];
/**********************************************/
/* Modify widths array with kerning adjustments. */
/**********************************************/
if(fm.sKerningPairs){
   for(i=0;i<(LONG)strlen(str);i++){
      for(j=0;j<fm.sKerningPairs;j++){
         if(str[i]==(UCHAR)pairs[j].sFirstChar &&
            str[i+1]==(UCHAR)pairs[j].sSecondChar){
            widths[i]+=pairs[j].lKerningAmount;
            break;
         }
      }
   }
}
if(fm.sKerningPairs)free(pairs);  // Free kerning values.
return;
}
```

LISTING 4.5 Draw program set widths table function.

When the graphic editor outputs text, most of the text attributes are left at the default value. This is because the graphic editor uses only outline fonts and things such as shear and angle are accomplished through transformations just like all the other objects you can draw. Also, because each individual text string is treated as an independent object, we don't make much use of

the font metrics available to help position the text. Look at BROWSE.C, however, and you can see more examples of how the font metrics are used to format text. This is because the Browse utility allows you to select from mutliple fonts and font sizes, and then formats the text to fit the confines of the browse window. Listing 4.6 shows some small code fragments from BROWSE.C that deal with formatting text for the Browse utility.

As you can, when the browse window is created, the presentation page units are defined as pels. Note that, because we do no transformations between world space and presentation page space in this Browse utility, we do not have to issue GpiConvert calls to convert units from pels to world space coordinates. After the presentation space is created, the Browse utility creates somes semaphores that are used to coordinate drawing between a background thread and the Browse utility window procedure. It then creates a default Courier font for the presentation space to use. If this default Courier logical font was not assigned to the browse presentation space, the default system monospaced font would be assigned to the presentation space. Once the logical font is assigned to the presentation space, the text alignment attribute is set so text strings will be drawn from the bottom-left point of the character cell and then font metrics are queried so we can set some global variables having to do with the font size. In particular, we save the font height and width information for later use.

A key place and time that font height and width information are used is during the processing of the WM_SIZE message. When the WM_SIZE message is received, the number of text lines and columns that can be displayed in the window can be calculated. (Note that columns is calculated for the worst case situations [monospaced]). Therefore, proportional fonts will appear to shift in the window even when there appears to be space to the right.) The number of text lines is merely the height of the client area divided by the font height. Because integer arithmetic is used, the possible fractional line is discarded. Then, because the Browse utility alway displays a status line at the bottom of the window, the number of text lines (rows) that can be displayed in the window is adjusted. Column information for the window is calculated in a similar way as text lines, but we always insure that one more column of text can be displayed in the window than might actually fit. After row and column information have been calculated, the starting position for the top line of text in the window is calculated. This starting position is simply the far left side of the window (0) and the height of the font subtracted

from the height of the window. (This program does not make an adjustment for internal leading for the top row of text, but you could!)

Whenever a user requests a new font and font size from the Fonts pull-down menu item for the Browse utility, the same basic steps are taken to recalculate the global variables just discussed. By looking at Listing 4.6, you can see how this is done. Earlier, we discussed how the Browse utility uses the Font Dialog function to acquire a new font. After the logical font has been created, however, you can also see that the new size is saved and the `WM_SIZE` message is invoked to recalculate row/column information.

By looking at the `WM_BUTTON1DONE` message routine in Listing 4 .6, you can see how cursor placement is determined by the Browse utility. What makes cursor placement interesting is that you can't assume a simple monospaced font is used; hence, more careful calculation is needed. As you can see, the first thing done by the `WM_BUTTON1DONE` routine is to record the vertical and horizontal coordinates of the mouse. A copy of the starting point for the top line of text is then made so we can use it as a working reference point for row/column calculations. After the text starting point has been copied, a count of the possible number of rows of text is determined for the current browse window. Now, by simply comparing the vertical mouse position with the vertical starting point to write text, we can see if the mouse pointer is in the current row of text. That is, if the vertical mouse position is less than the vertical position to write text, then we haven't found the row for the cursor. If we haven't found the row for the cursor, decrement the vertical position to write text by the height of the font and try again. We keep trying this procedure until we have determined the row in the window for which the cursor may apply. Once the possible row number for the cursor has been calculated, it is then checked to make sure it is on a possible line of text in the window. If the row number is larger than the number of possible lines of text in the window, the last line of text (count) is used. This row number is then used to calculate the line number in the text file to which the cursor will point (cursorY).

Now that the line number to which the cursor is pointing is known, the line of text is copied out of the text buffer into a work area (pointed to by str). This line of text, however, is not just copied byte for byte. Instead, tabs are looked for in the string and are replaced by the appropriate number of blank characters. (This logic assumes tab stops are located at columns that are multiples of eight.) As the string is copied, the length of the string is also

tracked and the ending carriage return is ignored. The length of the string is then adjusted by the current column number found on the left edge of the window. Furthermore, an index into the string which references the first character that would be found on the left edge of the window is set. Now that the visible portion of the string in the window is known, width information for this portion is retrieved so the correct column number can be determined. This column calculation is done in a very similar way as determining the cursor row, but the string width data is used as an incrementor instead of the font width metric. After the relative column postion has been determined, the starting absolute character postion within the text string is determined. This absolute position is the cursor horizontal position (cursorX).

In some cases, moving the cursor right will cause the window to scroll. For a proportional font, this scrolling may occur even if enough space appears to exist on the right side of the physical cursor location. In those cases, the horizontal scroll message is activated and the window is repainted. When the window is repainted, the physical cursor will be set. In the case where no scrolling needs to occur, the physical cursor is set using the function SetCursor which can also be found in `BROWSE.C`. We don't show `BROWSE.C` in Listing 4.6 because it is very similar in logic to what you've just seen and can be reviewed on your own.

```
switch (msg){
   case WM_CREATE:
      hdcBrowse=WinOpenWindowDC(hwnd); // Create window device context.
      sizl.cx=0;              // Create Micro-PS.  Keep as global.
      sizl.cy=0;
      hpsBrowse=GpiCreatePS(habBrowse,hdcBrowse,&sizl,
         PU_PELS | GPIT_MICRO | GPIA_ASSOC | GPIF_DEFAULT);
      /**************************************************************/
      /* Set semaphore before thread is created so it will wait      */
      /* right away.  This semaphore is used by the backGroundPaint  */
      /* thread.                                                     */
      /**************************************************************/
      DosCreateEventSem(NULL,&startPainting,0,0);
      DosCreateEventSem(NULL,&donePainting,0,0);
      /*******************/
      /* Create threads. */
      /*******************/
      DosCreateThread(&tidBack,backGroundPaint,0UL,0UL,8192);
      lcid=GetSetID(hpsBrowse);
      /*********************************/
      /* Set the attribute of the font. */
      /*********************************/
```

LISTING 4.6 Browse utility font draw routines.

```
      codePage=GpiQueryCp(hpsBrowse);
      fAttrs.usRecordLength=sizeof(FATTRS);
      fAttrs.fsSelection=0
      ; fAttrs.lMatch=0L;
       strcpy(fAttrs.szFacename,"Courier");
       fAttrs.idRegistry=0;
       fAttrs.usCodePage=codePage;
      fAttrs.lMaxBaselineExt=0L;
      fAttrs.lAveCharWidth=0L;
      fAttrs.fsType=0;
      fAttrs.fsFontUse=0;
      /************************/
      /* Create logical font. */
      /************************/
      fRet=GpiCreateLogFont(hpsBrowse,NULL,lcid,&fAttrs);
      GpiSetCharSet(hpsBrowse,lcid);
      first=TRUE;              // Set flag for first font selection.
      /*******************************************************/
      /* Get scroll bar handles and current font size information. */
      /*******************************************************/
      hwndTemp=WinQueryWindow(hwnd,QW_PARENT);
      hwndVScroll=WinWindowFromID(hwndTemp,FID_VERTSCROLL);
      hwndHScroll=WinWindowFromID(hwndTemp,FID_HORZSCROLL);
      GpiSetTextAlignment(hpsBrowse,TA_LEFT,TA_BOTTOM);
      GpiQueryFontMetrics(hpsBrowse,(LONG)sizeof(fm),&fm);
      xChar=(SHORT)fm.lMaxCharInc;      // Set global variable font width.
       yChar=(SHORT)fm.lMaxBaselineExt;  // Set global variable font height.
      return 0;
   /************************/
   /* Process size message. */
   /************************/
   case WM_SIZE:
      cynew=SHORT2FROMMP(mp2);
      cxnew=SHORT1FROMMP(mp2);
      rows=(cynew/yChar);   // Calculate number of row in window.
      rows=(rows>1) ? (rows-1) : 0;  // Leave row for status line.
      cols=(cxnew/xChar)+1; // Calculate number of columns in window.
      ptTopLeft.x=0;     // Calculate top left point to start drawing text.
      ptTopLeft.y=cynew-yChar;
      /*********************************************************/
      /* Calculate line number, column number, and scroll bar sizes. */
      /*********************************************************/
      scrollVMax=(curYPos>(lineCount-rows)) ? curYPos : (lineCount-rows);
      curYPos=(curYPos<scrollVMax) ? curYPos : scrollVMax;
      scrollHMax=(curXPos>(maxWidth-cols)) ? curXPos : (maxWidth-cols);
      curXPos=(curXPos<scrollHMax) ? curXPos : scrollHMax;
      WinSendMsg(hwndVScroll,SBM_SETTHUMBSIZE,
         MPFROM2SHORT(rows,lineCount),(MPARAM)(NULL));
      WinSendMsg(hwndVScroll,SBM_SETSCROLLBAR,
         (MPARAM)MAKEULONG(curYPos,0),MPFROM2SHORT(0,scrollVMax));
      WinEnableWindow(hwndVScroll,scrollVMax ? TRUE : FALSE);
      WinSendMsg(hwndHScroll,SBM_SETTHUMBSIZE,
```

LISTING 4.6 (Continued).

```
         MPFROM2SHORT(cols,maxWidth),(MPARAM)(NULL));
      WinSendMsg(hwndHScroll,SBM_SETSCROLLBAR,
         (MPARAM)MAKEULONG(curXPos,0),MPFROM2SHORT(0,scrollHMax));
      WinEnableWindow(hwndHScroll,scrollHMax ? TRUE : FALSE);
      return 0;
   /*******************************/
   /* Process mouse button 1 down. */
   /*******************************/
   case WM_BUTTON1DOWN:
      x=MOUSEMSG(&msg)->x; // Convert to grid coordinate.
      y=MOUSEMSG(&msg)->y;
      ptPaint.x=ptTopLeft.x;
      ptPaint.y=ptTopLeft.y;
      /************************************************************/
      /* count is the number of rows of text in the current window. */
      /************************************************************/
      count=((lineCount-curYPos)<rows) ? (lineCount-curYPos) : rows;
      /**************************************************/
      /* Find the row number from the top of window where */
      /* mouse was clicked.  (rowNum)                    */
      /**************************************************/
      rowNum=0;
      while((LONG)y<ptPaint.y){
         ptPaint.y-=yChar;
         rowNum++;
      }
      /*********************************************************/
      /* Set cursorY to the line number selected with the mouse. */
      /*********************************************************/
      if(rowNum<count)cursorY=curYPos+rowNum;
      else cursorY=curYPos+count-1;
      /*******************************************/
      /* Set the index to the line offsets array. */
      /*******************************************/
      offsetIndex=*((PULONG)fileOffsets+(cursorY));
      /****************************************************/
      /* Calculate offset to text line in file buffer (temp). */
      /****************************************************/
      temp=bufferPtr+offsetIndex-baseOffset;
      str=stringout;  // Set up pointer to work area for string copy.
      *str='\0';
      /**************************************************************/
      /* Copy string and replace tab characters with up to eight spaces.*/
      /**************************************************************/
      for(length=0;(*temp!='\n') && (length<250 );*temp++){
         if(*temp=='\t'){
            blkCnt=8-(length%8);
            length=length+blkCnt;
            while(blkCnt>0){
               *str++=' ';
               blkCnt--;
            }
         }
```

LISTING 4.6 (Continued).

```
   else
      *str++ =*temp;
   length++;
}
/****************************/
/* Remove carriage returns. */
/****************************/
if(*(str-1)==13)length--;
*(str+length)=0;
/*****************************************************************/
/* Adjust length by current column found on window left edge. */
/*****************************************************************/
if(length>curXPos){
   hIndex=curXPos;
   length-=curXPos;
}
else {
   hIndex=length;
   length=0;
}
/**********************************************************************/
/* Get widths information for part of text string shown in window. */
/**********************************************************************/
 SetWidthsTable(hpsBrowse,widthValues,widthTable,&stringout[hIndex],
   &strWidth);
colNum=0;
/*************************************************************/
/* Get the length of the text string found in the window. */
/*************************************************************/
 horzCnt=strlen(&stringout[hIndex]);
/********************************************************************/
/* Calculate the column number and position of character selected.*/
/********************************************************************/
 while((ptPaint.x<x) && (colNum<horzCnt)){
   ptPaint.x+=widthValues[colNum];
   colNum++;
}
if(colNum!=0)colNum--;
cursorX=curXPos+colNum;  // Set X cursor position.
/**********************************************************/
/* See if window should be scrolled or just set cursor. */
/**********************************************************/
if(cursorX>(curXPos+cols-2)){
   curXPos=cursorX;
   WinSendMsg(hwndHScroll,SBM_SETPOS,(MPARAM)curXPos,NULL);
   WinInvalidateRect(hwnd,NULL,TRUE);
}
else {
   offsetIndex=*((PULONG)fileOffsets+(cursorY));
   temp=bufferPtr+offsetIndex-baseOffset;
   SetCursor(hwnd,temp);
}
return(MRESULT)TRUE;
```

LISTING 4.6 (Continued).

```
/*************************/
/* Process paint message. */
/*************************/
case WM_PAINT:
   hpsBrowse=WinBeginPaint(hwnd,hpsBrowse,&rectPaint);
   filling=FALSE;
   DosWaitEventSem(donePainting,SEM_INDEFINITE_WAIT);
   DosResetEventSem(donePainting,&doneEventCount);
   WinFillRect(hpsBrowse,&rectPaint,CLR_BACKGROUND);
    gptPaint.x=ptTopLeft.x;
   gptPaint.y=ptTopLeft.y;
   /****************************************************/
   /* Get the first line to display and the count of them  */
   /* to display.                                          */
   /****************************************************/
   lineOut=curYPos;
   gCount=((lineCount-curYPos)<rows) ? (lineCount-curYPos) : rows;
   /*****************************************************/
   /* Get the minimum and maximum file offset values needed. */
   /*****************************************************/
   minOffset=*((PULONG)fileOffsets+(lineOut));
    if((lineOut+count+1)<lineCount)
      maxOffset=*((PULONG)fileOffsets+(lineOut+count+1));
   else
      maxOffset=*((PULONG)fileOffsets+(lineCount-1));
   /*********************************************/
   /* If needed, read in new chunk of input file. */
   /*********************************************/
   if(minOffset<baseOffset || maxOffset>(baseOffset+FILE_BUF_SIZE)){
      if(minOffset<(FILE_BUF_SIZE/2))
         baseOffset=0;
      else
         baseOffset=minOffset-(FILE_BUF_SIZE/2);
      DosSetFilePtr(fileHandle,baseOffset,FILE_BEGIN,&local);
      DosRead(fileHandle,bufferPtr,FILE_BUF_SIZE,&bytesRead);
   }
   /*******************************************************/
   /* Put cursor up here if it can be found within the window. */
   /*******************************************************/
   gOffsetIndex=*((PULONG)fileOffsets+(cursorY));
   gTemp=bufferPtr+gOffsetIndex-baseOffset;
   SetCursor(hwnd,gTemp);
   filling=TRUE;
   DosPostEventSem(startPainting);
   WinEndPaint(hpsBrowse);
   return 0;
/********************************/
/* Process pull-down menu items. */
/********************************/
case WM_COMMAND:
   switch (COMMANDMSG(&msg)->cmd){
```

LISTING 4.6 (Continued).

```
      /**********************************/
      /* Process font pull-down option. */
      /**********************************/
      case IDM_BFONT:
         if(first){
            memset(&fntDialog,0,sizeof(FONTDLG));
            fntDialog.cbSize=sizeof(FONTDLG);
            fntDialog.fl=FNTS_HELPBUTTON | FNTS_CENTER;
            fntDialog.clrFore=CLR_BLACK;
            fntDialog.clrBack=SYSCLR_WINDOW;
             fntDialog.fxPointSize=MAKEFIXED(10,0);
            fntDialog.hpsScreen=hpsBrowse;
            fntDialog.usWeight=5;
            fntDialog.usWidth=5;
            familyName[0]=0;
            fntDialog.pszFamilyname=familyName;
            fntDialog.usFamilyBufLen=FACESIZE;
            first=FALSE;
         }
          hwndFontDlg=WinFontDlg(HWND_DESKTOP,hwnd,&fntDialog);
          if((hwndFontDlg) && (fntDialog.lReturn==DID_OK)){
            memcpy(&fAttrs,&fntDialog.fAttrs,sizeof(FATTRS));
         }
         if(lcid<256)GpiDeleteSetId(hpsBrowse,lcid);
         lcid=GetSetID(hpsBrowse);
         GpiCreateLogFont(hpsBrowse,NULL,lcid,&fAttrs);
         GpiSetCharSet(hpsBrowse,lcid);
         if(fntDialog.fAttrs.fsFontUse==FATTR_FONTUSE_OUTLINE) {
            /*************************/
            /* Set the character box. */
            /*************************/
            sizefxCharBox.cx=fntDialog.lEmHeight*0x10000;
            sizefxCharBox.cy=fntDialog.lEmHeight*0x10000;
            GpiSetCharBox(hpsBrowse,&sizefxCharBox);
         }
          GpiQueryFontMetrics(hpsBrowse,(LONG)sizeof(fm),&fm);
         xChar=(SHORT)fm.lMaxCharInc;     // Set new font width.
         yChar=(SHORT)fm.lMaxBaselineExt; // Set new font height.
         WinSendMsg(hwnd,WM_SIZE,
            MPFROM2SHORT(cxnew,cynew),MPFROM2SHORT(cxnew,cynew));
         WinInvalidateRect(hwnd,NULL,FALSE);
            return(MRESULT)TRUE;
         /**************************************/
         /* Default processing for WM_COMMAND. */
         /**************************************/
      default:
         return WinDefWindowProc(hwnd,msg,mp1,mp2);
   }
      /**********************************************/
      /* Background thread for drawing browse text. */
      /**********************************************/
      VOID backGroundPaint(ULONG dummy){
```

LISTING 4.6 (Continued).

```
LONG i;
SHORT x,y,toggle;
HAB habt;
/****************************************************************/
/* Get an anchor block handle so thread can access PM functions. */
/****************************************************************/
habt=WinInitialize(0);
for(;;){
   DosPostEventSem(donePainting);
   /*********************************************/
   /* Wait for main process to clear semaphore. */
   /*********************************************/
   DosWaitEventSem(startPainting,SEM_INDEFINITE_WAIT);
   /*********************************/
   /* Display full window of data. */
   /*********************************/
   while((gCount--) && (filling)){
      gOffsetIndex=*((PULONG)fileOffsets+(lineOut));
      gTemp=bufferPtr+gOffsetIndex-baseOffset;
       gstr=stringout;
      *gstr='\0';
      /***********************************************/
      /* Replace tab characters with up to eight spaces. */
      /***********************************************/
      for(gLength=0; (*gTemp!='\n') && (gLength<250 ); *gTemp++){
         if(*gTemp=='\t'){
            gblkCnt=8-(gLength%8);
            gLength=gLength+gblkCnt;
            while(gblkCnt>0){
               *gstr++=' '; gblkCnt--;
            }
         }
         else *gstr++ =*gTemp;
         gLength++;
      }
      /****************************/
      /* Remove carriage returns. */
      /****************************/
      if(*(gstr-1)==13)gLength--;
      if(gLength>curXPos){
         hIndex=curXPos;
         gLength-=curXPos;
      }
      else {
         hIndex=gLength;
         gLength=0;
      }
      /***********************************************************/
      /* Output (partial) line to window and update loop variables.*/
      /***********************************************************/
       SetWidthsTable(hpsBrowse,widthValues,
         widthTable,&stringout[hIndex],&gstrWidth);
```

LISTING 4.6 (Continued).

```
            GpiMove(hpsBrowse,&gptPaint);
            GpiCharStringPos(hpsBrowse,NULL,CHS_VECTOR,
               (LONG)gLength,&stringout[hIndex],widthValues);
            gptPaint.y-=yChar;     // Move down 1 row.
            lineOut++;             // Increment to line offset.
         }
         if(filling){
            /******************************/
            /* Output line count position. */
            /******************************/
            stringout[0]=0;
             strcat(stringout,"Line ");
            CvtInt((cursorY+1),&stringout[strlen(stringout)]);
            strcat(stringout," of ");
            CvtInt((lineCount),&stringout[strlen(stringout)]);
            strcat(stringout,",  Column ");
            CvtInt((cursorX+1),&stringout[strlen(stringout)]);
            strcat(stringout,".  ");
            strcat(stringout,pBrws->filename);
            GpiSetColor(hpsBrowse,CLR_RED);
            SetWidthsTable(hpsBrowse,widthValues,
               widthTable,&stringout[0],&gstrWidth);
            gptPaint.y=0;
         GpiMove(hpsBrowse,&gptPaint);
          GpiCharStringPos(hpsBrowse,NULL,CHS_VECTOR,
            (LONG)strlen(&stringout[0]),&stringout[0],widthValues);
         }
         GpiSetColor(hpsBrowse,CLR_BLACK);
         DosResetEventSem(startPainting,&startEventCount);
      filling=FALSE;
   }
}
```

LISTING 4.6 (Continued).

Finally, let's look at how the lines of text are drawn in the Browse utility. In the Overview chapter, we discussed how the WM_PAINT message routine is structured so it can interact with a background thread that actually draws the text in the browse window. (The WM_PAINT routine really just sets up some global variables used by the background thread and insures that the correct portion of text data is in a global data buffer before the background thread is unblocked to paint the window.) By looking at the background thread procedure in Listing 4.6, you can see that most of the logic you've seen already in other places. The variable named lineOut is the first line number that is to be displayed in the window. Hence, lineOut is just an index into the data buffer that has the text to be displayed. The line of text referenced by lineOut is copied and adjusted for tabs and the visible portion of the string is calculated just as you saw in the WM_BUTTON1DOWN routine. Then width data is retrieved for this string and the text is displayed in the appropri-

ate row of the window. After the line of text is drawn, the lineOut variable is incremented and the vertical position for the next line of text is adjusted so the next line of text can be drawn. This drawing of the next line of text loops until all lines for the window have been drawn or until the WM_PAINT routine interrupts via the filling indicator. After all lines of text have been drawn, the status line which has the cursor row and column numbers and the number of lines found in the file being viewed is formatted and displayed at the very bottom left of the window. The backgound thread then allows the WM_PAINT routine to run again and waits for the next time it needs to redraw the screen with new information.

As stated when we first started this chapter, drawing text in a graphical environment is more tedious than what you may be used to. But think about the possibilities and look at the results!

CHAPTER 5
Building Blocks of the GPI

Earlier in this book when we were describing the viewing pipeline, we discussed how an object could be created in world space and then used multiple times in model space using different transformations. When we discussed this, we didn't talk about how objects could actually be created or manipulated with the GPI, but rather the conceptual steps of picture construction. When you look at the drawing functions of the GPI (things like GpiLine, GpiCharStringAt, or GpiPartialArc), you may begin to wonder what constitutes an object. Is an object something like a line or a curve? Of course these are objects, but how can something more complex, like a drawing of a nut or a bolt, become an object? If we use the simple drawing functions without grouping them in some way, then working in this environment may seem like a real mess! What is needed is a way to group drawing commands so they can be used as a unit. The drawing commands needed to generate something like a nut or a bolt coulc then be treated like a single object. Furthermore, it would also be desirable to be able to edit these more complex objects. Therefore, once you create a complex object, you need a way to index or move through the object parts to make modifications.

The OS/2 GPI allows this type of drawing control by providing a series of structures that relate to one another and a set of functions that allow you to

manipulate them. These structures are called *orders*, *elements*, and *segments*. The way these structures relate to one another is in a hierarchy. More precisely, a segment contains elements and elements contain orders. The following sections of this chapter discuss these different structures in detail so you can better understand their value and how you might manipulate them. Finally, we have provided a feature in the graphic editor that comes with this book so you can examine the orders, elements, and segments that are generated by the editor. We will conclude this chapter by reviewing how this feature was implemented by our application. By doing this, you should gain some knowledge of what these structures are and how you can manipulate them within your own program.

ORDERS

An order is the smallest unit or structure that defines a drawing primitive or attribute in a graphics segment. An example of a drawing primitive is *Box at Given Position* and an example of a drawing attribute is *Set Background Color*. The GPI has about 200 different orders and can be partially parsed by the following rules:

1. The first byte of an order is the order code and determines the type and length of the order.

2. If the order code is X'00' or X'FF' this is a 1-byte order (no data associated with the order.) As it turns out, X'00' is a *No-Operation* order and X'FF' is an *End of Symbol Definition* order.

3. If the order code is X'FE' this order is considered an extended order. The second byte of an extended order is the order qualifier which determines the order function. The third and fourth bytes of an extended order determine the length of the order data associated with it. Note that the third byte is the most significant byte from this length pair. Following the fourth byte is the order data. An example of an extended order is X'FEB0', *Character String Extended at Current Position.*

4. If the order code has the following bit configuration (0xxxx1xxxx) where x can be either 0 or 1, then the order is a 2-byte order. The second

byte in this situation is the order data. An example of a 2-byte order is X'7F', *End Path.*

5. If the order code does not fall under one of the previous rules, it is considered a long order. The second byte of a long order determines the length of the order data associated with it. Following the second byte is the order data. An example of a long order is X'C0', *Box at Given Position.*

As stated before, these rules only give you partial parsing capability of orders. The obvious cool part of parsing orders is to be able to determine the function and the data associated with the orders. In most cases, this knowledge may not seem too exciting if you let the OS/2 GPI generate these orders for you. But if you get into debug mode, which seems to happen to us a lot, seeing and understanding this order data can be very enlightening. To see a complete definition of all the orders available, refer to an IBM technical reference. Or, as you will soon see, our object viewing source code can be examined to find all the orders we were aware of when writing this book.

You can also generate order data from within your application (outside of the GPI draw functions) and have these orders drawn, saved to segment store, or both. To do this, you will need to understand the order data intimately and should therefore refer to an IBM technical reference. To transfer order data from your application memory to your presentation space, use the function called *GpiPutData.* If for some reason your order buffer does not contain a complete order, the GpiPutData function will not process the incomplete order and will return an offset of the incomplete order to your application so you can complete it for a subsequent put. If you would like to transfer order data from a graphic segment to application memory, use the function called *GpiGetData.* This function may also return a partial order. The graphic editor program that comes with this book uses neither of these functions.

Note that as we developed the graphic editor for this book, the object parsing feature was one of the first features we completed. We then used the object viewer to assist us during the remainder of our development. This approach worked great for us. It is our hope that you can use this tool to see how we created objects with the graphic editor by using the GPI, but furthermore, we believe you may find it worth while to use some of our parsing code to aid you in your application development.

ELEMENTS

An element is nothing more than a group of orders that is bracketed by a Begin Element order and an End Element order. Elements can not be contained within other elements. Therefore, there must be an End Element order between each Begin Element order in the order data stream. The purpose of elements is to allow you to index through groups of orders in a graphic segment for editing purposes. Shortly, we will discuss some GPI functions that allow you to edit elements, but first let's look at how elements are created.

The easiest way to create an element is by using the GPI drawing functions. It turns out that many of the GPI drawing functions automatically bracket the orders they produce with the Begin and End element orders if you have **not** explicitly started an element. For instance, if you issue the GpiCallSegmentMatrix function, it will generate the following orders:

1. Begin Element
2. Push and Set Model Transform
3. Call Segment
4. Pop
5. End Element

Another way you may generate an element is by issuing a function called *GpiBeginElement*. After you issue this function, you can issue the series of GPI drawing functions that you would like to group. Finally, issue the function called *GpiEndElement* to end the element. By producing the element in this way, the GPI drawing functions within the GpiBeginElement and GpiEndElement functions will not generate the begin/end element brackets like the method previously described. Hence, you can produce larger elements or order groupings that are more meaningful to your application.

Yet another function that lets you create elements is called *GpiElement*. The inputs to this function are the actual graphic orders you want to include in the element. Because nesting of elements is not allowed, you can not issue the GpiElement function within a GpiBeginElement/GpiEndElement bracket; and of course, the graphic orders you pass on this function should not include the begin or end element orders.

Finally, you could issue the GpiPutData function with all the graphic orders you want to place in segment store and include the begin and end element orders as part of this stream. But, as before, you must manage the placement of these begin and end element orders so there is no nesting.

Now that you have seen the different ways for and rules of generating elements, let's look a little closer at the Begin Element order. The Begin Element order can have a type and description associated with it. When you let the GPI drawing functions automatically generate elements, they will also set the element type. The GPI functions usually use the graphic order code for the element type. When you generate the Begin Element order yourself, you must specify your own element type and description. This element type, however, it must be in the range of X'81000000'–X'FFFFFFF'. Element types outside this user-defined range are reserved by OS/2. The element description you assign is nothing more than descriptive text that you would like to have associated with the element. To determine the type and description of an existing element, use the *GpiQueryElementType* function. (You will see how this may be used later.)

So, now that you know more about elements and the information associated with them, what's the information good for? Obviously, to help identify the content of the element! And of course, the reason you would want to identify an element is so that you can edit it within the graphic segment. The OS/2 graphic environment provides two modes in which you can edit graphic segments. These are `SEGEM_INSERT` and `SEGEM_REPLACE`, which stand for element insert mode and element replace mode, respectively. (Insert edit mode is the default.) Note that the edit mode is not an attribute of a particular segment but of the presentation space. Furthermore, you can change the edit mode anytime you like with a function called *GpiSetEditMode*. If you want to know what the current edit mode is for a presentation space, use the function called *GpiQueryEditMode*. But before you can edit a graphic segment by replacing or inserting elements, you need a way to move through the graphic segment to locate the different elements. A feature called an *element pointer* is the mechanism that allows you to move from element to element within a graphic segment. To understand how the element point works, we must jump ahead a little bit to graphic segments.

When a graphic segment is opened with a function called *GpiOpenSegment*, the element pointer is set to zero. If you want to add new elements to

the segment, you must insure your edit mode is SEGEM_INSERT. Then position the element pointer right before the location at which you want to add the new element. The way you move the element pointer is with a function called *GpiSetElementPointer* or *GpiOffsetElementPointer*. GpiSetElementPointer lets you set the element pointer to a specific location whereas GpiOffsetElementPointer lets you specify an offset from the current element pointer location. You can move the element pointer in both directions with these functions. If you specify an offset value that causes the element pointer to be out of range, the element pointer will be set to either 0 or the last element depending on if the offset was negative or positive. Likewise, if you specify an element pointer position with the GpiSetElementPointer function that is too large , the element pointer will be set to point to the last element in the graphic segment. Once you have located the element pointer correctly, you can then create the new element with any of the methods previously described except with GpiPutData. You can only use GpiPutData with the edit mode SEGEM_INSERT, but the element pointer must always be pointing at the last element in the segment. Hence, GpiPutData will only add elements to the end of a segment. Once you add an element, the element pointer is automatically incremented by 1.

To understand the structure of orders and elements in a graphic segment, examine Figure 5.1. The *before* part of Figure 5.1 shows a representation of a graphic segment that has just been opened and contains three elements. Notice that the element pointer is located at zero when the graphic segment is opened which doesn't really point to an element at all! The *after* part of Figure 5.1 shows a representation of adding a new element after element 2. To add this element, the element pointer was first positioned to element 2 and then the element was inserted after element 2. Notice that after the element is added the element pointer is automatically incremented to point to the new element 3.

It may seem odd to you that your element pointer is positioned to the element right before the insertion point instead of at the insertion point, but consider how a new segment is created! When you open a new segment with the same GpiOpenSegment function, the element pointer is still zero. Then, as you issue GPI drawing functions, the elements are automatically inserted in order using this auto-incrementing technique. Furthermore, the first element in a graphic segment is being pointed at when the element pointer is

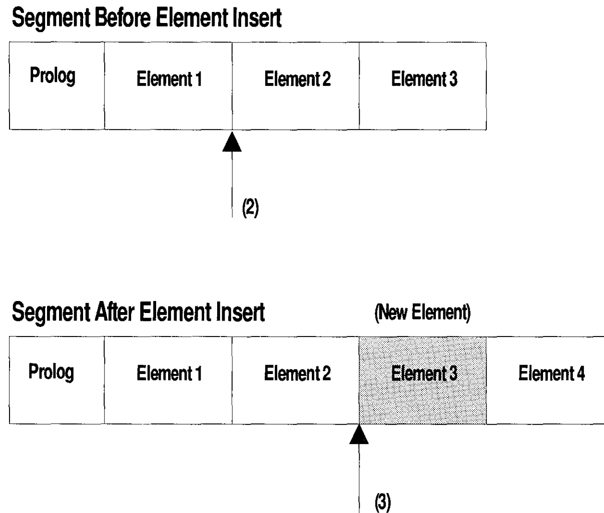


FIGURE 5.1 Inserting an element in a graphic segment.

one. Hence, when you add an element and the element pointer is set to zero, you are inserting the element at the very beginning of the segment.

When using SEGEM_REPLACE mode, however, the rules are a little different. When using replace mode, you position the element pointer to point at the element to be replaced. Therefore, using an element pointer of zero is invalid in replace mode. This also implies that when you open a new segment, you must ensure the edit mode is SEGEM_INSERT. If the edit mode is not SEGEM_INSERT for a new segment, an error will occur on the first element you try to add.

Now you can see how to move between elements in a graphic segment and how to add or replace elements; but unless we have intimate knowledge of the content of all elements, editing the segment may still seem very difficult and tedious. There are a couple GPI functions, however, that help you find out the content of an element or quickly locate the element pointer to a particular variable location in a graphic segment. First, when you generated your element, a type and description were associated with the element. By using the function GpiQueryElementType that we mentioned earlier, you can quickly determine the element type and description so you can identify if

it is an edit point. Second, you can put labels in your graphic segment with a function called *GpiLabel*. (A label turns out to be just a number or ID.) When you want to edit the graphic segment, you can use a function called *GpiSetElementPointerAtLabel* to position the element pointer at the label element you created. After the element pointer is at the label, you can use GpiOffsetElementPointer to reposition the element pointer as needed. Note that you can use the same label multiple times in a graphic segment. When you use the GpiSetElementPointerAtLabel function, it will set the element pointer to the first occurrence of the label from the elements pointer's current position.

Besides adding or replacing elements, the GPI also has a set of functions that allow you to delete elements from a graphic segment. These functions are called *GpiDeleteElement*, *GpiDeleteElementRange*, and *GpiDeleteElementsBetweenLabels*. As their function names imply, these functions allow you to delete a single element, a range of elements, or all elements between two labels.

If you want to see the order content of an element, you can use a function called *GpiQueryElement*. This function will return all the order data contained within the element (given that you provide a large enough buffer for the function). If you do not provide enough buffer space, as much of the element content is returned to you as possible. As you will see shortly, we use this function and many of the other element-related functions to provide the object viewing feature in our graphic editor.

SEGMENTS

When using Presentation Manager to display simple output to a display window, it is common to just issue the GPI drawing functions to interactively update the window. This works great for simple applications, but when you want to produce more complex drawings that you may want to manipulate, this doesn't work so well. Your first reaction to fix this situation may be to maintain your own list of drawing functions that you can update when necessary and then redraw. This technique may work well for you, but the GPI has a feature called *retained graphics* which is designed to help with just this situation. At the center of retained graphics is the notion of graphic segments and segment store.

So far, we've talked a lot about a graphic segment as a container of elements and orders, and we've started showing how graphic segments allow you to edit your drawing. However, we haven't really talked about how graphic segments come in to existence. After all, it is quite possible you have been using Presentation Manager for some time and have never created a single graphic segment. As mentioned in the overview, before you can use retained graphics (hence graphic segments), you must create a normal presentation space. Once you have a normal presentation space, you are just about ready to start using retained graphics and create graphic segments *if you want to*. As mentioned in the previous section, to create a new graphic segment or edit an existing one, you use a function called *GpiOpenSegment*. To end a graphic segment you've started, use a function called *GpiCloseSegment*. Too easy! But before a graphic segment is really created, you must make sure your drawing mode is set correctly.

By default, Presentation Manager will set the drawing mode of your presentation space to `DM_DRAW` when it is created. A drawing mode of `DM_DRAW` informs the GPI to route output to the associated device and not to segment store. It turns out, however, that opening a graphic segment and using GPI drawing functions while in `DM_DRAW` mode is a legal and useful thing to do. The graphic segment in this case is called a *nonretained graphic segment*. The usefulness of a nonretained graphic segment is that it will initialize primitive attributes just like a retained segment, as well as set and reset the view transform matrix just like a retained graphic segment. Furthermore, these nonretained graphic segments can be recorded in a MetaFile just like retained graphic segments (MetaFiles are discussed later in this book). Finally, if you want to convert these nonretained graphic segments to retained graphic segments, it's easy to do.

To have graphic drawing information saved in an opened graphic segment, you must set the drawing mode of the presentation space to `DM_RETAIN` or `DM_DRAWANDRETAIN`. If you use the `DM_RETAIN` drawing mode, the output is directed only to segment store and not to an associated output device. For this reason, you can use the `DM_RETAIN` drawing mode and not even have a device context associated with the presentation space. Also, if you want to edit a graphics segment, you must use the `DM_RETAIN` drawing mode. If you use the drawing mode `DM_DRAWANDRETAIN`, the output is directed to both the associated output device and segment store. This mode allows you to create new graphic segments; but, you can not edit

an existing graphic segment while using this mode. When editing unchained segments, you are only allowed to use a draw mode of `DM_RETAIN`.

To set the drawing mode for a presentation space, you can use a function called *GpiSetDrawingMode*. If you want to query which mode is currently active for the presentation space, you can use a function called *GpiQueryDrawingMode*. Before you create or edit a graphic segment, however, you need to be aware of the attributes associated with the segment. These graphic segment attributes are key in how the graphic segments can be used during picture construction and also how you can interact with your drawing for editing purposes. Following is a list of all the attributes that can be associated with a graphic segment and a brief description of their use:

ATTR_CHAINED – This segment attribute determines if the segment should be added to the presentation space segment chain. Therefore, as you create graphic segments with this attribute on, the segments are added to a segment list or chain. When the chain is drawn, the segments are drawn in the order in which they exist in the chain; hence, graphic segments have priority. Also implied by this attribute, graphic segments do not have to be part of the presentation space segment chain. An *unchained graphic segment*, however, can be drawn explicitly or called from within another graphic segment. The default value for this segment attribute is *on*.

ATTR_FASTCHAIN – This segment attribute is used to prevent the GPI from resetting primitive attributes before it draws each segment in the presentation space segment chain. This implies a performance improvement, but you must be careful to set the primitive attributes to the desired state before the chain is drawn. The default value for this segment attribute is on.

ATTR_DETECTABLE – This segment attribute allows your application to perform correlation operations with the segment. Correlation is a feature that lets you identify drawing primitives that intersect a special region of an area called a *pick aperture*. Correlation and how it is used is discussed later in this book. The default setting for this segment attribute is *off*.

ATTR_PROP_DETECTABLE – This segment attribute will propagate its `ATTR_DETECTABLE` value to segments that it calls. Hence, if the segment is set up for correlation, the graphic segments it calls will also be set up for correlation. The default value for this segment attribute is on.

ATTR_VISIBLE – This segment attribute allows the segment drawing primitives to generate output on the attached device. The default value for this segment attribute is on.

ATTR_PROP_VISIBLE – This segment attribute will propagate its ATTR_VISIBLE value to segments that it calls. The default value for this segment attribute is on.

ATTR_DYNAMIC – This segment attribute applies to chained segments and is used to inform the GPI to generate the segment output using the XOR raster operation. Note that the XOR raster operation is commonly used to move objects around within in a window without affecting the reset of the drawing. Hence, like the ATTR_DETECTABLE segment attribute, this attribute is designed to help you during the interactive editing of a drawing. The default value for this segment attribute is off.

Just reading about the graphic segment attributes probably gives you a lot of insight as to what you can do with segments. However, before we discuss how graphic segments can be used and how they relate to each other, let's see how we can query-and-set the segment attributes. To query how an attribute will be set when a graphic segment is first created, you can use a function called *GpiQueryInitialSegmentAttrs*. If you want to change the initial value of a graphic segment attribute for when new segments are created, you can use a function called *GpiSetInitialSegmentAttrs*. These two functions only deal with segment attributes for newly created graphic segments. If you want to query or change an attribute of an existing graphic segment, you can use the functions called *GpiQuerySegmentAttr* and *GpiSetSegmentAttrs*. All of these query-and-set segment attribute functions only operate on one segment attribute at a time.

Now let's look a little closer at what it means when a graphic segment is chained. If the ATTR_CHAINED segment attribute is on, newly created graphic segments are appended to the presentation space segment chain. (A graphic segment which is chained is sometimes called a *root segment*.) The GPI has functions that allow you to draw all or part of the presentation space segment chain. When all or part of the segment chain is drawn, it is done in the same order as the segments are found in the chain. Therefore, the last graphic segment in the segment chain has the highest priority. This may seem backward at first, but the last object drawn with the GPI will appear on top of the picture.

To draw the entire presentation space segment chain, you can use a function called *GpiDrawChain*. If you want to draw a range of root segments, you can use the function called *GpiDrawFrom*. Note that the GpiDrawFrom function specifies a range of segment IDs and not the relative location of segments in the chain. Furthermore, note that segment IDs must be unique and **do not** imply segment priority. Graphic segment ID values can be placed in any order you want in the segment chain. If you want to draw a particular graphic segment no matter if it is chained or not, you can use a function called *GpiDrawSegment*.

Because the order of the graphic segments found in the segment chain makes a difference in how the picture is construction, care must be taken in how the segment chain is constructed. The GPI does have functions, however, that allow you to interrogate segment IDs and change the priority of the segments in the segment chain. (This ought to give you a big clue as to how we implemented the "Front", "Back", "Forward 1", and "Backward 1" features in the graphic editor!) To query the graphic segment IDs in a range, you can use a function called *GpiQuerySegmentNames*. Again, this range is of segment IDs and not relative locations in the segment chain. Furthermore, the GpiQuerySegmentNames function returns both chained and unchained segment IDs. Nonretained graphic segments, however, are not returned by the GpiQuerySegmentNames function. To sort out the priority of segments in the segment chain, you can use a function called *GpiQuerySegmentPriority*. This function works only for chained segments. However, you may use this function to test if a graphic segment is in the segment chain by testing the error returned. Finally, you can use a function called *GpiSetSegmentPriority* to change the priority of a segment in the segment chain. You can also use the GpiSetSegmentPriority function to add an unchained graphic segment to the segment chain!

To use the GpiQuerySegmentPriority function, you must provide a reference graphic segment ID and a value (`LOWER_PRI` or `HIGHER_PRI`) to indicate which neighboring segment ID you want returned. If you end up querying for either the next lowest or the next highest graphic segment ID and it doesn't exist, a value of 0 is returned. This implies that the reference ID that you were using is the lowest or highest priority graphic segment ID in the chain. The GpiSetSegmentPriority function works in a similar way, but you must also supply the graphic segment ID for which you want to change priority. If you specify a value of `LOWER_PRI` for the GpiSetSeg-

mentPriority function, the graphic segment will be placed in the graphic segment chain one position lower in priority than the reference segment ID. Likewise, if you specify a value of `HIGHER_PRI` for the GpiSetSegmentPriority function, the graphic segment will be placed in the graphic segment chain one position higher in priority than the reference segment ID.

Maybe you've already guessed, but graphic segments that are not chained can be drawn directly from your application, called by other graphic segments, or called directly from your application. We've already seen how the GpiDrawSegment can be used to draw any chained or unchained segment on demand. But the function called *GpiCallSegmentMatrix* lets us call a graphic segment, provide a transformation to apply to that graphic segment, and then have that graphic segment return control after its completion; of course, a called graphic segment can call other graphic segments. Cool! Now you can begin to see how you can create one object and apply it several times to your picture!

To help visualize how chained and unchained graphic segments can relate to each other, look at Figure 5.2. As you can see by this example, the

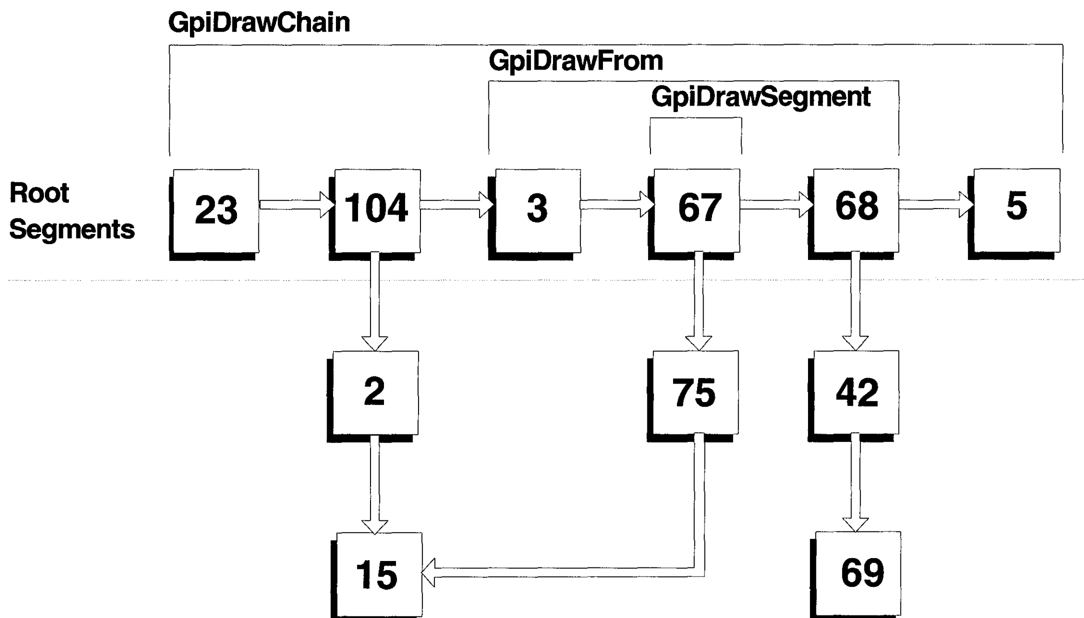


FIGURE 5.2 Chained and unchained graphic segments.

root or chained graphic segment IDs do not have to be in any particular order. When a GpiDrawChain function is issued, the lowest priority chained graphic segment starts to draw and all other root segments and their call segments are drawn in order until the highest priority graphic segment is drawn. Hence, in our example, the segment drawing order for the GpiDrawChain function is 23,104,2,15,3,67,75,15,68,42,69, and 5. When a GpiDrawSegment function is issued, the graphic segment specified and all the graphic segments it calls are drawn. Therefore, the order of graphics segments drawn in our example when you issue the GpiDrawSegment function is 67,75, and 15. Lastly, when a GpiDrawFrom function is issued, a subset of the segment chain is drawn. For our example, the order the segments are drawn in is 3,67,75, 15, 68, 42, and 69.

Another segment attribute you may want to use if you are interactively editing graphic segments is the `ATTR_DYNAMIC` attribute. This attribute is used by root or chained segments. If a graphic segment has the `ATTR_DYNAMIC` attribute on, it is considered a dynamic segment. When you have graphic segments that are dynamic, you can issue a function called *GpiDrawDynamics* which will draw a specified range of dynamic segments on top of the normal picture. The range of graphic segment IDs is defined by start and end graphic segment IDs that exist in your presentation space's segment chain. This range may include non-dynamic graphic segments; however, only dynamic graphic segments and the graphic segments they call are drawn. When the dynamic graphic segments are drawn with the GpiDrawDynamics function, the foreground mix is set to `FM_XOR` and the background mix is set to `BM_LEAVEALONE`. These mix attributes are set this way so when you use a function called *GpiRemoveDynamics*, the dynamic segments will appear to be erased from the picture without disturbing the original picture. Therefore, if you want to let a user interactively move some objects in a picture, you could set the root segments to be dynamic. Then you would use GpiDrawDynamics to bring the objects to the top of the normal picture. When the user starts a move operation via mouse movement, you would issue the GpiRemoveDynamics function, edit the dynamic segments to relocate the objects, and then use the GpiDrawDynamics function to redisplay the objects in the new location. You would repeat removing and redrawing to dynamic graphic segments until the user has indicated that a new permanent location has been found. Finally, you would issue one final GpiRemoveDynamics function to remove the objects from the top of the pic-

ture and redraw the segment chain. You have just interactively moved some objects!

When you use the GpiDrawDynamics function to draw a set of objects on top of the normal picture, you must take care to erase the objects with GpiRemoveDynamics function before you edit your graphic segments or use other draw functions on your presentation space. If you do not do this, the normal picture may appear disturbed until you erase and redraw your normal picture. Also, for better draw performance, it helps to have the dynamic graphic segments located as early in the chain as possible. As it turns out, we chose not to use dynamic graphic segments in our graphic editor because of how we use our own list structures. The technique we use to move objects is similar to that done with dynamic segments but our program logic manages the drawing of the mix attributes and segments. You may, however, find using dynamic graphic segments is an excellent design choice for your project.

From your previous experience programming Presentation Manager, you should already know that any intensive drawing activity should **not** be done in your main window procedure. This is because drawing complex items can be time consuming and you still need to service your main window procedure's message queue for real time events. If you don't do this, your application and OS/2 can appear to be sluggish and nonresponsive. Hence, drawing complex items in a background thread is a desirable design. In many cases, drawing the segment chain would be considered a time consuming task and should not be done in a window procedure's thread of execution. Because of OS/2's multithreaded capability, drawing the segment chain in a background thread is easy to do.

When drawing the segment chain in a background thread, you may find it desirable to interrupt the drawing so you can direct other operations to your presentation space. (Even though a presentation space can be shared by multiple threads, only one operation can be issued to the presentation space at a time.) A function call *GpiSetStopDraw* is designed just for this purpose. The GpiSetStopDraw function will cause the GpiDrawChain, GpiDrawDynamics, GpiDrawFrom, GpiDrawSegment, GpiPlayMetaFile, and GpiPutData functions to stop their current drawing operation to a presentation space if a value of `SDW_ON` is used as an input parameter. Once the GpiSetStopDraw function returns, issue the function again with a value of `SDW_OFF`, so future draw segment chain operations will work. One last

neat feature about the GpiSetStopDraw function is that if the GPI function that was interrupted was GpiDrawDynamics, the next GpiRemoveDynamics function will only erase the parts of the dynamic segments that were actually completed!

Well, from the description and rules we've just given, you should have a good idea of what graphic segments are and the type of things you can do with them. But like almost everything else in our world, there are exceptions to the rules. In the case of graphic segments, the rules are broken if the graphic segment ID is 0!

Unlike other graphic segments that must have a unique segment ID, you can have multiple graphic segments with an ID of 0 in your presentation space. A graphic segments with an ID of 0, however, can not be edited. Therefore, if you open a graphics segment with an ID of 0, you are actually creating a new instance of a graphic segment that will have an ID of 0. Following is a list of additional rules that apply to graphic segments with an ID of 0:

- A graphic segment with an ID of 0 must be a root or chained segment.
- Graphic segments with an ID of 0 cannot have their segment transform set or queried or their segment attributes set or queried.
- You can not use a graphics segment ID of 0 as an input parameter to the GpiDrawSegment or GpiDrawFrom functions. If graphic segments with an ID of 0 are found within the specified range for these GPI functions, however, they will be drawn and will contribute to boundary information. (Boundary accumulation is discussed later in this book.)
- You can not query or change the priority of a graphic segment with an ID of 0.
- Graphic segments with an ID of 0 will not participate in correlation operations. (Correlation is discussed later in this book.)

You may find that working with graphic segments that have an ID of 0 is not particularly useful, but you should be aware of their existence and rules. The graphic editor that comes with this book does not generate graphic segments with IDs of 0. One place where you may see graphic segments with IDs of 0, however, is in a MetaFile. (OS/2 MetaFiles are described later in this book.) This is because when a MetaFile is being created or recorded,

any drawing done outside of a graphic segment is automatically placed in a segment with an ID of 0 for the MetaFile.

The OS/2 GPI also has controls to help you manage drawing with graphic segments, as well as to aid you with interactive editing of retained graphics. These draw controls are maintained by your presentation space and can be queried with a function called *GpiQueryDrawControl* and can be set with a function called *GpiSetDrawControl*. The values for these draw controls are actually Boolean values and you can use the values DCTL_ON and DCTL_OFF when working with the different controls. Because these controls are maintained by the presentation space, they are given default values when the presentation space is created. Following is a list of the different draw controls available to your application and a brief description of how you may use each control:

DCTL_ERASE – When this control is on, an implicit erase of the presentation space is done before any GpiDrawSegment, GpiDrawFrom, or GpiDrawChain function is performed. The default state for this draw control is DCTL_OFF.

DCTL_DYNAMIC – When this control is on, the GPI automatically performs a GpiRemoveDynamics function before retained output is drawn and then automatically performs a GpiDrawDynamics after retained output is drawn. Hence, this control helps manage the drawing of dynamic segments when drawing normal retained output. The default state for this draw control is DCTL_OFF.

DCTL_DISPLAY – When this control is on, the GPI will generate output on the device specified by the device context. If the presentation space is not associated with a device context or if this draw control is off, then no output is generated. The default state for this draw control is DCTL_ON.

DCTL_CORRELATE – When this control is on, the GPI will perform correlation on any primitive or output generated with GpiElement or GpiPutData. (Correlation is discussed later in this book.) The default state for this draw control is DCTL_OFF.

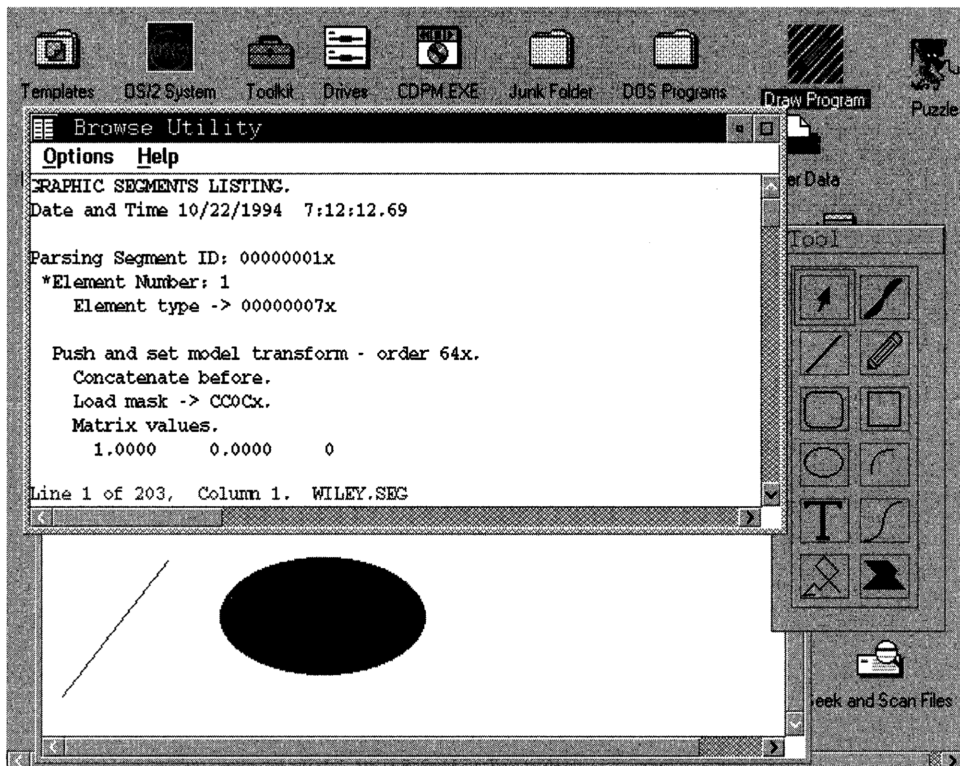
DCTL_BOUNDARY – When this control is on, the GPI will accumulate boundary data and calculate the smallest rectangular area that would contain output generated via retained drawing. (Boundary accumulation is discussed later in this book. You might have guessed that boundary accumulation is used by our graphic editor to produce the dashed rectangle around

selected objects or groups!) The default state for this draw control is DCTL_OFF.

Now that you have some background on what orders, elements, and graphic segments are, it's time to see how these structures apply in an example situation. Normally, a book of this type would just give a hypothetical example of how these structured are used to produce a drawing. In fact, our book will show example fragments of these structures as well. Our examples are different in that they were produced using part of the graphic editor provided with this book. Therefore, you can create your own example objects with our graphic editor and then interactively see what orders, elements, and segments were used to produce the actual output. Besides using the object viewer feature of the graphic editor to show you how orders, elements, and graphic segment structures work, we will also discuss how the object viewer was implemented. By reviewing the implementation of the object viewer, you will see many of the GPI functions discussed in this chapter in actual use. You will also develop a stronger understanding of how these structures relate to one another.

GRAPHIC EDITOR OBJECT VIEWER

When using the graphic editor that comes with this book, you can interactively view how selected objects in the picture are generated with the object viewer. To activate the object viewer, all you need to do is select the "View Selected Objects" menu item under the "File" pull-down menu. If you do not have any objects selected when this menu item is used, then all graphic segment data for the graphic editor presentation space is displayed in the Browse utility window. What is actually displayed in the Browse utility window when using the object viewer is a text file named WILEY.SEG. This file contains a description of all the graphic segments that were used to generate the selected objects. (In the case of no objects being selected, all the graphic segments known to the presentation space are displayed.) Screen 5.1 shows an example of what you would see if you used the object viewer. In this example, the graphic editor was used to produce a line and an ellipse, but neither of these objects was selected when the object view menu item was used. Because no objects were selected, all objects in for the



SCREEN 5.1 Object viewer.

graphic editor presentation space are displayed in the Browse utility window. Listing 5.1 shows the complete output produced by using the object viewer for the drawing shown in Screen 5.1.

```
GRAPHIC SEGMENTS LISTING. Date and Time 10/22/1994  7:12:12.69
Parsing Segment ID: 00000001x
   *Element Number: 1
   Element type -> 00000007x
   Push and set model transform - order 64x.
      Concatenate before.
      Load mask -> CC0Cx.
      Matrix values.
      1.0000      0.0000      0
      0.0000      1.0000      0
      0           0           1
      Call Segment - order 07x.
          Segment name -> 4.
      Pop - order 3Fx.
```

LISTING 5.1 Sample object viewer output.

```
    *Element Number: 2
    Element type -> 00000007x
    Push and set model transform - order 64x.
       Concatenate before.
       Load mask -> CC0Cx.
       Matrix values.
       1.0000      0.0000      0
       0.0000      1.0000      0
       0           0           1
    Call Segment - order 07x.
       Segment name -> 5.
    Pop - order 3Fx.

Parsing Segment ID: 00000002x
Parsing Segment ID: 00000003x
Parsing Segment ID: 00000004x*
    Element Number: 1
    Element type -> 00000023x
    Push and set pick identifier - order 23x.
       Pick identifier -> 0.
    *Element Number: 2
    Element type -> 00000001x
    Comment - order 01x.
    *Element Number: 3
    Element type -> 00000001x
    Comment - order 01x.
       Line
    *Element Number: 4
    Element type -> 81000000x
    Attributes
    Push and set individual attribute - order 54x.
       Attribute type -> color.
       Primitive type -> line.
       Set individual attribute.
       Use value directly.
       Attribute value -> 010000.
    Push and set stroke line width - order 55x.
       Set to value.
       Stroke width value -> 0.
    Push and set line type - order 58x.
       Line type -> solid line.
    Push and set individual attribute - order 54x.
       Attribute type -> color.
       Primitive type -> character.
       Set individual attribute.
       Use value directly.
       Attribute value -> 010000.
    Push and set character set - order 78x.
       Local identifier for character set ->  2.
    Push and set character cell - order 03x.
       X part of character cell-size attribute -> 24.
       Y part of character cell-size attribute -> 24.
       Fractional X part of character cell-size attribute -> 0.
```

LISTING 5.1 (Continued).

```
      Fractional Y part of character cell-size attribute -> 0.
      A cell size of 0 sets to 0.
   Push and set individual attribute - order 54x.
      Attribute type -> color.
      Primitive type -> pattern.
      Set individual attribute.
      Use value directly.
      Attribute value -> 020000.
   Push and set pattern symbol - order 09x.
      Value for pattern-symbol attribute -> 10.
      Solid shading.
      *Element Number: 5
      Element type -> 00000064x
   Push and set model transform - order 64x.
      Concatenate before.
      Load mask -> CC0Cx.
      Matrix values.
      1.0000      0.0000      0
      0.0000      1.0000      0
      19      36      1
   *Element Number: 6
   Element type -> 81000000x
      Drawing Details
   Push and set pick identifier - order 23x.
      Pick identifier -> 1.
   Set current position - order 21x.
      Position -> [0, 0].
   Lines starting at current position - order 81x.
      Line points. [104, 125]

   Parsing Segment ID: 00000005x
      *Element Number: 1
      Element type -> 00000023x
      Push and set pick identifier - order 23x.
         Pick identifier -> 0.
      *Element Number: 2
      Element type -> 00000001x
      Comment - order 01x.

      *Element Number: 3
      Element type -> 00000001x
      Comment - order 01x.
         Ellipse
      *Element Number: 4
      Element type -> 81000000x
         Attributes
      Push and set individual attribute - order 54x.
         Attribute type -> color.
         Primitive type -> line.
         Set individual attribute.
         Use value directly.
         Attribute value -> 010000.
      Push and set stroke line width - order 55x.
         Set to value.
```

LISTING 5.1 (Continued).

```
    Stroke width value -> 0.
Push and set line type - order 58x.
   Line type -> solid line.
   Push and set individual attribute - order 54x.
   Attribute type -> color.
   Primitive type -> character.
   Set individual attribute.
   Use value directly.
   Attribute value -> 010000.
Push and set character set - order 78x.
   Local identifier for character set ->  2.
Push and set character cell - order 03x.
   X part of character cell-size attribute -> 24.
   Y part of character cell-size attribute -> 24.
   Fractional X part of character cell-size attribute -> 0.
   Fractional Y part of character cell-size attribute -> 0.
   A cell size of 0 sets to 0.
Push and set individual attribute - order 54x.
   Attribute type -> color.
   Primitive type -> pattern.
   Set individual attribute.
   Use value directly.
   Attribute value -> 020000.
Push and set pattern symbol - order 09x.
   Value for pattern-symbol attribute -> 10.
   Solid shading.
*Element Number: 5
Element type -> 00000064x
Push and set model transform - order 64x.
   Concatenate before.
   Load mask -> CC0Cx.
   Matrix values.
   1.0000      0.0000      0
   0.0000      1.0000      0
   276         110         1
*Element Number: 6
Element type -> 81000000x
   Drawing Details
Push and set pick identifier - order 23x.
   Pick identifier -> 1.
Begin path - order D0x.
   Path identifier -> 1.
Set current position - order 21x.
   Position -> [0, 0].
Push and set arc parameter - order 62x.
   P = 101, Q = -54, R = 0, S = 0.
Full arc at current position - order 87x.
   Multiplier -> 1.0000
End path - order 7Fx.
Fill path - order D7x.
   Alternate mode.
   Do not modify before filling.
   Path identifier -> 1.
```

LISTING 5.1 (Continued).

```
         Set current position - order 21x.
            Position -> [0, 0].
         Push and set arc parameter - order 62x.
            P = 101, Q = -54, R = 0, S = 0.
         Full arc at current position - order 87x.
            Multiplier -> 1.0000

CHAIN SEGMENT ORDER
Segment ID 00000001x

END OF LISTING.
```

LISTING 5.1 (Continued).

From studying Listing 5.1, you can determine a lot about how these graphical objects were generated and even get a good idea about how the graphic editor uses graphic segments to produce its drawings. As you study Listing 5.1, you should notice how indentations are used to help parse up the segment information. When a new segment is started, the string "Parsing Segment" is printed followed by the segment ID. No indentation is used when the segment identifier text string is printed. After the segment identifier text, however, orders are indented 2 spaces and order information is indented 4 spaces. You may expect that the text line that marks the beginning of an element would be formatted the same as all the other orders in the listing, but they're not. (Example – Begin element – order D2x. Then element order data.) You might also notice that there are not any end element orders in the listing. The apparent lack of begin/end element order in the list is caused by the way element data is obtained (as you will see shortly). Also, as you look at this listing, you should not assume the segments are listed in any particular order. If all the graphic segments are parsed, however, the bottom of the listing will list the chained segments in priority order. This means lowest priority is listed first.

Because all the graphic segments in our example were parsed , look at the bottom of Listing 5.1 to see what segments were included in the presentation space segment chain. In this example, segment 1 was the only chained segment. Hence, if you look for segment 1 in our list (which happens to be at the very top of the listing) you can see the sequence of events used to generate the entire picture.

Segment 1 in our example contains only two elements. By comparing these two elements, you can see that they almost do the same thing. In particular, the elements push the current model transform, call another non-chained graphic segment, and then pop the model transform. The only

difference between the two elements in segment 1 is which graphic segments each calls. The first element calls graphic segment 4 and the second element calls graphic segment 5.

Graphic segment 4 contains several elements. The first element contains a push and set pick ID order. As you will see later in this book, pick IDs are used with correlation and are a way to help identify which part of a segment the user is selecting with the mouse. Elements 2 and 3 of graphic segment 4 contain comment orders. A comment order can really hold any type of information the application wishes to store in it. Because element 2 doesn't appear to have readable information stored in it, this comment order doesn't really give us much of a clue as to its value to the graphic segment. The comment order in element 3, however, does contain readable text and appears to be labeling the graphic segment as containing a line object. (In fact, this is the purpose of element 3.) Element 4 of graphic segment 4 contains a series of orders. What is interesting about these orders is that they all have to do with setting the attributes for the line that was drawn. Element 5 contains a push and set model transform order. You should already understand the concept of transformations, but later you will learn much more about transformation matrices. For now, however, knowing that this order is used for orientation of the object is all that is important. (Orientation includes rotation, shear angle, scaling, translation, and so on.) The last element in graphic segment 4, element 6, contains another set pick identifier order, a set position order, and a draw line order. Hence, element 6 is used to set up correlation for the line to be drawn and then draws the line.

As you study graphic segment 5 in Listing 5.1, you will notice the same basic flow of orders that were found in graphic segment 4. The only difference between graphic segments 4 and 5 is that one draws a line object and the other draws an ellipse! If we would have selected the line object before the graphic object viewer was used, then this listing would have only shown graphic segment 4. Furthermore, graphic segment 4 would also have included additional orders for drawing the dashed rectangle around the selected object and some markers.

Now that you know the basics of orders, elements, and segments, and you know how to use the object viewer, you can create objects with the graphic editor and then quickly see how these objects were constructed using the GPI. Again, we found this tool extremely helpful in debugging our own graphic editor! You may also find that including the object viewer as part of

your own graphical project is a worthwhile thing to do. This leads us into our next topic, reviewing the source code for the object viewer. By reviewing some of this code, you will see how many of the GPI functions discussed earlier in this chapter can be used. You will also see how you may modify or use the object viewer code for your own purposes.

First, let's review how the graphic editor interfaces with the object viewer logic. Listing 5.2 shows a small subset of the code found in DRAW.C. The first section of code shown is the logic that deals with the "View Selected Objects" pull-down menu item (IDM_VIEW). As you can see, this logic first allocates variables that will be used to identify the number of segments found in the graphic editor select list and then points to an array which will contain the IDs of all the segments in the select list. The IDM_VIEW menu item is then disabled so it can not be used again until the browsing of the object data is complete. Once the menu item is disabled, the select list is interrogated to get the number of objects that it contains and then memory is allocated to save the segment IDs that are contained in the select list. Once the memory is allocated, the select list is walked through to fill in the memory just allocated with segment IDs. Finally, the function called ParseSeg is used to perform the actual segment parsing. The items passed to the ParseSeg function are a handle to the presentation space where the objects are located, a pointer to a string which is the filename the parser will use to save ASCII text, a pointer to the array of segment IDs, and a count for the number of segment IDs that are found in the segment ID array. (Note the filename passed to the ParseSeg function is always WILEY.SEG.) The return from the ParseSeg function is a Boolean indicator which designates the success or failure of the function. As you can see, if the function succeeds, another function called OutWindow is used. If the ParseSeg function fails, however, the IDM_VIEW menu item is re-enabled and a message box is displayed describing the error condition.

The OutWindow function used in the IDM_VIEW menu item logic enables a file to be passed to the Browse utility and display. As you can see, the OutWindow function requires two parameters in order to use it. The first parameter is the handle to our window procedure and is used by the OutWindow function to notify us when the Browse utility is done displaying the file we passed to it. The message it sends to our window procedure when it is done is WM_ENDBROWSE. (As you may have already guessed, the reason a message is passed to our window procedure is because the Browse utility

runs in its own thread of execution.) The second parameter for the OutWindow function is a pointer to a filename that is to be displayed by the Browse utility.

The last part of Listing 5.2 shows the logic used when the OutWindow function sends the WM_ENDBROWSE message to our window procedure. In this case, the processing is to merely re-enable the IDM_VIEW menu item. Too easy!

```
/********************************/
/* Process pull-down menu items. */
/********************************/
case WM_COMMAND:
   switch (COMMANDMSG(&msg)->cmd){

      /***********************************************/
      /* Process view graphic objects pull-down option. */
      /***********************************************/
      case IDM_VIEW:
         {
            int segmentCount, i;
            LONG (*segmentArray)[];
            GOBJ selectedObj;
            WinSendMsg(WinWindowFromID(hwndFrame,FID_MENU),
               MM_SETITEMATTR,
               MPFROM2SHORT(IDM_VIEW,TRUE),
                MPFROM2SHORT(MIA_DISABLED,MIA_DISABLED));
            segmentCount=selectList->Do->GetCount(selectList);
            segmentArray=malloc(sizeof(LONG)*segmentCount);
            selectList->Do->Top(selectList);
            for (i=0;i<segmentCount;i++) {
               selectList->Do->GetNext(selectList,&selectedObj);
               (*segmentArray)[i]=selectedObj->Do->GetSeg(selectedObj);
            }
            if (ParseSeg(hps,segFile,*segmentArray,segmentCount)) {
               OutWindow(hwnd,segFile);
            }
            else {
               WinSendMsg(WinWindowFromID(hwndFrame,FID_MENU),
                  MM_SETITEMATTR,
                  MPFROM2SHORT(IDM_VIEW,TRUE),
                  MPFROM2SHORT(MIA_DISABLED,0));
               WinMessageBox(HWND_DESKTOP,HWND_DESKTOP,
                  (PSZ)"List file is not available for use.",
                  (PSZ)"Browse Error",
                  1, MB_OK | MB_APPLMODAL | MB_MOVEABLE);
               }
               free(segmentArray);
            }
            break;
```

LISTING 5.2 Object viewer interface to Browse utility.

```
      /**************************************/
      /* Default processing for WM_COMMAND. */
      /**************************************/
      default:
         return WinDefWindowProc(hwnd,msg,mp1,mp2);
   }
   break;
/********************************/
/* Process END Browse message. */
/********************************/
case WM_ENDBROWSE:
   WinSendMsg(WinWindowFromID(hwndFrame,FID_MENU),
      MM_SETITEMATTR,
       MPFROM2SHORT(IDM_VIEW,TRUE),
      MPFROM2SHORT(MIA_DISABLED,0));
   return(MRESULT)FALSE;
```

LISTING 5.2 (Continued).

Now that we've seen how the ParseSeg object interfaces with our main Draw program, it's time to look at the logic in the ParseSeg program object. Listing 5.3 shows the source code found in PARSESEG.C. Once you read past the defines, includes, and local data, you see that the first thing done by the ParseSeg function is to allocate and initialize two memory areas that will hold graphic segment IDs. One memory area will hold all graphic segments IDs, while the other will hold only chained graphic segment IDs. Once these two memory areas are initialized, a third area is allocated to hold element data. The ParseSeg function then attempts to open the filename that was passed to it. From trying to open this file, the ParseSeg function can collect return code information about the status of the file. If the filename is available for use, the ParseSeg function will close and recreate the filename for storing parsed graphic segment information. Otherwise, the ParseSeg function will return a FALSE indicator to the caller.

Assuming the filename for storing graphic segment information is created, the ParseSeg function will then test to see how many graphic segment IDs were passed to it by the caller. If no IDs were passed, the ParseSeg function will query up to the first 5000 graphic segment IDs into the SegIDs data area. This is done by using the GpiQuerySegmentNames function. If the caller did pass some graphic segment IDs to the ParseSeg function, then these IDs are copied to the SegIDs data area. Once the number of IDs to work with is known and the graphics segment IDs are placed in the SegIDs data area, the ParseSeg function will begin to generate textual information to write to the created file.

The first information to be written to the file is the title for the listing followed by the current date and time. Then the ParseSeg function goes into a loop and parses each graphic segment. The first thing done is to create a heading for the graphic segment ID that is about to be parsed. After this heading is written to the file, the graphic segment ID is opened with the GpiOpenSegment function and the element pointer is set to point to the first element with the GpiSetElementPointer function. The current element pointer position is then queried with the GpiQueryElementPointer function. By testing the queried location of the element pointer with what location we explicitly set the pointer to, we can tell when we have reached the end of all elements in a graphic segment. The explicit setting will be one greater than the actual pointer position. (The pointer position will never go past the last element in a segment.) Then, with the element pointer pointing to the element about to be parsed, the ParseSeg function uses GpiQueryElementType to retreive the element description and GpiQueryElement to retrieve the element data. (Element data includes all the data encapsulated between the begin and end element orders.) Once the data has been queried, the ParseSeg function writes element information to the file and then proceeds to parse the order data.

```
/**********************************************************************/
/* Parse graphic segments.                                            */
/* Copyright (c) 1994 John Wiley & Sons, Inc.  All rights reserved.   */
/* Reproduction or translation of this work beyond that permitted in  */
/* Section 117 of the 1976 United States Copyright Act without the    */
/* express written permisson of the copyright owner is unlawful.      */
/* Request for further information should be addressed to the         */
/* Permission Department, John Wiley & Sons, Inc.  The purchaser may  */
/* make back-up copies for his/her own use only and not for distribution*/
/* or resale.  The Publisher assumes no responsibility for error,     */
/* omissions, or damages, caused by the use of these programs of from */
/* the use on the information contained herein.                       */
/**********************************************************************/
   #define INCL_DOSDATETIME
   #define INCL_DOSFILEMGR
   #define INCL_WINHELP
   #define INCL_DOS
   #define INCL_WIN
   #define INCL_GPI
   #define INCL_PM
   #define INCL_GPIBITMAPS
   #define INCL_DOSMEMMGR
   #define INCL_GPISEGEDITING
```

LISTING 5.3 ParseSeg routine.

```
   #define OPEN_FILE 0x01
   #define CREATE_FILE 0X10
   #define FILE_ARCHIVE 0x20
   #define DASD_FLAG 0
   #define INHERIT 0x80
   #define WRITE_THRU 0
   #define FAIL_FLAG 0
   #define SHARE_FLAG 0x10
   #define ACCESS_FLAG 0x02
   #define FILE_SIZE 0L
   #define EABUF 0
   #define MAXIDS 5000
   #define ELEMENTSIZE 5000
   #define INITID 1
   #define ENDID 0x7FFFFFFF
   #include <os2.h>
   #include <stdio.h>
   #include <math.h>
   #include <stdlib.h>
   #include <string.h>
   #include <pmerr.h>
   #include "parseseg.h"
   #include "porders.h"
   #include "ordertab.c"
/***********************************/
/* Function parse graphic segment. */
/***********************************/
BOOL ParseSeg(HPS hpsCaller,CHAR *filename,LONG *segments,INT numSegments){
   LONG *segChain;
   LONG *segIDs;
   BYTE *elementData;
   static CHAR descrBuf[64];
   static CHAR fileData[256];
   DATETIME dateTime;
   LONG elementType,elementLen,descrLen=64,elementPtr,currentElementPtr;
   HFILE fileHandle;
   ULONG wrote,action,rc;
   INT stringlen,i,G=4;
   LONG segIDCnt,index,link,segVal,maxcnt,elementDataLen,
      linecnt,orderOffset,orderAdder,blankcnt,chainCnt;
   BOOL funcRet;
/****************************************/
/* Start of output to listing function. */
/****************************************/
segChain=malloc(sizeof(LONG)*MAXIDS);
segIDs=malloc(sizeof(LONG)*MAXIDS);
memset(segChain,0,sizeof(LONG)*MAXIDS);
memset(segIDs,0,sizeof(LONG)*MAXIDS);
elementData=malloc(ELEMENTSIZE);
/*****************************************/
/* Open file that was typed or selected. */
/*****************************************/
```

LISTING 5.3 (Continued).

```
action=2;
rc=DosOpen(filename,&fileHandle,
   &action,FILE_SIZE,FILE_NORMAL,1,0x40,EABUF);
/************************************************************************/
/* If file was available, not found, or explicit open failed,           */
/* then continue to create the list file; otherwise, file not available. */
/************************************************************************/
if((rc==0) || (rc==2) || (rc==110)){
   DosClose(fileHandle);
   DosOpen(filename,&fileHandle,&action,FILE_SIZE,FILE_ARCHIVE,
      OPEN_FILE | CREATE_FILE,
      DASD_FLAG | INHERIT | WRITE_THRU |
      FAIL_FLAG | SHARE_FLAG | ACCESS_FLAG, EABUF);
   DosSetFileSize(fileHandle,FILE_SIZE);
/*********************************************************************/
/* If segment count from client is 0 then get IDs from PS, else copy */
/* segment IDs from client list.                                     */
/*********************************************************************/
if(numSegments==0){
   segIDCnt=GpiQuerySegmentNames(hpsCaller,INITID,ENDID,MAXIDS,segIDs);
   }
else {
   segIDCnt=numSegments;
   for(i=0;i<segIDCnt;i++)segIDs[i]=segments[i];
   }
index=0;
/*****************************************************/
/* Print out heading title and time info for listing. */
/*****************************************************/
stringlen=sprintf(fileData,"GRAPHIC SEGMENTS LISTING. \n");
DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
DosGetDateTime(&dateTime);
stringlen=sprintf(fileData,"Date and Time %d/%d/%d  %d:%d:%d.%d \n",
   dateTime.month,
   dateTime.day,
   dateTime.year,
   dateTime.hours,
   dateTime.minutes,
   dateTime.seconds,
   dateTime.hundredths
   );
DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
/*******************/
/* Parse segments. */
/*******************/
 while(index<segIDCnt){
   stringlen=sprintf(fileData,"\nParsing Segment ID: %08Xx\n",
      segIDs[index]);
   DosWrite(fileHandle,(PVOID)fileData,stringlen,&wrote);
   GpiOpenSegment(hpsCaller,segIDs[index]);
   elementPtr=1;
   GpiSetElementPointer(hpsCaller,elementPtr);
   currentElementPtr=GpiQueryElementPointer(hpsCaller);
```

LISTING 5.3 (Continued).

```
   /***********************************/
   /* Parse all data in all elements. */
   /***********************************/
   while(elementPtr==currentElementPtr){
      elementLen=GpiQueryElementType(hpsCaller,&elementType,
         descrLen,descrBuf);
      elementDataLen=GpiQueryElement(hpsCaller,0L,ELEMENTSIZE,elementData);
       stringlen=sprintf(fileData,
      " *Element Number: %d\n",currentElementPtr);
       DosWrite(fileHandle,(PVOID)fileData,stringlen,&wrote);
      stringlen=sprintf(fileData,
      "    Element type -> %08Xx\n    ",
      elementType);
      DosWrite(fileHandle,(PVOID)fileData,stringlen,&wrote);
       DosWrite(fileHandle,descrBuf,strlen(descrBuf),&wrote);
      sprintf(fileData,"\n");
       DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote); or-
derOffset=0;
      /***************/
      /* Parse data. */
      /***************/
      while(elementDataLen){
         /*********************************************/
         /* Determine if this is a single-byte order. */
         /*********************************************/
         if((elementData[orderOffset]==0) ||
            (elementData[orderOffset]==255)){
            orderAdder=1;
            parseTab[elementData[orderOffset]]
               (fileHandle,elementData+orderOffset,G);
            }
         /***********************************/
         /* Determine if this is an extended */
         /* order and print its data.        */
         /***********************************/
         if(elementData[orderOffset]==254){
            orderAdder=(elementData[orderOffset+2])*256;
            orderAdder=orderAdder+(elementData[orderOffset+3]);
            orderAdder=orderAdder+4;
            parseTab[elementData[orderOffset]]
               (fileHandle,elementData+orderOffset,G);
            }
         /**********************************/
         /* Determine if this is a two      */
         /* byte order and print its data.  */
         /**********************************/
         if(   (elementData[orderOffset]&8) &&
            (elementData[orderOffset]<=127)){
         orderAdder=2;
         parseTab[elementData[orderOffset]]
            (fileHandle,elementData+orderOffset,G);
         }
```

LISTING 5.3 (Continued).

```
          /******************************/
          /* Determine if this is a long */
          /* order and print its data.   */
          /******************************/
          if(   (elementData[orderOffset]!=0) &&
             (elementData[orderOffset]!=255) &&
             (elementData[orderOffset]!=254) &&
             ( !((elementData[orderOffset]<=127) &&
             (elementData[orderOffset]&8)) )){
          orderAdder=elementData[orderOffset+1]+2;
          if(elementDataLen<orderAdder){
             sprintf(&fileData[0]," DATA LENGTH PROBLEM!!\n");
              DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
             break;
             }
          else parseTab[elementData[orderOffset]]
             (fileHandle,elementData+orderOffset,G);
          }
       elementDataLen=elementDataLen-orderAdder;
       orderOffset=orderOffset+orderAdder;
    }
    elementPtr++;
    GpiSetElementPointer(hpsCaller,elementPtr);
    currentElementPtr=GpiQueryElementPointer(hpsCaller);
 }
 GpiCloseSegment(hpsCaller);
 index++;
 }
 /*********************************************************************/
 /* If segments did not come from client list, then order and list them.*/
 /*********************************************************************/
 if(numSegments==0){
    /**************************************/
    /* Find lowest segment value in chain. */
    /**************************************/
    chainCnt=0;
    for (index=0;index<segIDCnt;index++) {
       segVal=GpiQuerySegmentPriority(hpsCaller,segIDs[index],LOWER_PRI);
       if (segVal==0){
          link=segIDs[index];
          chainCnt++;
          }
       if (segVal!=GPI_ALTERROR) segIDs[index]=0;
       }
       /*********************************************/
       /* Order the segments from lowest to highest. */
       /*********************************************/
       if(chainCnt!=0){
          index=0;
          segChain[index]=link;
          while (GpiQuerySegmentPriority(hpsCaller,link,HIGHER_PRI)!=0){
             link=GpiQuerySegmentPriority(hpsCaller,link,HIGHER_PRI);
```

LISTING 5.3 (Continued).

```
            segChain[++index]=link;
            }
         segChain[++index]=GpQuerySegmentPriority(hpsCaller,
            link,HIGHER_PRI);
         }
      stringlen=sprintf(fileData,"\n\nCHAIN SEGMENT ORDER\n\n");
      DosWrite(fileHandle,(PVOID)fileData,stringlen,&wrote);
      index=0;
      while (segChain[index]!=0){
         stringlen=sprintf(fileData,"  Segment ID %08Xx\n",segChain[index]);
         DosWrite(fileHandle,(PVOID)fileData,stringlen,&wrote);
         index++;
         }
      }
   /**********************************/
   /* Print out end note for listing. */
   /**********************************/
   stringlen=sprintf(fileData,"\nEND OF LISTING. \n");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   rc=DosClose(fileHandle);
   funcRet=TRUE;
   }
else
   funcRet=FALSE;
free(segChain);
free(segIDs);
free(elementData);
return funcRet;
}
```

LISTING 5.3 (Continued).

As you can see in Listing 5.3, order data is parsed while element data has not been exhausted. When the order parser loop is entered, the offset (orderOffset) into the element data (elementData) always points to the beginning of a graphic order. By interrogating the order found at this offest, the ParseSeg function determines if the order is single-byte, extended, double-byte, or long order. Once the order type is discovered, a variable called orderAdder is set with the length of the order. This length is later used at the bottom of the order parsing loop to adjust the offset variable to point to the next order in the element data and to adjust the length of element data that still needs to be parsed. Once the orderAdder varible is set with the correct length, a function that actually parses the order data is then called through a branch table called parseTab. This table has 256 entries and contains the addresses of the functions that are used to parse the graphic orders. As you can see, each function takes three input parameters. These input parameters are a file handle, a pointer to the order to be parsed, and a value called G. The file handle should be to an opened file where the order parser can write

textual data; furthermore, this file should have its file pointer set to the position where data can be written. G is used to indicate the type for some of the data that may contained in the order data. This type is really determined by how the presentation space was created to hold coordinate information (GPIF_LONG or GPIF_SHORT). In the case of our graphic editor, a coordinate format of GPIF_LONG was used (which is indicated by G being set to 4). To get a sense of how this branch table works, Listing 5.3 shows the first 16 entries of the branch table called parseTab. ORDERTAB.C contains all 256 entries for the branch table. Then, if you look at Listing 5.4, you will see an example of a few of the functions that actually parse order data. PORDERS.C contains the source code for all the graphic order parsing functions. These order parsing functions are not particularly difficult to understand and are not going to be discussed in this book. If you would like to understand the structure of an actual order, however, you may look at these functions or refer to an IBM technical reference.

Once all the graphic orders for all the elements for all the graphic segments have been parsed, the ParseSeg function then checks again to see if the user passed specific graphic segments to be parsed or if the entire presentation space is to be parsed. If the entire presentation space was parsed, the ParseSeg function proceeds to locate the lowest priority graphic segment in the segIDs array. Finding the lowest priority graphic segment is done by using the GpiQuerySegmentPriority function. We actually use every segment ID as a reference and look for a lower priority segment. If the GpiQuerySegmentPriority function ever returns 0, we know we have found the lowest priority graphic segment in the segment chain. The variable called *ChainCnt* is used only as an indicator that we found the lowest priority graphic segment and the variable called *link* is used to save the graphic segment ID. If a low priority graphic segment was found, the graphic segments in the segment chain are then ordered in the segChain data area. The GpiQuerySegmentPriority function is also used to create our sorted list by saving the lowest priority graphic segment in our sorted list, and then using this graphic segment ID as a reference ID to find the next highest priority segment ID. We then save the returned segment ID as our next reference segment ID for the GpiQuerySegmentPriority function, as well as save the ID in our sorted list. We continue to loop finding the next highest priority segment ID until we find the highest priority segment. This segment is then placed in the sorted list and the ParseSeg function prepares to print the list content.

To print the chain order, the ParseSeg function first generates a header to indicate that the segment chain follows and then outputs the segment ID found in the sorted list. Finally, this function prints out one last message indicating the list is complete, frees up allocated memory, and then returns to the caller. Now we have our object viewer!

```
static ParseFunc parseTab[]={
   PNOP,
   PComment,
   PReturn,
   PSetCharCell,
   PSetSeg,
   PSetBrkE,
   PReturn,
   PCallSeg,
   PSetPat,
   PPushPatSym,
   PSetCol,
   PReturn,
   PSetMix,
   PSetBackMix,
   PReturn,
   PReturn  // Real code has 256 entries.};
/********************/
/* Parse NOP order. */
/********************/
void FAR PNOP(HFILE handle, CHAR *pData, int G)
   {
   CHAR fileData[256];
   ULONG wrote;
   int length;
   sprintf(fileData,"  No operation - order 00x.\n");
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   return;
   }
/************************/
/* Parse comment order. */
/************************/
void FAR PComment(HFILE handle, CHAR *pData, int G)
   {
   CHAR fileData[256];
   ULONG wrote;
   int length;
   sprintf(fileData,"  Comment - order 01x.\n    ");
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   length=pData[1];
   strncpy(fileData,pData+2,length);
   fileData[length]=0;
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   sprintf(fileData,"\n");
```

LISTING 5.4 Order parser routines.

```
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   return;
   }
/*************************************************************/
/* This order parser is used when the order found is unknown. */
/*************************************************************/
void FAR PReturn(HFILE handle, CHAR *pData, int G)
   {
   CHAR fileData[256];
   ULONG wrote;
   sprintf(fileData,"  Unknown order - order %02Xx.\n",pData[0]);
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   return;
   }
/*********************************************/
/* Parse (push and) set character cell order. */
/*********************************************/
void FAR PSetCharCell(HFILE handle, CHAR *pData, int G)
   {
   CHAR fileData[256];
   ULONG wrote;
   if(pData[0]==3)
      sprintf(fileData,"  Push and set character cell - order 03x.\n");
    else
      sprintf(fileData,"  Set character cell - order 33x.\n");
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
    if(G==4){
      sprintf(fileData,"    X part of character cell-size attribute
-> %d.\n",
         *((PLONG)(pData+2)));
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   sprintf(fileData,"    Y part of character cell-size attribute
-> %d.\n",
      *((PLONG)(pData+6)));
   DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
   if(pData[1]>10){
      sprintf(fileData,"    Fractional X part of character cell-size attribute
-> %d.\n",
         *((PSHORT)(pData+10)));
      DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
      sprintf(fileData,"Fractional Y part of character cell-size attribute
-> %d.\n",
         *((PSHORT)(pData+12)));
      DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
      if(pData[1]>12){
         if(pData[14]&'\x80')
            sprintf(fileData,"    A cell size of 0 sets to 0.\n");
         else
            sprintf(fileData,"    A cell size of 0 sets drawing default.\n");
         DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
         }
      }
```

LISTING 5.4 (Continued).

```
    else {
       if(pData[1]>8){
          if(pData[10]&'\x80')
             sprintf(fileData,"    A cell size of 0 sets to 0.\n");
          else
             sprintf(fileData,"    A cell size of 0 sets drawing default.\n");
           DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
          }
       }
    }
 else {
    sprintf(fileData,"    X part of character cell-size attribute -> %d.\n",
       *((PSHORT)(pData+2)));
    DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
    sprintf(fileData,"    Y part of character cell-size attribute -> %d.\n",
       *((PSHORT)(pData+4)));
    DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
       if(pData[1]>6){
          sprintf(fileData,"    Fractional X part of character cell-size attribute
          -> %d.\n",
             *((PSHORT)(pData+6)));
       DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
          sprintf(fileData,"    Fractional Y part of character cell-size attribute
          -> %d.\n",
             *((PSHORT)(pData+8)));
       DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
       if(pData[1]>8){if(pData[10]&'\x80')
          sprintf(fileData,"    A cell size of 0 sets to 0.\n");
       else
          sprintf(fileData,"    A cell size of 0 sets drawing default.\n");
       DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
       }
    }
 else {
    if(pData[1]>4){
       if(pData[6]&'\x80')
          sprintf(fileData,"    A cell size of 0 sets to 0.\n");
       else
          sprintf(fileData,"    A cell size of 0 sets drawing default.\n");
       DosWrite(handle,(PVOID)fileData,strlen(fileData),&wrote);
       }
    }
 }
 return;
 }
```

LISTING 5.4 (Continued).

GRAPHIC SEGMENTS IN THE GRAPHIC EDITOR

The graphic editor makes heavy use of graphic segments. Each object drawn using the editor is stored in its own retained segment. There are also three

additional segments that represent special lists in the editor. One of these segments is the display list segment, which contains calls to the segments of each object that is currently visible on the screen. The display list segment is the only root segment in the segment chain; thus, a GpiDrawChain can still be used to draw all visible objects.

Another list segment is called the paste list segment, which contains calls to objects that have been removed (by a cut operation) from the display list and placed in the paste list. This is an unchained segment; therefore, drawing the chain will not result in these segments being displayed.

A final list segment is called the select list segment, which contains references to all graphic objects that are currently selected. Segments referred to by the select list also are referred to by the display list. This list segment is not displayed by drawing the chain; in fact, its purpose is only for keeping track of which objects are selected. It is not used for display purposes (much like the paste list segment).

Due to the way the segments are organized, there is basically a two-level segment hierarchy with the top of the hierarchy being the list segments. Each list segment can then have one or more child segment that represents the objects in that list. Object segments typically make no calls to other segments. The one exception to this rule, however, is group objects. Each group object actually points to another list segment, which then contains calls to all the segments of the graphic objects that are in the group. Because group objects can contain other group objects, the actual depth of the segment hierarchy can grow arbitrarily deep. Figure 5.3 shows an example of the segment hierarchy.

In fact, a general purpose list object is used to support all the list segments described above. This list object keeps an in-memory list that points to the graphic objects in the list; plus, an associated segment is maintained that contains calls to the corresponding graphic objects segment. Therefore, by adding a graphic object to the list, the lists segment is automatically updated to contain a call to the segment of the newly added graphic object. This is done by editing the list segment and inserting a call to the new object segment. Likewise, if an object is removed from a list, the list segment is edited and the call to that object's segment is deleted.

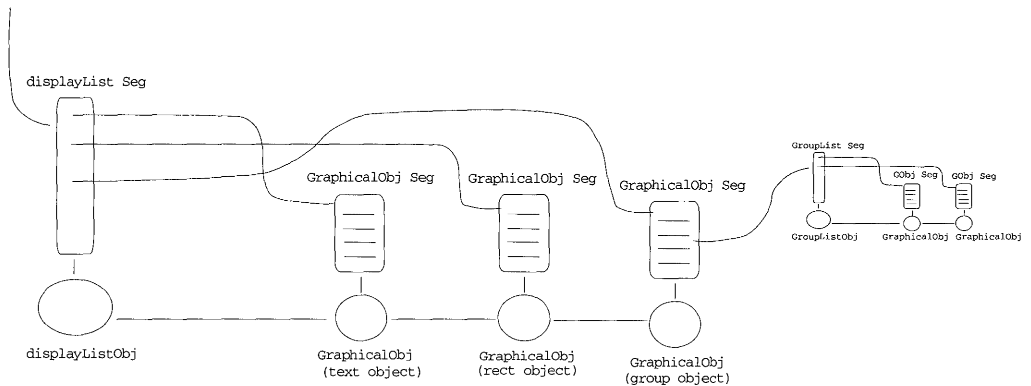


FIGURE 5.3 Segment hierarchy in the graphic editor.

Segments are also important in letting the user interact with the objects on the display. For example, after the user has drawn several items, he can go back and select one of those items for editing. This selection involves the use of retained segments and the use of correlation. See Chapter 9 for more details on how correlation works and how we use it in the graphic editor.

CHAPTER 6

Transformations

One of the frequently misunderstood areas of the GPI is that of coordinate space mapping. Have you ever wanted to draw an item (whose physical dimensions are known) on to the screen but found yourself trying to figure out how many pels wide or high it should be? If so, you're not alone. Wouldn't it be nice if the GPI would just let you draw the picture using whatever measurements you wanted? So, if you knew the dimensions of an object in inches, then you could just draw it in inches. Or, if the dimensions were in feet, you could draw it in feet (or miles, or millimeters, or whatever...). Well, the GPI does provide this capability! By proper use of the viewing pipeline, applications can draw pictures in whatever units they wish. In fact, they can draw various parts of their pictures using different units and still have them fit into the overall picture with the right scale.

As mentioned in Chapter 2, the viewing pipeline is a set of coordinate spaces that pictures go through from the time they are drawn until the time they appear on the device. Pictures are typically drawn into the first coordinate space, which is called the *world coordinate space*. Pictures are then transferred from one coordinate space to another by what is called a *transformation*. Each movement between coordinate spaces is accomplished by a different transformation. Finally, after passing through all the coordinate spaces, the picture is left in the device coordinate space. It is now in units that can be drawn to the output device (for example, pels for the display). These

transformations are all performed in single operation, rather than a series of operations. It is easier to understand, however, if we think of these transformations as being done separately.

To illustrate the use of transformations, imagine that you want to draw a picture showing the floor plan of your dream house. Since this is your dream house, you know all of its specifications (wall dimensions, heating/cooling vents, plumbing, furniture, etc.). The units of measure for each of these floor plan items will vary (walls in feet, furniture in inches, and so on). Let's see how we might use transformations to construct such a floor plan. Suppose the picture in Figure 6.1 is the floor plan you want to construct (it may not look like your dream home but for our purposes it will do).

Transformations let you translate or move a picture from one position to another. This transformation can be used to simplify the drawing of complex pictures. By breaking the complex picture down into subpictures, each subpicture can be drawn at a convenient origin and then moved to its correct position through the use of a transformation. Using the floor plan example above, you could draw each peice of furniture separately, each time drawing

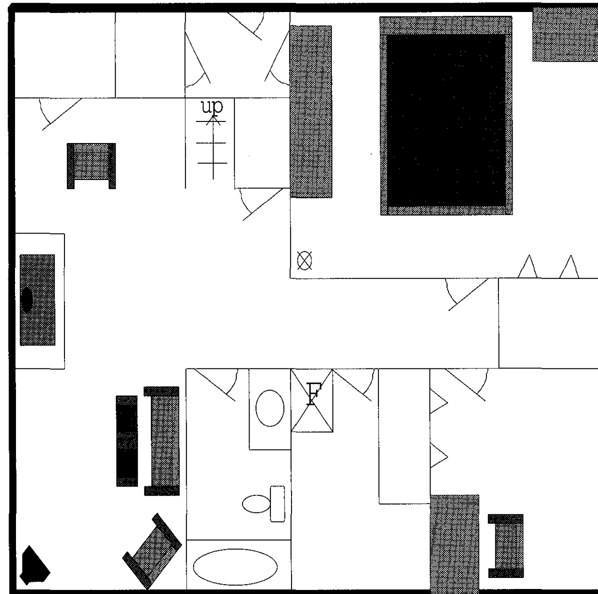


FIGURE 6.1 Desired floor plan.

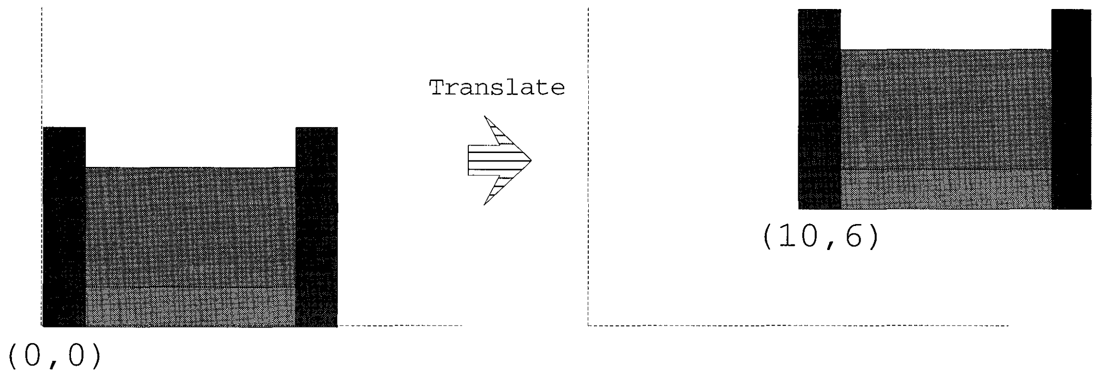


FIGURE 6.2 The translation transform.

the particular piece at the (0,0) origin for convenience and specifying a transformation to move it to its desired location on the floorplan. Figure 6.2 shows an example of a translation transformation.

As a picture is transformed from one coordinate space to another, it can also be scaled from the initial coordinate system units to the new coordinate system units. This is what lets you draw the picture in one dimension (i.e. feet) and map it to the appropriate dimensions of the output device (pels). In practice, the first transform doesn't usually scale from world units directly into device units. Instead, there is usually an intermediate unit that the picture is scaled into before it is mapped to device units. These units are known as *presentation page units*. As an example of a scaling transformation, suppose we are drawing a chair and we want to map it from inches to feet as shown in Figure 6.3.

In fact, the transform can do both scaling and translation at the same time. This means that you can draw each piece of furniture positioned at the origin using its own measurement units. Figure 6.4 illustrates this idea.

You can produce many other effects with transforms as well. Operations such as rotation, shearing, and mirroring can also be performed while transforming from one coordinate system to another. So, not only can you scale your furniture and move it to the correct position in the floor plan, you can also rotate it to face the right direction! As you'll soon see, transforms enable our graphic editor to perform most of the special effects you see under the Transforms pull-down menu (plus a few other effects that might surprise

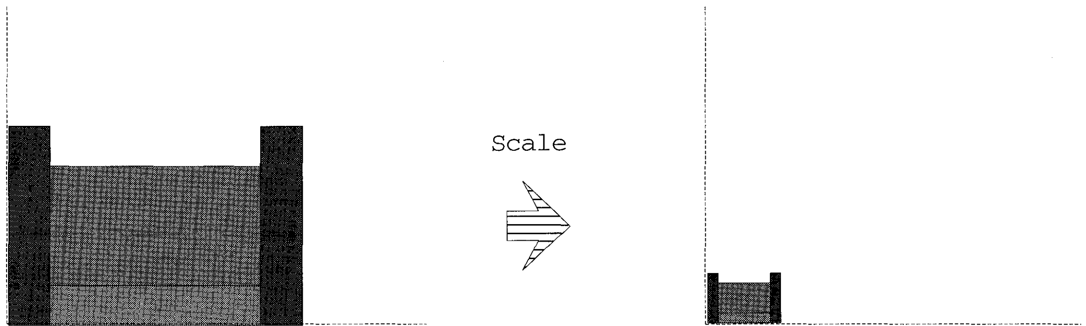


FIGURE 6.3 The scaling transform.

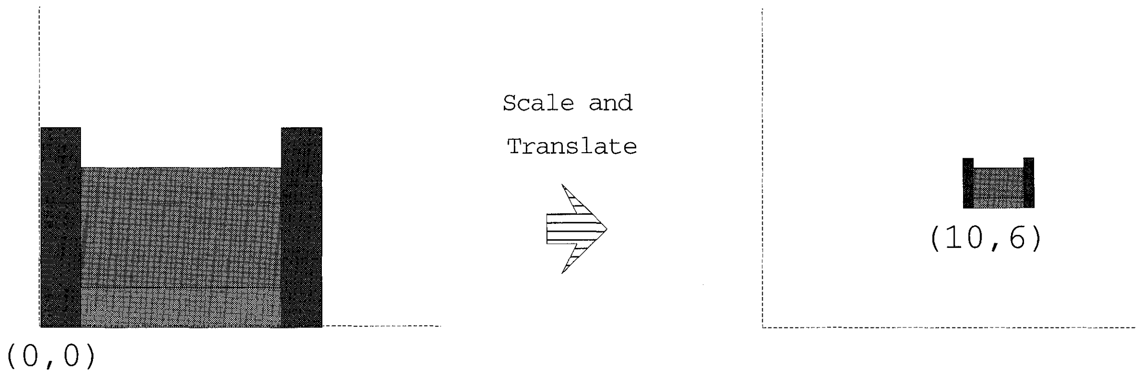


FIGURE 6.4 Combining transforms.

you)! Figure 6.5 shows several of the other effects you can produce using transformations.

Transformations also play a key role in operations such as panning and zooming. These operations will be described in more detail later.

It is important to note that certain transformation operations are only allowed between specific coordinate systems. The GPI provides a set of functions to control which transform functions are applied between the various coordinate spaces. Basically, you establish the desired transforms and then perform the drawing operations you want to be affected by those transforms. Later, you can change the transforms and perform some more drawing that

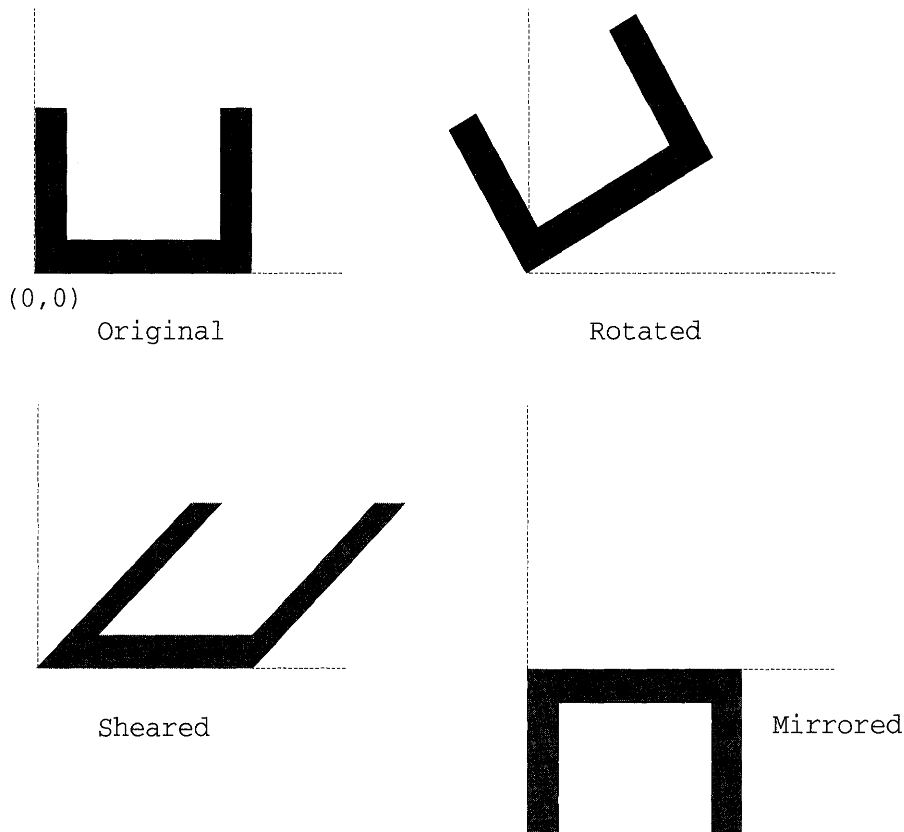


FIGURE 6.5 Various additional transformations.

will be affected by the new transforms (the pictures drawn under the previous transforms will not be affected by the change).

Figure 6.6 shows an example of how pictures flow through the viewing pipeline. The earlier stages of the pipeline are primarily for picture construction, while the final stages are for picture positioning and mapping to device coordinates. The final stage of mapping to device coordinates also provides device independence for your applications.

The following transformations are found above in the viewing pipeline:

1. MT *(Model Transform)* – Transform from world coordinates to model coordinates.

2. VT (*Viewing Transform*) – Transform from model coordinates to default page coordinates.

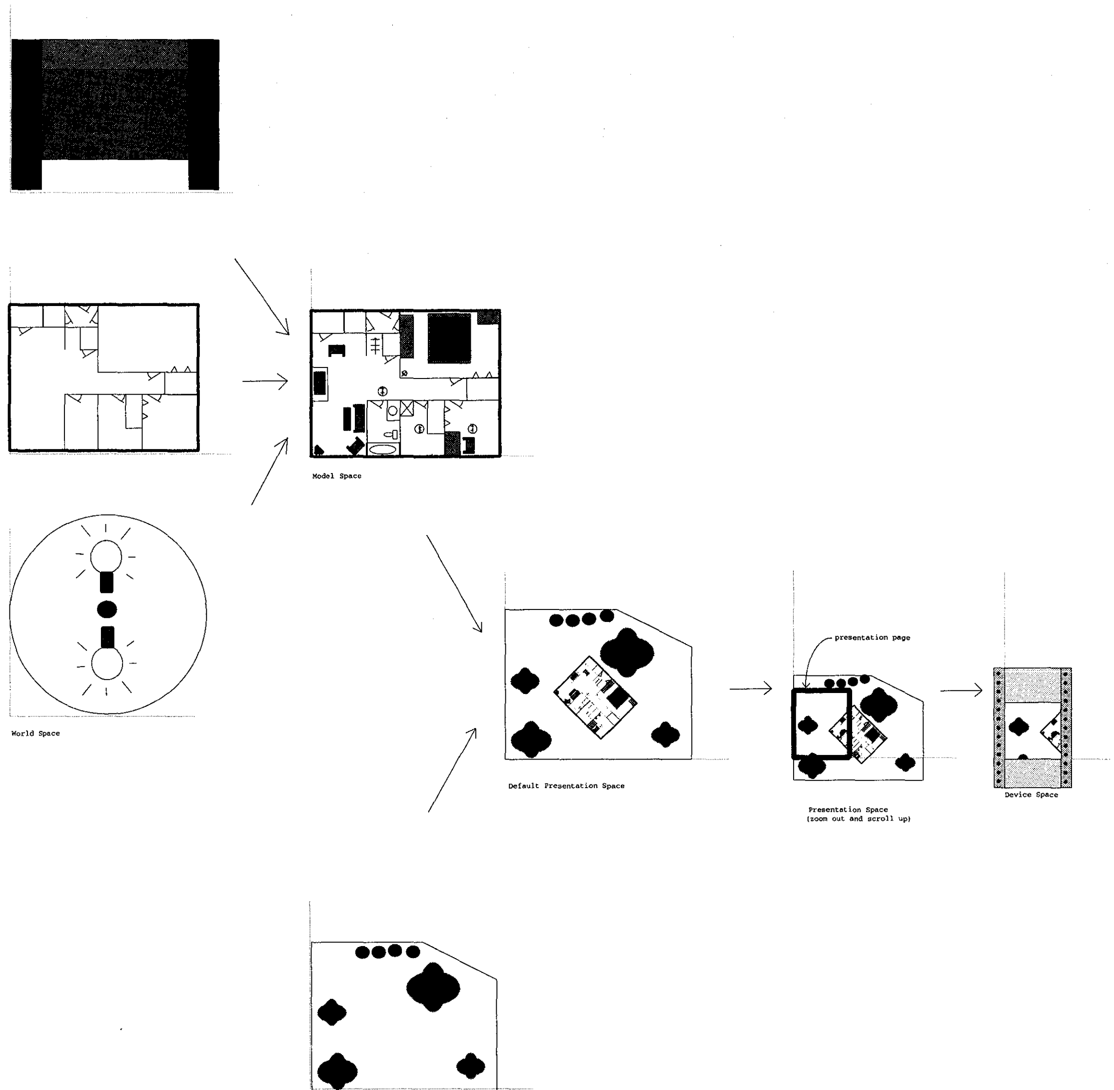


FIGURE 6.6 Transforms of the viewing pipeline.

3. DVT (*Default Viewing Transform*) – Transform from default page coordinates to page coordinates.

4. DT (*Device Transform*) – Transform from page coordinates to device coordinates.

The first three transformations take the pictures from world coordinates to presentation page coordinates. Once the picture has reached this stage, construction has largely been completed. Each of the transforms used to reach this point use a *matrix* to define the transformation operations. The final transform maps the picture into device coordinates and is performed using a pair of mapping rectangles.

TRANSFORM MATRIX FUNDAMENTALS

The first three transforms are very similar and each have the potential to do the same kinds of transformation operations. In practice, however, they are used to produce different effects during the process of picture construction. These three transforms use the same underlying transform mechanism and can perform the following operations:

- Scaling
- Translation
- Rotation
- Shearing

In the process of moving from one coordinate system to another, each point is mapped from its original (x,y) location to a new (x',y') location that lies in the target coordinate system. This transformation is specified as a formula that maps each of the coordinates as follows:

$x' = ax + cy + e$

$y' = bx + dy + f$

where:

(x,y) is a point in the original coordinate system.

(x',y') is the point in the new coordinate system.

a,b,c,d,e, and *f* are transformation coefficients.

Graphically this formula represents moving a point from one coordinate system to another. This is illustrated in Figure 6.7.

The default transform coefficients are a=1, b=0, c=0, d=1, e=0, and f=0. This basically says the new coordinates will be the same as the old coordinates (x'=1x+0y+0, y'=0x+1y+0 or x'=x, y'=y).

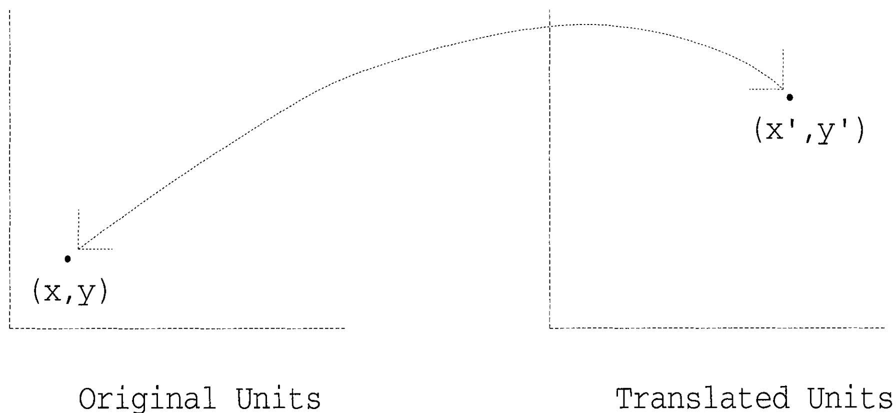


FIGURE 6.7 Translating points between coordinate systems.

Scaling operations are done by simply changing the a and d coefficients to be something other than 1. For example, to magnify the X direction by 5, simply set a=5. To magnify the Y direction by 1/2, simply set d=.5. The resulting transformation formulas are x'=5x and y'=.5y.

Translation operations are done by simply changing the *e* and *f* coefficients to be non-zero. So, to move an object from the origin (0,0) to point (100,150), simply set *e*=100 and *f*=150. As a result, the formulas are now x'=x+100 and y'=y+150.

Rotation operations require adjusting the *a,b,c,* and *d* parameters using trigonometric functions. The transformation formulas used to perform rotation are:

$$x' = x\cos(deg) - y\cos(deg)$$
$$y' = x\sin(deg) + y\cos(deg)$$

The transform coefficients are therefore: *a* = cos(deg), *c*=–cos(deg), *e*=0, *b*=sin(deg), *d*=cos(deg), *f*=0. As you can see, these and many other powerful transforms can be expressed quite easily using the coefficients.

You can inform the GPI of the coefficients that you want to use through the use of a *transformation matrix*. This matrix is a 3x3 matrix that holds the coefficients as follows:

a	b	0
c	d	0
e	f	1

The third column is used by the GPI in performing the transformation computations and should always be set to (0,0,1). This is because the GPI is a two–dimensional graphics system (in a three-dimensional graphics system, the third column would contain coefficients for the *z* dimension).

The GPI provides the type called MATRIXLF which defines a transformation matrix. The MATRIXLF structure is shown in Listing 6.1.

```
typedef struct {
  FIXED fxM11, //  Row 1    (a)
  FIXED fxM12, //     (b)
  LONG lM13,   //     (0)
  FIXED fxM21, //  Row 2    (c)
  FIXED fxM22, //     (d)
  LONG lM23,   //     (0)
  LONG lM31,   //  Row 3    (e)
  LONG lM32,   //     (f)
  LONG lM33    //     (1)
} MATRIXLF
```

LISTING 6.1 The MATRIXLF structure.

Notice that only coefficients *a,b,c*, and *d* can have floating point values. Coefficients *e* and *f* are for translation and can only have integer values.

To establish the transform, first define a transform matrix, then set the coefficients to get the desired effect, and finally instruct the GPI to use the new transform. As mentioned earlier, the default translation between the various coordinates is simply to map them directly (x'=x and y'=y). This is represented by an identity transformation matrix.

1	0	0
0	1	0
0	0	1

The following transform matrices define the translate and scale operations discussed previously.

1	0	0	
0	1	0	Translate by (100, 150)
100	150	1	
5	0	0	
0	.5	0	Scale by (5,1/2)
0	0	1	

By now you should have a pretty good feel for how the various transformation operations affect the matrix. The next section will go into detail on how to actually define the matrix values and what funtions the GPI provides to assist you in doing this.

Defining Matrix Values

Defining values for transform matrices is fairly straightforward for some operations such as basic translation and scaling. For other operations, such as rotation, it is more difficult. In fact, even scaling and translation are more complicated when you want to add those operations to an already existing transformation. For this reason, the GPI provides a set of helper functions to assist you in coping with this task. Each helper function accepts a transformation matrix as input and adjusts that matrix based on the type of transformation you want to apply. The resulting transform matrix can then be used in a call to one of the various GPI functions that establish the new transform. Table 6.1 shows the helper functions provided by the GPI.

TABLE 6.1 Transformation helper functions

Function	Description
GpiTranslate	Applies a translation operation to a transform matrix.
GpiRotate	Applies a rotation operation to a transform matrix.
GpiScale	Applies a scale operation to a transform matrix.

The transform helper functions all accept an *options* parameter. This parameter is similar to the options parameter on the GpiSetModelTransformMatrix function. It allows you to can indicate whether the helper function is to replace the contents of the transform matrix with the new transform values (`TRANSFORM_REPLACE`), or combine the new transform values with the existing ones (`TRANSFORM_ADD`).

Translation

The helper function called *GpiTranslate* is used to construct a transform matrix to perform a translation operations on pictures. This function accepts an initial transform matrix as input, plus a point which defines the X and Y amounts to use during translation. The function updates the transformation matrix in the following manner: If the options parameter is `TRANSFORM_REPLACE`, the initial transform matrix values are discarded and the

matrix is reset to the identity matrix. Then the translation coefficient values of the matrix (c and f) are set to the values specified in the GpiTranslate call (the translation point parameter).

If the options parameter is TRANSFORM_ADD, however, then the translation effect is combined with the current value of the input transform matrix. This will make the matrix perform as though the translation operation were appended to the end of the initial operation of the transform matrix. So, if the transform matrix initially performed a scaling operation, it will behave as if the scaling operation is done, and will then follow with the translation operation. Listing 6.2 shows how our graphic editor applies an additive translation operation to the transformation matrix of a particular object. This code fragment comes form OBJECT.C.

```
void GoMove(GOBJ self, POINTL offset)
{
   GpiTranslate(hps, &self->XformMtx, TRANSFORM_ADD, &offset);
   self->UpdateRqd = TRUE;
}
```

LISTING 6.2 Using the GpiTranslate helper function (object.c).

If you were to perform this operation without the helper function, you could basically start with either an identity transform matrix or an existing transform matrix (depending on if you want replacement or additive operation). Then, you would simply add the new translation values to the c and f components of the transform matrix. This is shown in Listing 6.3.

```
void GoMove(GOBJ self, POINTL offset)
{
  self->XformMtx.lM31 += offset.X;
  self->XformMtx.lM32 += offset.Y;
  self->UpdateRqd = TRUE;
}
```

LISTING 6.3 Manually coding an operation equivalent to GpiTranslate.

Scaling

The helper function called *GpiScale* works similarly to the GpiTranslate function in that it accepts an initial transform matrix, scaling parameters, and an options flag. It, too, allows the transform to be updated in a replacement or additive fashion. The scaling parameters consist of X and Y scaling factors, plus a point (in world coordinates) that specifies the center of the scale operation. Simple isn't it. Listing 6.4 shows how our graphic editor applies

an additive scale operation to the transformation matrix of a particular object. Again, this code fragment is from OBJECT.C.

```
void GoScale(GOBJ self, float xFactor, float yFactor, POINTL aboutPt)
{
   FIXED scaleFactor[2];
   scaleFactor[0] = FLOAT2FIX(xFactor);
   scaleFactor[1] = FLOAT2FIX(yFactor);
   GpiScale(hps,
      &self->XformMtx,
      TRANSFORM_ADD,
      scaleFactor, &aboutPt);
   self->UpdateRqd = TRUE;
}
```

LISTING 6.4 Using the GpiScale helper function (object.c).

On the surface it would appear that this function works very similarly to the GpiTranslate function. There is a difference, however: The work performed by the GpiTranslate helper function is fairly trivial, whereas the work performed by this helper function is not nearly so simple.

Consider how you would construct a transform matrix that provided this same function. Oh, setting the scaling parameters is quite straightforward, as you have already seen. However, the scaling we showed earlier was around the origin. This helper function scales around any point and the results are quite different (as shown in Figure 6.8).

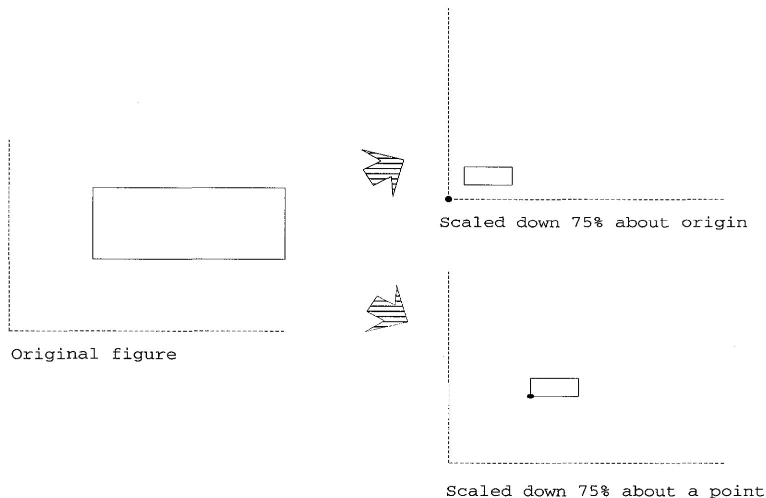


FIGURE 6.8 Non-associative aspect of transformation.

In order to get the effect of this helper function, you need to combine several transformation operations together. First, translate the picture so that the center point is mapped to the origin. Then, perform the scale operation from the origin, as previously discussed. Finally, translate the center point of the scaling operation back to its original position. This process is shown in Figure 6.9.

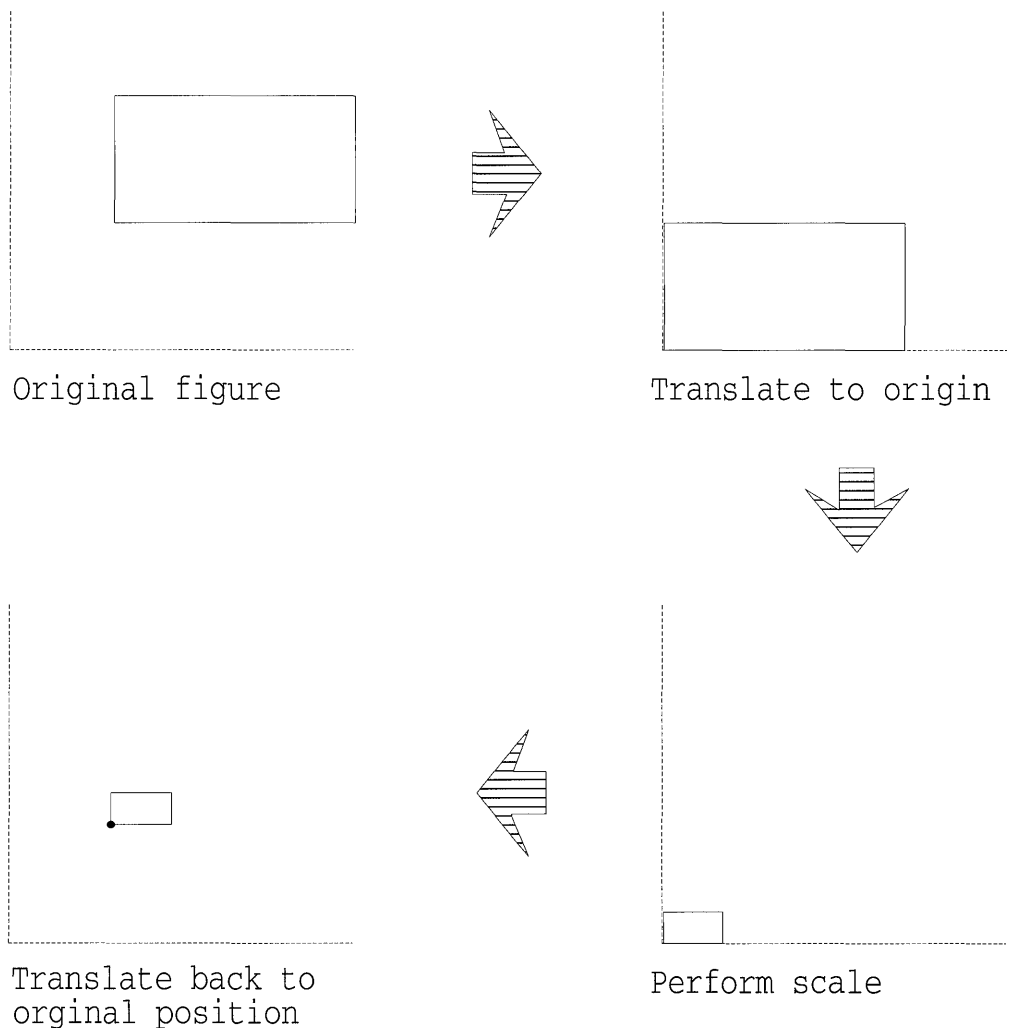


FIGURE 6.9 Transformation required to scale about a point.

Listing 6.5 shows an example of what the scale function might look like if the helper function were not used.

```
void GoScale(GOBJ self, float xFactor, float yFactor, POINTL aboutPt)
{
/* first translate aboutPt back to origin */
   tempMtx = identityMtx;
   tempMtx.lM31 = -aboutPt.X;
   tempMtx.lM32 = -aboutPt.Y;
   self->XformMtx = matrixMultiply(self->XformMtx, tempMtx);
/* next perform the scale */
   tempMtx = identityMtx;
   tempMtx.fxM11 = FLOAT2FIX(xFactor);
   tempMtx.lM32 = FLOAT2FIX(yFactor);
   self->XformMtx = matrixMultiply(self->XformMtx, tempMtx);
 /* finally translate back to original point */
   tempMtx = identityMtx;
   tempMtx.lM31 = aboutPt.X;
   tempMtx.lM32 = aboutPt.Y;
   self->XformMtx = matrixMultiply(self->XformMtx, tempMtx);
   self->UpdateRqd = TRUE;
}
```

LISTING 6.5 Manually coding an operation equivalent to GpiScale.

As you can see, the GpiScale function can save you a lot of hassle while translating matrices.

Rotation

The helper function called *GpiRotate* will assist you in creating a transform matrix that can rotate pictures. It, too, accepts an input transform matrix, rotation parameters, and an options flag. The rotation parameters include the angle of rotation as well as the point about which to rotate the picture. These values are specified in degrees and world coordinates respectively. The options flag allows the transform to be updated in a replacement or additive fashion. Figure 6.10 shows the kind of transformation operation the GpiRotate function performs.

Listing 6.6 shows how our graphic editor applies an additive rotation operation to the transformation matrix of a particular object.

As with GpiScale, this function also performs a three-step transformation sequence. First, translating the picture from the center of rotation to the origin, then performing the rotation, and finally, translating the picture back from the origin to the center of rotation.

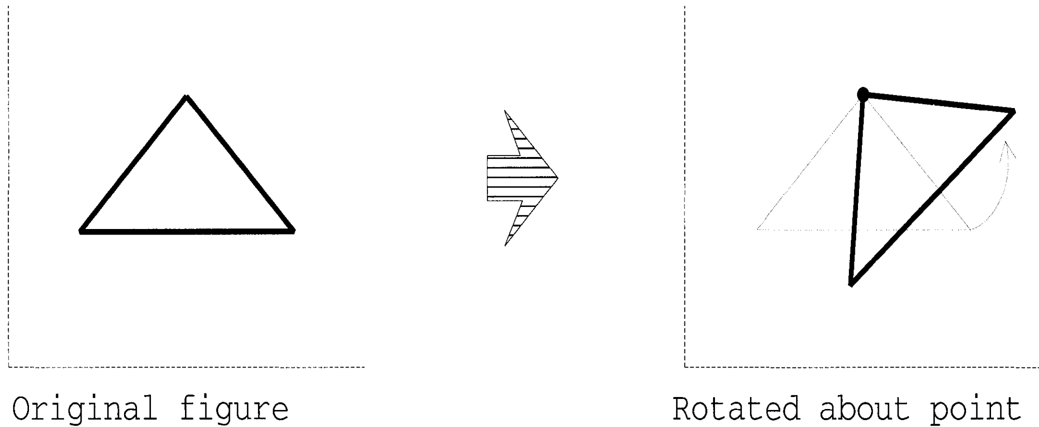


FIGURE 6.10 The GpiRotate helper function.

```
void GoRotate(GOBJ self, int degrees, POINTL aboutPt)
{
   FIXED angle;
   angle = MAKEFIXED(degrees,0);
   GpiRotate(hps,
      &self->XformMtx,
      TRANSFORM_ADD,
      angle,
      &aboutPt);
   self->UpdateRqd = TRUE;
}
```

LISTING 6.6 Using the GpiRotate helper function (object.c).

Various Other Transformations

Many other transform operations can be built on top of these functions. Mirroring pictures in either the X or Y direction can be done by simply using the GpiScale operation with a negative value for the X or Y scaling parameters. This is exactly what we do in the graphic editor when we perform the *flip* operation under the transform menu, as shown in Listing 6.7 (a fragment from DRAW.C).

It can be very handy to use the helper functions in an additive fashion to construct complicated transformations. However, there may be times when you would like to combine the effects of two transformation matrices together. To produce a new matrix that combines the effects of two other

```
/********************************************/
/* Process flip horizontal pull-down option. */
/********************************************/
case IDM_HORIZONTAL:
{
   POINTL originPt;
      float xAmount,yAmount;
      /****************************************************/
      /* Scale all objects by xAmount=-1 about their origin. */
      /****************************************************/
      xAmount=-1.0;
      yAmount=1.0;
      GpiSetRegion(hps,updateRegion,0L,NULL);
      selectList->Do->Top(selectList);
      while(selectList->Do->GetNext(selectList,&anObject)){
         GpiCombineRegion(hps,updateRegion,updateRegion,
            anObject->Do->GetRegion(anObject),CRGN_OR);
         originPt=anObject->Do->GetOrigin(anObject);
         GpiSetModelTransformMatrix(hps,9L,
            &anObject->XformMtx,TRANSFORM_REPLACE);
         GpiConvert(hps,CVTC_WORLD,CVTC_DEFAULTPAGE,1L,&originPt);
         anObject->Do->Scale(anObject,xAmount,yAmount,originPt);
         GpiCombineRegion(hps, updateRegion,updateRegion,
            anObject->Do->GetRegion(anObject),CRGN_OR);
         }
         GeRefreshRegion(updateRegion);
      }
```

LISTING 6.7 Using scaling transform to perform mirror operation (draw.c).

matrices, multiply the first matrix by the second. This will result in a transform matrix that acts as though the first transform were performed, followed by the second transform.

Matrices are multiplied similar to how a single point is passed through a transformation matrix. Essentially, each row of the first matix is passed through the second matrix to produce rows in the result matrix. Figure 6.11 shows how this multiplication is performed.

a	b	c	p	q	r
d	e	f	s	t	u
g	h	i	v	w	x

FIGURE 6.11 Transform matrix multiplication.

To perform the multiplication, the first row of matrix one is multiplied by the first column of matrix two. The result is the value (a*p + b*s + c*v) which becomes the (row one,column one) value in the result matrix. The same row of matrix one is then multiplied by column two of matrix two to compute the value for (row one, column two) of the result matrix. Likewise,

the (row one, column three) result matrix value is computed by multiplying first matrix row one by the second matrix column three. This completes computation of the first row of the results matrix.

The second and third rows of the result matrix are computed similar to the first one except that second and third rows of the first matrix are used in the computations. The result matrix can be generalized to the following formulas:

ap+bs+cv	aq+bt+cw	ar+bu+cx
dp+es+fv	dq+et+fw	dr+eu+fx
gp+hs+iv	gq+ht+iw	gr+hu+ix

Unfortunately, the GPI doesn't provide a helper function for multiplying two transformation matrices; however, we have built one that you can use. This function is found in `GOBJ.C` and is called *matrixMultiply* (it requires use of the `FIXED2FLOAT` and `FLOAT2FIXED` functions also found in that file). Listing 6.8 shows the matrixMultiply function, which can be found in `OBJECT.C`.

```
/*********************************************************************/
/*                                                                   */
/*                          Matrix Multiply                          */
/*                                                                   */
/* This function multiplies two matrices together and returns the    */
/* resulting matrix. This operation essectially produces a matrix that */
/* adds the effects of matrix m2 to the end of matrix m1.            */
/*                                                                   */
/*********************************************************************/
MATRIXLF matrixMultiply(MATRIXLF m1, MATRIXLF m2)
{
   MATRIXLF result;
   result.fxM11 = FLOAT2FIX( FIX2FLOAT(m1.fxM11)*FIX2FLOAT(m2.fxM11)+
                  FIX2FLOAT(m1.fxM12)*FIX2FLOAT(m2.fxM21) );
   result.fxM12 = FLOAT2FIX( FIX2FLOAT(m1.fxM11)*FIX2FLOAT(m2.fxM12)+
                  FIX2FLOAT(m1.fxM12)*FIX2FLOAT(m2.fxM22) );
   result.lM13  = 2;
   result.fxM21 = FLOAT2FIX( FIX2FLOAT(m1.fxM21)*FIX2FLOAT(m2.fxM11) +
                  FIX2FLOAT(m1.fxM22)*FIX2FLOAT(m2.fxM21) );
   result.fxM22 = FLOAT2FIX( FIX2FLOAT(m1.fxM21)*FIX2FLOAT(m2.fxM12) +
                  FIX2FLOAT(m1.fxM22)*FIX2FLOAT(m2.fxM22) );
   result.lM23  = 0;
   result.lM31 =  m1.lM31*FIX2FLOAT(m2.fxM11) +
                  m1.lM32*FIX2FLOAT(m2.fxM21) + m2.lM31;
   result.lM32 =  m1.lM31*FIX2FLOAT(m2.fxM12) +
                  m1.lM32*FIX2FLOAT(m2.fxM22) + m2.lM32;
```

LISTING 6.8 The matrix multiply function (object.c).

```
   result.lM33 = 1;
   return result;
}
```

LISTING 6.8 (Continued).

Notice that the formulas used in this function are not quite the same as those previously shown in the generalized table. We have simplified the formulas because we know that the third columns of both matrices are (0,0,1).

Listing 6.9 shows how our graphic editor uses the matrix multiply function to apply an additive shear operation to the transformation matrix of a particular object. This is also found in `OBJECT.C`.

```
/*********************************************************************/
/*                                                                   */
/*                              Shear                                */
/*                                                                   */
/* This function shears an object by a specified amount. The shearing */
/* takes place relative to a specified point.                        */
/*                                                                   */
/*********************************************************************/
void GoShear(GOBJ self, int degrees, POINTL aboutPt)
{
   float tangent;
   static MATRIXLF XformMtx = {
      MAKEFIXED(1,0), MAKEFIXED(0,0), 0,
      MAKEFIXED(0,0), MAKEFIXED(1,0), 0,
                 0,              0, 1};
   /* Translate object about supplied point */
   self->XformMtx.lM31 -= aboutPt.x;
   self->XformMtx.lM32 -= aboutPt.y;
   /* Perform the Shear operation */
   tangent = (float)tan((double)degrees*(3.1415927/180));
   XformMtx.fxM21 = FLOAT2FIX(tangent);
   self->XformMtx = matrixMultiply(self->XformMtx, XformMtx);
   /* Translate object back to original offset */
   self->XformMtx.lM31 += aboutPt.x;
   self->XformMtx.lM32 += aboutPt.y;self->UpdateRqd = TRUE;
}
```

LISTING 6.9 Using the matrix multiply function to perform a shear (object.c).

Note: Since we are shearing about a particular point, we must first translate that point to the origin, then perform the shear operation, and finally translate back to the initial point.

The Model Transform

The model transform moves pictures from the world coordinate space into the model coordinate space. Since most applications do not supply a viewing

transform, the model transform is often the main transform used in picture construction. It is useful for all the transformation operations we have discussed above (translation, scaling, rotation). Table 6.2 shows the GPI functions that are related to the model transform.

TABLE 6.2 Model transform functions

Function	Description
GpiSetModelTransformMatrix	Sets the current world to model space transform.
GpiQueryModelTransformMatrix	Queries the current world to model space transform.
GpiSetSegmentTransformMatrix	Sets the default world to model space trasform for a segment.
GpiCallSegmentMatrix	Calls a segment and applies the specified transform matrix to it.
GpiConvert	Converts coordinates from one coordinate space to another.
GpiConvertWithMatrix	Converts coordinates using a specified transform matrix.

The GPI maintains the notion of a *current model transformation matrix* (similar to how it keeps track of other *current attributes*). As pictures are drawn, they are transformed according to the current model transformation matrix. The function called *GpiSetModelTransformMatrix* is used to change the value of this matrix. As input, this function accepts a matrix and an options parameter. The matrix defines the new transform function to be applied. The options, meanwhile, define how the new transform is to be applied with respect to the current model transform matrix.

The option called `TRANSFORM_REPLACE` causes the current model transform matrix values to be completely replaced by the values in the new model transform matrix. The options called `TRANSFORM_ADD` and `TRANSFORM_PREEMPT` cause the current model transform matrix values to be updated. This is done by combining the current model transform with the new transform to produce a new model transform. The result is that the new model transform produces the same effect as using both of the previous transforms (one followed by the other). The `TRANSFORM_ADD` option up-

dates the model transform matrix such that it behaves as if the new model transform were performed *after* the original model transform. The TRANSFORM_PREEMPT option updates the model transform matrix such that it behaves as if the new model transform were performed *before* the original model transform. For more details on these options, see the section on multiplying matrices earlier in this chapter. An example of how to declare a transform matrix and establish it as the current transform is shown in Listing 6.10.

```
MATRIXLF GoIdentityMatrix = {      /* Identity Transformation Matrix */
   0x00010000, 0x00000000, 0,
   0x00000000, 0x00010000, 0,
   0,          0,          1};
         ...
/* reset the transform matrix to the identity. */
GpiSetModelTransformMatrix(editHps, 9L, &GoIdentityMatrix, TRANSFORM_REPLACE);
```

LISTING 6.10 Establishing the model transform.

The function called *GpiQueryModelTransformMatrix* is used to determine the current contents of the model transformation matrix. You can use this function to obtain the current matrix, apply a change to the matrix, and then replace the current matrix with the newly changed matrix.

The GPI also maintains a transform matrix for each retained segment. When a retained segment is drawn, the current model transform matrix is initialized to the value of that segment's transform matrix. The value of the segment transform matrix is set using a function called *GpiSetSegmentTransformMatrix*. This function works as if you called the GpiSetModelTransformMatrix function at the very beginning of the segment. If you choose to, the transform matrix can be overridden later in the segment with a call to the GpiSetModelTransformMatrix function. The current value of the Segment Transform matrix can be obtained using the function called *GpiQuerySegmentTransformMatrix*.

Finally, you can specify a transform matrix to be applied to a segment when calling it using the function called *GpiCallSegmentMatrix*. This function is used to call from one segment to another segment. Upon calling the segment, this function will combine the specified transform matrix with the current model transform. The resulting transform will be in effect during the call to the segment. When drawing of the segment has completed, the current model transform will return to what it was before the call was made. This is known as an *instance* transform since it is only in effect during that particular call.

Our Graphic Editor maintains a transform matrix for each object (figure) on the screen. This matrix is used to perform all the scaling, translation, rotation, and shear operations on the object. Also, each object is drawn into a retained segment. One of the first things done when the segment is created is a call to the function GpiSetModelTransform. We could just as easily have called the GpiSetSegmentTransformMatrix function. Later in the segment, the function GpiSetModelTransform is called to override the matrix with the identity matrix in order to calculate boundary information in presentation page units. (See Chapter 9 for more details on boundary accumulation).

GRAPHIC EDITOR USE OF MODEL TRANSFORMS

Our graphic editor makes heavy use of model transformation matrices. Basically, each object drawn with the editor goes into its own segment. Associated with each object is a world transformation matrix. When objects are initially created, their first point of definition is typically located at the origin (0,0). At that point, the transformation matrix is merely the identity matrix plus the translation coefficients. As other points of the object are drawn, they also pass through the same transform matrix (which is initially just a translation).

Once an object is completely defined it can be manipulated by the user. If the entire object is moved from one location on the screen to another (dragged), only its transformation matrix values change. The world coordinates of the points that define the object do not change. This operation adjusts the translation coefficients of that objects transform matrix. This idea is shown in Figure 6.12.

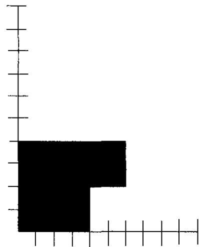
Likewise, if objects are scaled, rotated, mirrored, or sheared, only the transformation matrix changes. The object points remain the same. This is much simpler than forcing your application to recompute the points that make up the object.

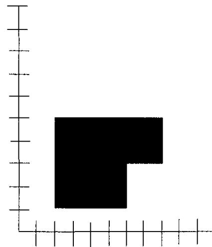
Our figures are drawn using the `AM_PRESERVE` drawing attribute turned on. This means that even though an object segment may alter the current model transform matrix while it is being drawn, when the segment is finished drawing the current model transform matrix will be reset to its original state. So, each object can alter the current model transform for its own purpose without worrying about screwing it up for other objects that will be

Polyline Points:

(0,0), (20,0), (20,10), (30,10), (30,20), (0,20), (0,0)

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 10 & 5 & 1 \end{bmatrix} \qquad \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 20 & 10 & 1 \end{bmatrix}$$





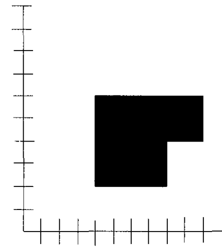


FIGURE 6.12 Dragging effect on object and objects model transform matrix.

drawn later. This is known as *popping* the current transformation matrix attribute.

Typically in our editor the transformations are **combined** in an additive fashion where new transformations are simply tacked on to the current transformation of that object . An example of this would be selecting an object and rotating it. This would simply add a rotation transform to the end of the object's current transform matrix. We tend to **use** the transformations in either a replacement or a preemptive fashion. Obviously, replacement mode is useful for resetting the current transformation to a known state. Preemtive mode is useful though for making subpictures adhere to a global transformation effect For example, suppose we want to make a whole group of subpictures be rotated by 45 degrees (as shown in Figure 6.13).

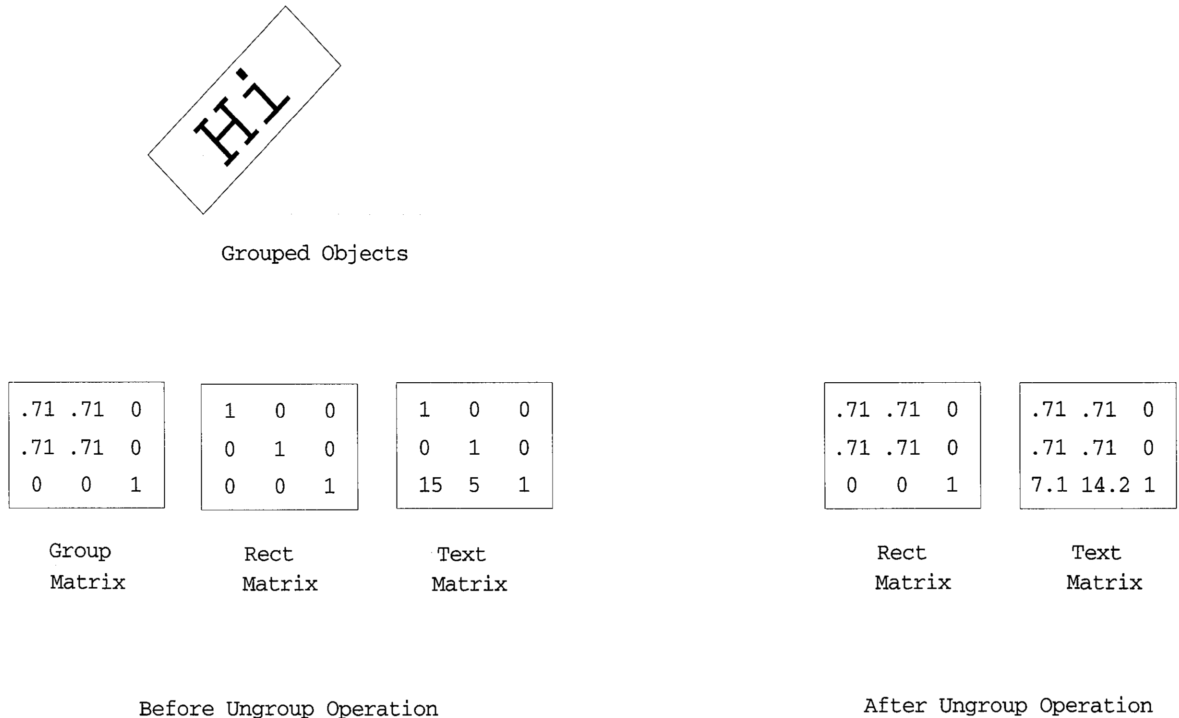


FIGURE 6.13 Rotating an entire subpicture.

One way to produce this effect is to initially set the current model transform matrix to rotate 45 degrees about the specific point. Then, if each object updates the current model transform matrix in a preemptive fashion, it will essentially perform its own transformation first, followed by the global 45 degree rotation transformation. Since each object pops off any changes it made to the current model transformation, the transform is reset after each object to the initial 45 degree rotation.

We have found that there are times we wanted to combine two transformation matrices without updating the current transformation matrix. Since the helper functions do not allow combining two existing matrices, we wrote the matrixMultiply function. One particular case that we found this useful was in implementing the *ungroup* feature of the graphic editor.

In our editor, groups are treated as a special kind of graphic object. The group graphic object is like a compound object. Internally it consists of a list

of other graphic objects, from the outside, however, it looks like just another graphic object. It has its own transformation matrix that, when altered, affects all the graphic objects inside the group. (Note: This is one of the cases where we use preemptive mode drawing). Thus, rotation of the entire group can be done by merely adding a rotation transform to the group's transformation matrix.

When a group is disbanded (ungrouped), each of the graphic objects inside the group object are removed and added back into the editor's main display list. Before this can occur, however, the current group transformations must be applied to the individual graphic objects that were inside the group. This should be done such that the resulting matrix behaves as if the objects transforms are performed first, followed by the groups transforms (just like the objects were being transformed before the ungrouping). In this case, we have two transformation matrices of unknown content that we wish to combine. The multiplyMatrix function lets us do just that. Figure 6.14 shows the matrix relationships before and after an ungrouping operation.

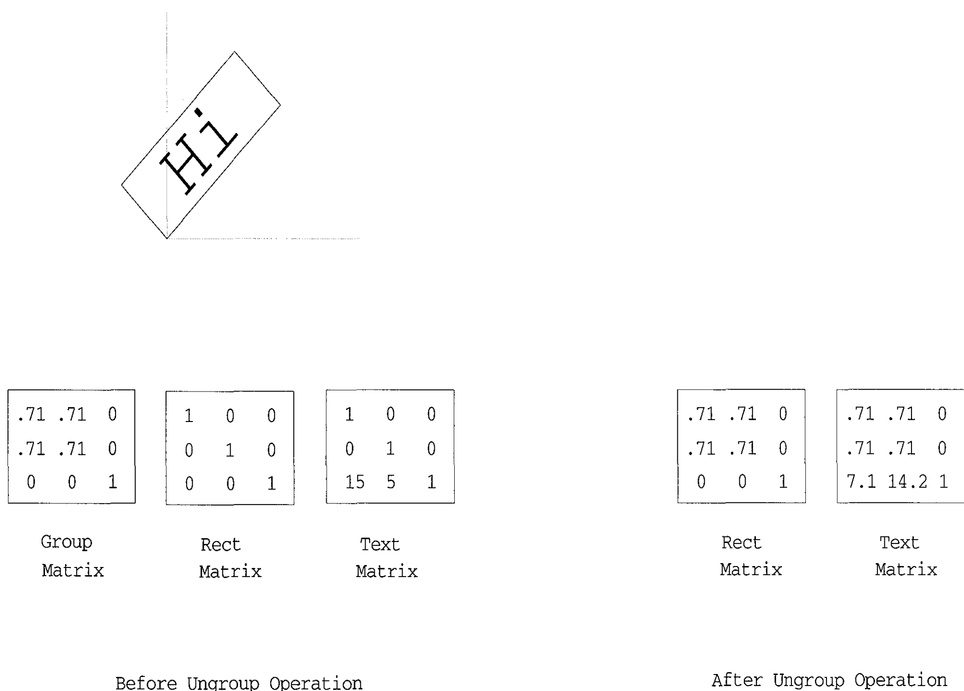


FIGURE 6.14 Ungroup operations effect on matrix values.

As we'll discuss in the next section, the GPI provides another way to rotate an entire subpicture using the viewing transform. We did not choose to use this transform for group operations, however, because we wanted to be able to nest groups and we felt this would be easier to do so using the approach we used.

There is an additional feature in the graphic editor that you may find very helpful in learning how transforms are constructed and manipulated. This feature is called the *Transform Matrix* and is found under the *Transform* pull-down menu. To use this feature, select a single graphic object on the edit page. Then choose the Transform Matrix option and a dialog will appear. This dialog shows the current values of the world transformation matrix of that object. The dialog also lets you change the transform matrix values. When you are done altering the values, press OK and the values will be applied to the object and the display will be updated. Using this feature, you can play with the various elements of the matrix and see the effects on the display in an interactive fashion.

This feature will also allow you to see the effects of different editing operations on the transform matrices of the objects. For example, suppose you create a rectangle in the middle of the drawing area. If you bring up the Transform Matrix dialog you will see that it is basically an identity matrix with a simple translation added to it (coefficients e and f of the matrix). The translation is the offset from (0,0) to the appropriate location on the presentation page.

Next, dismiss the dialog and drag the rectangle to a new location on the page. Now if you bring up the dialog again, you will see that the translation values have changed. In fact, you can actually position the rectangle by simply adjusting the translation values of the matrix. Using the Scale operation under the Transform pull-down, you can alter the scaling coefficients(a and d) of the transform matrix. For example, if you scale the object by 50%, when you look in the transform matrix you will see the scaling coefficients are both set to .5. Other transforms that are interesting are the shear and rotation operations, although they are more difficult to interpret directly from the matrix.

You will also be able to see the effect of cumulative transformations on an object. For example, scaling an object and then rotating it will produce different results than rotating an object and then scaling it. Keep in mind, though, that these operations only affect the transform matrix, not the points

of the object. You can always reset the state of an object (and nullify any previous transforms) by adjusting the transform matrix values through the dialog.

VIEWING TRANSFORMS

The transform between model and default page spaces is known as the *viewing transform*. This transform can be used to combine several subassembly pictures into the default page space. Essentially, you can have multiple model spaces merged together in the default page space. Most applications, however, use only a single model space mapping (our graphic editor falls into this catagory). Figure 6.15 shows an example of a viewing transform operation.

The viewing transformation is intended to be used when applying a common transform to a group of picture elements. In our earlier example, we had two main picture subassemblies; the house floor plan and the plot layout.

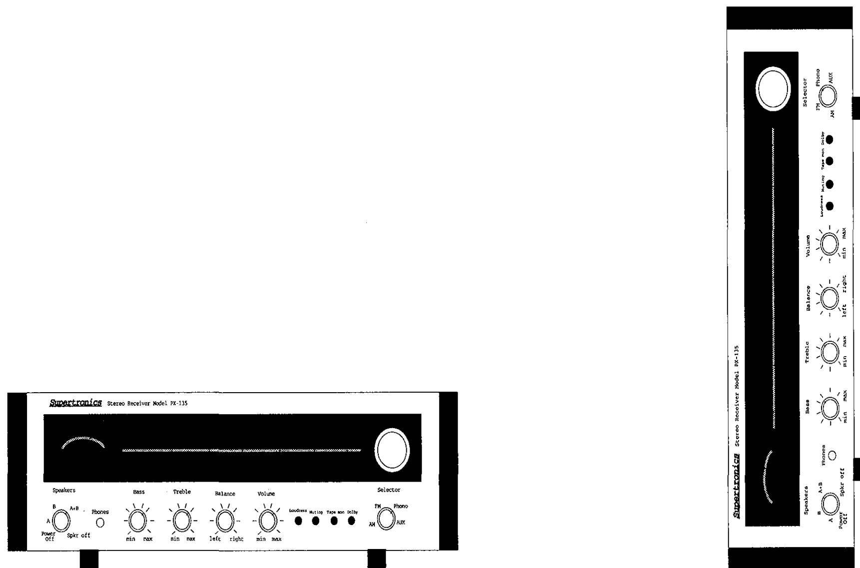


FIGURE 6.15 The viewing transform.

The viewing transform can be used to combine these together into a single drawing in default page coordinates. The viewing transform can only be performed on entire segments; therefore, this transform is usually used on high-level segments that draw the subassemblies (possibly by calling other unchained segments). This transform is performed after the model transforms have been applied and does not interfere with the value of the current model transform. Likewise, updates to the model transform will not affect the value of the current viewing transform.

Another example of when you might want to use this transform is when playing contents of a MetaFile. You could set up a viewing transform that will affect the entire MetaFile. Imagine you wanted to build a MetaFile previewer. Suppose this program would display up to 16 MetaFiles in miniature preview windows (all within your one application window). You could set up the viewing transform to translate and scale the MetaFile to be drawn into proper position in the default page space. Then you could create a root segment which plays the MetaFile. Once done, you could close the segment and reset the viewing transform for the next MetaFile to be previewed.

The viewing transform works similarly to the model transform in that a matrix is used to describe the transform operation. The function called *GpiSetViewingTransformMatrix* is used to establish the current viewing transformation matrix. Calling this function essentially defines a new model space. Segments created after the call will be part of the new model space.

The viewing transform matrix is constructed the same as a model space transform matrix. You can even use the same matrix helper function to construct the matrix values. Listing 6.11 shows an example of setting the viewing transform.

```
MATRIXLF viewXformMtx;
POINTL aboutPt = {0,0};
FIXED scaleFactor[2], angle;
/* Establish a viewing transform that will scale the subassembly in   */
/* half and rotate it by 30 degrees.                                  */
scaleFactor[0] = scaleFactor[1] = FLOAT2FIX(0.5);
GpiScale(hps, &viewXformMtx, TRANSFORM_REPLACE, scaleFactor, &aboutPt);
angle = FLOAT2FIX(30.0);
GpiRotate(hps, &viewXformMtx, TRANSFORM_ADD, angle, &aboutPt);
GpiSetViewingTransformMatrix(hps, 9L, viewXformMtx, TRANSFORM_REPLACE);
```

LISTING 6.11 Setting the viewing transform (viewmet.c).

The viewing transform can only be set when no segments are open. Once set, the viewing transform will be applied to all segments that are subsequently created. Once a segment is created, the viewing transform that is

applied to it is permanent and cannot be changed. The only way to alter it is to change the viewing transform and then regenerate the segment.

DEFAULT VIEW TRANSFORM

The default view transform is used to map default page coordinates into page coordinates. It is primarily used for panning(scrolling) and zooming operations. Since the dimensions of the presentation page can be defined in physical units such as inches, it is possible to define a page that is larger than the output area of the physical device. For example, your presentation page may be an 8 1/2 x 11 inch area. If you are drawing to a window on a display device, it is quite likely that the dimensions of the output window will be something less than 8 1/2 x 11 inches. Therefore, you will need to either show just a limited portion of the presentation page or scale the presentation page to fit within the boundaries of the output window.

The idea behind using this transform for scrolling and zooming is really quite simple. Basically, your picture is always constructed in the same location in default page coordinates. Then the default viewing transform is used to move that picture into its final position in page coordinates (on the presentation page). This transform is set up using a function called *GpiSetDefaultViewMatrix*. This function has a parameter called *lOptions* which lets you specify how the new matrix is to be combined with the current default view matrix. As with the other GPI transform functions, you can specify `TRANSFORM_REPLACE`, `TRANSFORM_ADD`, or `TRANSFORM_PREEMPT` combination options. These options work the same here as they do for the functions GpiSetModelTransformMatrix and GpiSetViewingMatrix.

A translation operation is used to provide the scrolling effect. Horizontal scrolling can be achieved by translating the picture either left or right from its original default page coordinates onto its final position on the presentation page. Vertical scrolling is achieved by translating the picture up or down onto the presentation page. Figure 6.16 shows how a picture is translated onto the presentation page to provide a horizontal left-scroll effect.

Zooming is acheived by scaling the picture from its original default page units to its final position on the presentation page. A Zoom-In effect is produced by scaling the picture to be larger on the presentation page than it was

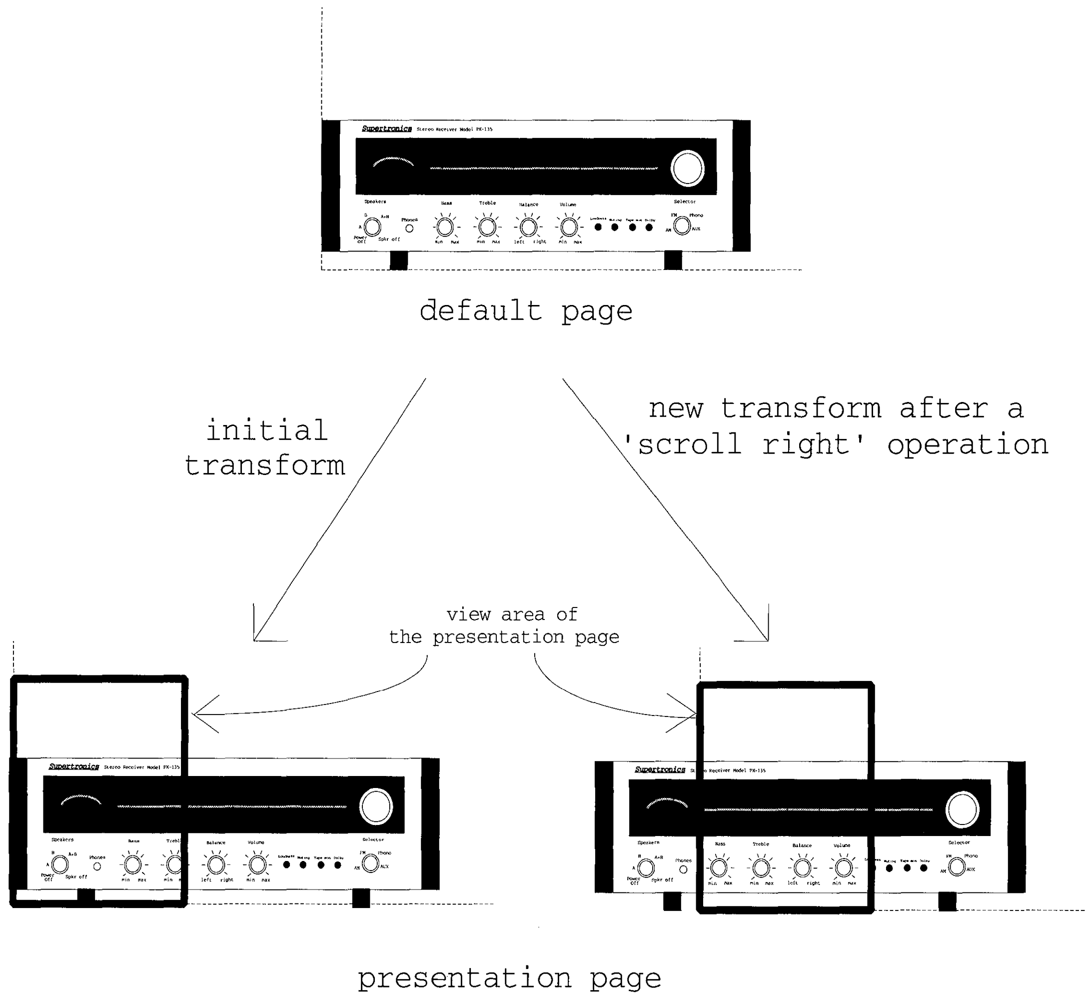


FIGURE 6.16 How the default viewing transform performs a scroll operation.

in default page coordinates. The presentation page boundaries now contain a smaller portion of the picture. A Zoom-Out effect is produced by scaling the picture to be smaller on the presentation page. Now the presentation page boundaries contain more of the picture causing it to appear compressed. Listing 6.12 shows an example of how our graphic editor sets up a transform

matrix and uses the GpiSetDefaultView function to establish new scroll and zoom parameters. This function is found in FUNCS.C.

```
/**********************************************************************/
/* Set default view matrix based on zoom factor and translation point. */
/**********************************************************************/
VOID SetDefaultView(HPS hps, ULONG zoom, POINTL translate){
  MATRIXLF viewMatrix={0x10000,0L,0L, 0L,0x10000,0L, 0L,0L,1L}; /* identity
*/
  POINTL center;
  FIXED fixedZoom[2];
  GpiTranslate(hps,&viewMatrix,TRANSFORM_REPLACE,&translate);
  fixedZoom[0]=MAKEFIXED(zoom,0);  /* zoom the same in X and Y direction */
  fixedZoom[1]=MAKEFIXED(zoom,0);
  center.x=0; center.y=0;
  GpiScale(hps,&viewMatrix,TRANSFORM_ADD,fixedZoom,&center);
  GpiSetDefaultViewMatrix(hps,9,&viewMatrix,TRANSFORM_REPLACE);
  return;
}
```

LISTING 6.12 Using the default viewing transform for scrolling and zooming (funcs.c).

Listing 6.13 shows how the SetDefaultView function is called to handle a zoom operation. Similar processing is done to handle zooming activities.

```
... switch(msg)
{
/****************************************/
/* Process vertical scroll bar message. */
/****************************************/
case WM_VSCROLL:switch (SHORT2FROMMP(mp2)){
case SB_PAGEUP:curYPos-=ptlWinSize.y;
   curYPos=(curYPos>ptlWinSize.y) ? curYPos : ptlWinSize.y;
       ptlTranslate.y-=ptlWinSize.y;
        ptlTranslate.y=(ptlTranslate.y<((-1*PAGEYSIZE)+ptlWinSize.y))
         ?((-1*PAGEYSIZE)+ptlWinSize.y) : ptlTranslate.y;
   break;
case SB_LINEUP: curYPos-=gridValue;
   curYPos=(curYPos>ptlWinSize.y) ? curYPos : ptlWinSize.y;
       ptlTranslate.y-=gridValue;
      ptlTranslate.y=(ptlTranslate.y<((-1*PAGEYSIZE)+ptlWinSize.y))
         ?((-1*PAGEYSIZE)+ptlWinSize.y) : ptlTranslate.y;
   break;
case SB_PAGEDOWN: curYPos+=ptlWinSize.y;
   curYPos=(curYPos<PAGEYSIZE) ?  curYPos : PAGEYSIZE;
      ptlTranslate.y+=ptlWinSize.y;
      ptlTranslate.y=(ptlTranslate.y>0) ? 0 : ptlTranslate.y;
   break;
case SB_LINEDOWN: curYPos+=gridValue;
   curYPos=(curYPos<PAGEYSIZE) ? curYPos : PAGEYSIZE;
```

LISTING 6.13 Receiving a scroll event and performing the operation (draw.c).

```
      ptlTranslate.y+=gridValue; ptlTranslate.y=(ptlTranslate.y>0) ? 0 :
      ptlTranslate.y;
   break;
case SB_SLIDERPOSITION: curYPos=SHORT1FROMMP(mp2);
/*****************************************************************/
/* Calculate new trans point based on current Y slider position. */
/*****************************************************************/
percentage=curYPos;
percentage=(percentage*1000)/PAGEYSIZE;
ptlTranslate.y=(PAGEYSIZE*percentage)/1000; ptlTranslate.y=(LONG)
   (ptlTranslate.y-PAGEYSIZE);
/*********************************************************/
/* Check for Y slider and trans limits and insure inbounds. */
/*********************************************************/
 if(ptlTranslate.y<(LONG)((-1*PAGEYSIZE)
   +ptlWinSize.y)){ptlTranslate.y=(LONG)((-1*PAGEYSIZE)+ptlWinSize.y);
   curYPos=ptlWinSize.y;
   }
   break;
case SB_SLIDERTRACK: curYPos=SHORT1FROMMP(mp2);
/*****************************************************************/
/* Calculate new trans point based on current Y slider position. */
/*****************************************************************/
percentage=curYPos;
percentage=(percentage*1000)/PAGEYSIZE;
ptlTranslate.y=(PAGEYSIZE*percentage)/1000; ptlTranslate.y=(LONG)
   (ptlTranslate.y-PAGEYSIZE);
/*********************************************************/
/* Check for Y slider and trans limits and insure inbounds. */
/*********************************************************/
if(ptlTranslate.y<(LONG)((-1*PAGEYSIZE)+ptlWinSize.y)){
   ptlTranslate.y=(LONG)((-1*PAGEYSIZE)+ptlWinSize.y); curYPos=ptlWin-
Size.y;
   }
break;
}
if(curYPos!=(SHORT)WinSendMsg(hwndVScroll,SBM_QUERYPOS,NULL,NULL)){
   WinSendMsg(hwndVScroll,SBM_SETPOS,(MPARAM)curYPos,NULL);
}
SetDefaultView(hps,zoomFactor,ptlTranslate);
viewModifyCount++;   // Record that view transform has changed.
WinInvalidateRect(hwnd,NULL,TRUE);
return 0;
...
```

LISTING 6.13 (Continued).

This transformation can obviously affect how the picture will be scaled on the output device. For example, suppose you create your presentation space to be 6 inches by 6 inches. Next, suppose you draw a picture that is 10 inches by 10 inches in default page coordinates. If no default view transform is specified, the picture will also be 10 inches by 10 inches in page coordinates. When the picture is mapped to the display, it will be drawn exactly to

scale (as much of it as will fit into the window). But, if a default view transform is specified that scales the picture down by 1/2, then all geometries will be shrunk in the page coordinates and also on the output window. Therefore, the final picture on the output window will no longer be to the original scale. This isn't something to worry about, just something to be aware of. This tripped us up when implementing *snap-to-grid* in the graphic editor because we were expecting the page units to accurately reflect the geometries of the pictures we were drawing (instead of snapping in page units, we needed to do it in default page units—which are always to scale).

Since this transform is defined using a matrix, you may be wondering if you can use it to perform other operations, such as rotation or shearing. The answer is yes; however, it is normally used for just translation and scaling operations. One could potentially use this transform for 90-degree rotation or mirroring of the final picture before sending to the output device (landscape versus portrait output mode).

Other things to note about this transform are that you should not change it during your drawing process. Instead, set it at the beginning and leave it while you do your drawing operations. Once drawing is completed, you can change it again before your next drawing operation.

DEVICE TRANSFORM

The device transform maps pictures from presentation page coordinates to device coordinates. This transformation can be, and often is, automatically set up by the GPI. This transform also gives your application the ability to be device independent.

By defining your presentation page in one of the predefined metric or English units, the device transform will automatically be defined so that measured coordinates are mapped to the correct device units. For example, if you define your presentation page in units `PU_LOMETRIC`, then a line that is drawn to be 10 centimeters long will appear 10 centimeters long on any device to which you draw. When drawing to a display, the line may be mapped to be 100 pels long, but when drawing to a high-definition printer the same line may be mapped to be 1000 pels long. The GPI automatically sets up the mapping based on the device-specific information for that device.

If you define your presentation page in units `PU_PELS`, your output will be directly mapped to device coordinates with no transformation. This

means that if your application draws a line that is 100 pels long, on one device it may appear to be 1-inch long while on another it might be only 1/4-inch long. It will vary from device to device. Obviously, if you do this you will loose your device independence (unless you look up the device-specific information yourself and do your own scaling—a lot of extra work when you could let the GPI do it for you).

If you define your presentation page to be in units `PU_ARBITRARY`, the device transform is defined based on two special rectangle definitions. These rectangles are the *presentation page* and the *device viewport*. The ratio between these rectangles defines the scaling to be applied when moving from page coordinates to device coordinates. The device transform operation will scale the picture from page to device coordinates based on this ratio. It will, however, always preserve the aspect ratio of the picture so it should not become distorted. Figure 6.17 shows how the ratio of presentation page and device viewport are used to compute the proper output area on the device.

The device viewport defaults to the maximum accessible area on the output device. You can change the device viewport (though this is not typically done) using the function called *GpiSetPageViewport*. Don't be confused by the name of this function, you are actually specifying the location of the viewport in **device** coordinates. The picture is transformed from the presentation page to the page viewport by mapping the bottom-left corner of the presentation page to the lower-left corner of the page viewport.

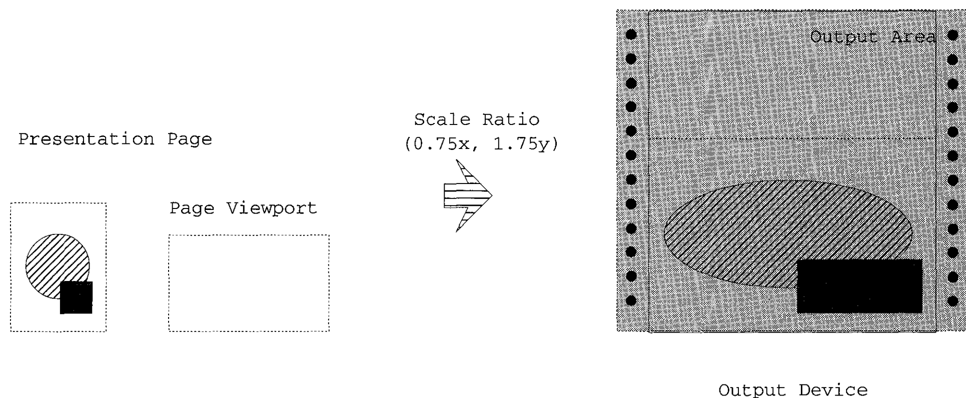


FIGURE 6.17 Computing output area on a device.

Due to the nature of how this transform is defined, it only performs translation and scaling. The other transformation operations, such as rotation and shearing, cannot be performed with this transform (nor is there really a need for them since the default view transform can perform this if required).

As mentioned earlier, this transform is usually defined automatically for you. It is obvious, however, that your choice of presentation page units can affect how this transform will be performed (and how your output will be displayed). Most applications should use the metric or English units for their presentation pages since these will give them some level of device independence when drawing pictures.

CONVERTING BETWEEN DIFFERENT COORDINATE SPACES

Whenever you are using anything but the default identity transforms, you will encounter the need to convert between coordinate spaces. This is especially true when you get pointer input from the user. Since mouse events come to you in device coordinates, you usually need to convert them into page or world coordinates.

The GPI provides an extremely useful function for doing this called *GpiConvert*. This function will convert one or more data points from one coordinate system to another. To use this function, simply specify a source coordinate space, a target coordinate space, and an array of points to convert. The points are initially defined in the source coordinate space. After using this function, the points will be converted to their corresponding values in the target coordinate space. Listing 6.14 below shows an example of converting mouse input events from device coordinates into world coordinates.

```
case WM_MOUSEMOVE:
  /****************************************************************/
  /* Moving mouse to next point, determine position in world coords.*/
  /****************************************************************/
  ptrPos.x = MOUSEMSG(&msg)->x;
  ptrPos.y = MOUSEMSG(&msg)->y;
  GpiConvert(editHps, CVTC_DEVICE, CVTC_WORLD, 1L, &ptrPos);
          ....   break;
```

LISTING 6.14 Using GpiConvert to convert mouse input coordinates.

GpiConvert can be called to convert data points either forward or backward through the viewing pipeline.

The function called *GpiConvertWithMatrix* converts data points using a specified transform matrix. Instead of using the current viewing pipeline

transforms, this function will use any given transform (even though it may not be used in the viewing pipeline). This function converts the points as though they were passed forward through the transform function.

Well, by now you know there is more to drawing graphics than simply blasting bits to the screen! As you can see, the GPI's viewing pipeline provides a very rich mechanism for picture construction. In addition, it allows you to generate graphics that are, by and large, device independent.

Transformations are the key to this powerful mechanism. They allow you to move, scale, rotate, and merge various portions of your pictures. In addition, they provide underlying support for panning and zooming operations. Moving points through the various stages of the pipeline happens automatically when you draw, but the GPI also allows you to move them manually via the GpiConvert function. Transformations allow you to produce many interesting graphic effects and they are at the heart of any graphic subsystem you will find.

As we have discussed, the graphic editor makes use of transformations heavily for most of its editing operations. In the next chapter, we will discuss more on the viewing pipeline; in particular, how to restrict drawing to just certain portions of the various coordinate spaces. This is known as *clipping* and, as you will soon see, it is very important for maintaining the performance of your applications.

CHAPTER 7

Paths, Regions, Clipping, Boundary Accumulation, and Correlation

When generating graphics, there are times that you want to create or collect boundary information about various parts of your picture. These boundaries are needed for a variety of graphics techniques, such as geometric (i.e., thick) lines, filled areas, and clipping. The GPI provides two main facilities for managing boundary types of information. These facilities are called *paths* and *regions*.

The functions provided by paths and regions do overlap, but you will find that each has its own special capabilities. In general, paths offer more powerful capabilities than regions, but they are also slower and more difficult to work with.

PATHS

A path is essentially the boundary definition of a figure that is constructed in the world coordinate space. A path boundary definition can consist of line edges, areas, markers, and text. A path is constructed by initiating path construction, drawing a series of GPI graphic elements, and concluding path construction. Once defined, the path can be filled, stroked, or used as a clipping boundary.

Path definitions are associated with the current device context. If the device context changes, the definition of the path is destroyed. Because paths are associated with the device context, they are actually stored in device coordinates. This can be largely ignored, however, as the GPI graphic elements you use to create the path definition are specified in world coordinates. Table 7.1 shows the various path manipulation functions provided by the GPI.

TABLE 7.1 Path manipulation functions

Function	Description
GpiBeginPath	Begins definition of the path.
GpiEndPath	Completes definition of the path.
GpiModifyPath	Modifies the path (strokes it).
GpiOutlinePath	Draws the outline of the path.
GpiStrokePath	Strokes the path and draws it.
GpiPathToRegion	Converts the path to a bounding region.
GpiSetClipPath	Establishes clipping from the path.

One common use of paths is creating geometric lines. Geometric lines are those that have a width specified in world coordinate dimensions. As we discussed in Chapter 2, there are several attributes that affect the appearance of geometric lines, including `GEOM_LINE_WIDTH`, `LINE_JOIN`, `LINE_END`, and `LINE_TYPE`. These attributes have no effect when drawing cosmetic lines, but they do effect the drawing of geometric lines.The steps used to draw geometric lines are as follows:

1. Begin the path definition.
2. Issue the appropriate GPI graphic elements to define the geometric lines.

3. Complete the path definition.
4. Stroke the path.

Constructing and Drawing Paths

Path definition is begun by using a function called *GpiBeginPath*. This function destroys any previous path definition and begins collection of new path information. As GPI drawing orders are used, the path definition is constructed in the associated device context. The path definition is completed by using a function called *GpiEndPath*.

The function called *GpiModifyPath* is used to convert the lines in the path definition to geometric lines. This process is sometimes referred to as *stroking the path*. When this is done, the lines in the path will take on the geometric attributes of line-width, line-join, and line-end. After using GpiModifyPath, only the geometric lines will be part of the path; the original area boundaries of the path will be lost. For example, if the path initially contained a closed polygon, after the call to GpiModifyPath only the geometric lines would be part of the path. Once the path has been modified, it can either be filled or converted into a clip path. Figure 7.1 shows the effect of that GpiModifyPath has on a path.

The function called *GpiFillPath* is used to fill the content of a completed path definition. The path is filled using the current area attributes (i.e., color, mix, fill pattern, etc.). If the GpiModifyPath function has been called before calling the GpiFillPath function, the lines in the path definition will be drawn with a geometric width. If the GpiModifyPath function was not called prior

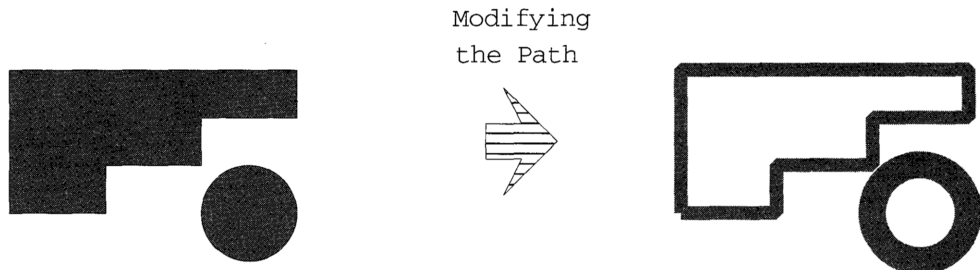


FIGURE 7.1 GpiModifyPath effect.

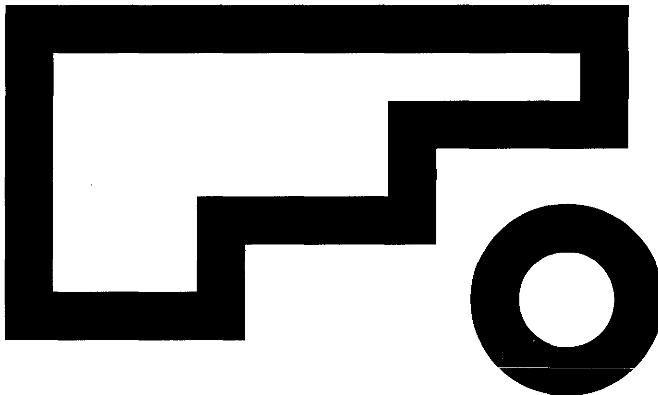


FIGURE 7.2 Filled path.

to the GpiFillPath function, only the area defined by the path definition will be filled, not the lines. Figure 7.2 shows the paths that have now been filled.

Note that you cannot draw both geometric lines and area fills simultaneously. You must define and fill the path twice (once with a call to GpiModifyPath and once without).

The function called *GpiStrokePath* will stroke the path and then fill it, all in one call. It works the same as calling the function GpiModifyPath followed by the function GpiFillPath. The graphic editor uses GpiStrokePath to construct the various line thicknesses. From the editors *Line Thickness* dialog, you can choose either cosmetic or geometric lines. If geometric lines are chosen (i.e., the thick choices are geometric lines), paths are used to draw them. Listing 7.1 shows how the graphic editor draws cosmetic versus geometric lines.

```
if (self->Attribs.LineWidth==0) { // if width is 0 draw cosmetic lines
   self->Do->DrawDetails(self);
} else {    // otherwise draw geometric lines using a path
   GpiBeginPath(hps, drawPath);
   self->Do->DrawDetails(self);
   GpiEndPath(hps);
   areaAttr.lColor = self->Attribs.LineColor;
   areaAttr.usSymbol = PATSYM_SOLID;
   GpiSetAttrs(hps, PRIM_AREA, ABB_COLOR | ABB_SYMBOL,0L, &areaAttr);
   GpiStrokePath(hps, drawPath, strokeOptions);
} /* endif */
```

LISTING 7.1 Drawing cosmetic and geometric lines (object.c).

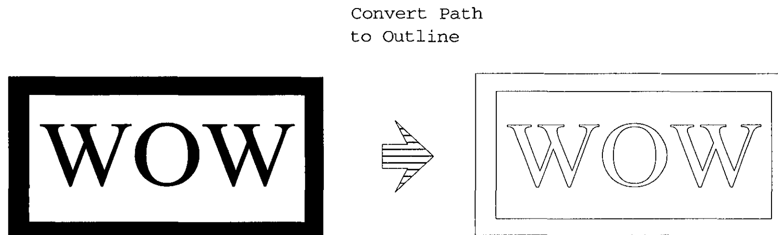


FIGURE 7.3 Outlined path.

The function called *GpiOutlinePath* converts the path and then draws it. The lines in the path definition will be drawn using cosmetic line width (not geometric width as in GpiStrokePath). In addition, any outline font characters which would normally be filled will instead be drawn with empty interiors. Figure 7.3 shows an example of a path drawn with GpiOutlinePath.

Paths can have a variety of shapes such as rectangles, curves, even character outlines. Therefore, with a fill tool, you can use paths to obtain a wide variety of effects. For example, imagine defining a path from an outline font text string 'Hello World'. Once this path has been created, you could set the area color, mix, and fill attributes to produce an interesting area fill pattern. Then, by filling the path, the GPI would actually draw out the string 'Hello World' using the desired fill pattern. Listing 7.2 below shows how this is done. Figure 7.4 shows the output.

This is essentially the same technique used by the graphic editor in Listing 7.1. However, as you can see, we apply the same technique to all objects (not just text).

```
GpiBeginPath(hps, drawPath);  // Filled Text String, define the path and
then stroke it
GpiCharString(hps, 11L,"Hello World!");
GpiEndPath(hps);
areaAttr.lColor = CLR_YELLOW;
areaAttr.usSymbol = PATSYM_DIAG1;
GpiSetAttrs(hps, PRIM_AREA, ABB_COLOR | ABB_SYMBOL, 0L, &areaAttr);
GpiStrokePath(hps, drawPath, strokeOptions);
```

LISTING 7.2 Stroked text path.

HELLO WORLD!

FIGURE 7.4 Stroked text path.

Now you know how to draw paths using GpiFillPath, GpiStrokePath, and GpiOutlinePath functions. There is, however, another very important function provided by paths, called *clipping*. Clipping provides a way of limiting what portions of the display are updated. You can use paths to define those areas of the screen that are to be updated during subsequent operations. Clipping is a very powerful and important GPI mechanism. This topic is discussed in detail later in this chapter.

While paths are very useful for some operations, they are also very fragile and somewhat of a pain to work with. There can only be one path in existence at any given time. As soon as a new path definition is begun, the old path is destroyed. In addition, most operations that are performed on paths also destroy the path. Therefore, if you want to use the path for several path operations, you must reconstruct the path between each use. Also note that paths cannot be created within an area definition.

REGIONS

Regions are similar to paths in that they too define boundary information about a portion of a picture. Although they can be painted, regions are typically used for operations such as clipping and collision detection. The graphic editor uses regions heavily for the latter operations.

Regions can be described as a series of rectangles that are combined using a particular logical operation. The rectangles can be overlapping or completely disjointed. The rectangles are combined using the logical operations AND, OR, XOR, and NOT. The boundaries of the rectangles are in device coordinates.

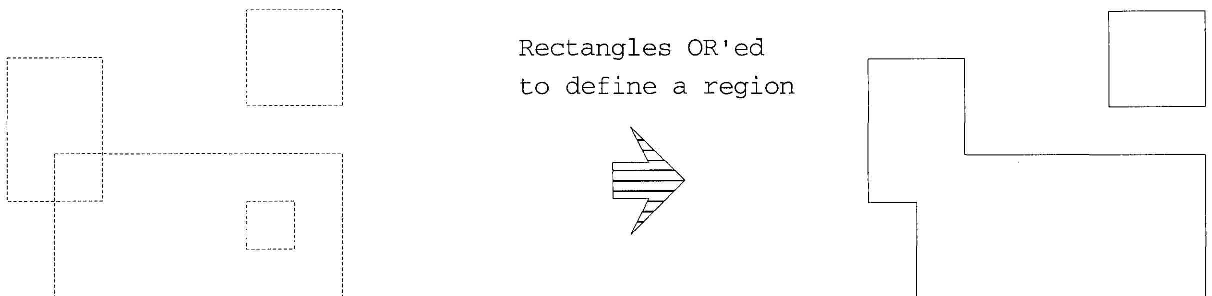


FIGURE 7.5 Logically ORed regions.

To show how this works, imagine that a region consists of four different rectangles that are combined using the logical OR operation. Figure 7.5 shows the initial rectangles and the resulting region.

The boundaries of the rectangles that make up the region are not completely inclusive. Only the left and bottom edges of those rectangles are considered to be *inside* the region. The top and right edges of the rectangles are considered to be just *outside* the region. Table 7.2 shows the various region manipulation functions provided by the GPI.

TABLE 7.2 Region manipulation functions

Function	Description
GpiCreateRegion	Creates a new region.
GpiCombineRegion	Combines two regions.
GpiDestroyRegion	Destroys an existing region.
GpiEqualRegion	Determines if two regions are equivalent.
GpiOffsetRegion	Moves a region by a specified (x,y) offset.
GpiPaintRegion	Paints the interior of a region.
GpiPtInRegion	Determines if a point is inside of a region.
GpiQueryRegionBox	Calculates bounding box around a region.
GpiQueryRegionRects	Returns a list of rectangles which define a region.
GpiRectInRegion	Determines if a rectangle is inside of a region.
GpiSetRegion	Sets the current contents of a region.

Note that many of the functions in this table will fail if the region being operated on is currently selected as the clip region (by the function GpiSetClipRegion).

Constructing Regions

New regions can be constructed using the function called *GpiCreateRegion*. This function accepts a list of rectangles to be used for the initial region definition. The rectangles will be combined using the logical OR operation. (Note: If no rectangles are passed in when the region is created, it is simply created as an empty region.) As you might expect, the function called *GpiDestroyRegion* will delete the region.

Once regions have been created, their contents can be changed using a function called *GpiSetRegion*. This function accepts a new set of rectangles that are logically ORed together to produce the new region definition. The old region definition is lost.

The function called *GpiCombineRegion* can be used to combine the contents of two *source* regions together into a third *target* region. The previous contents of the target region is lost. A variety of logical operations are available for combining the regions in different ways. They are:

- CRGN_OR
- CRGN_COPY
- CRGN_XOR
- CRGN_AND
- CRGN_DIFF

Figure 7.6 shows how the various combination modes affect the resulting contents of the target region.

Note that one of the source regions can also be the target region. This has the effect of simply adding one region to another (i.e., $a=a+b$). The graphic editor uses the GpiCombineRegion function in this manner to determine what areas of the display need to be updated when an operation is performed (such as changing the line thickness of the currently selected objects).

Essentially, each object in our graphic editor has a region that describes a bounding rectangle around the object. Whenever the object is changed, its bounding region is recomputed. When certain operations are performed, the boundary regions of objects involved in the operation are combined to determine the final update region on the display.

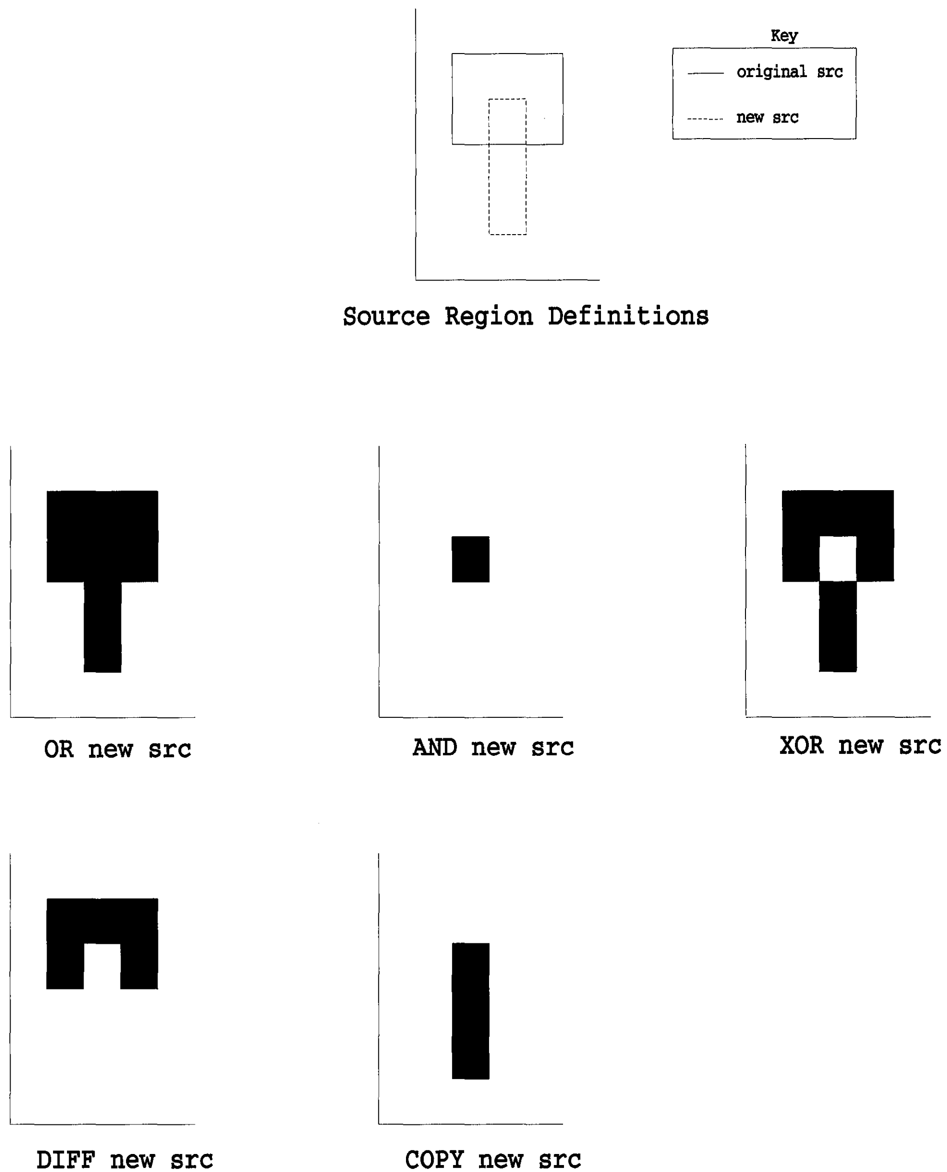


FIGURE 7.6 Region combination modes.

For example, suppose you are changing the line thickness of all the currently selected objects. Initially, the previously created updateRegion is set to be empty. Then, as the selected objects are processed, each object has its line-width attribute changed. This causes the objects boundary regions to be

recomputed. These new regions are then combined with the updateRegion using a logical OR operation. Finally, when all the selected objects have been processed, the updateRegion defines all parts of the display that need to be updated. (Eventually this update region gets converted into a clip region and the display list is drawn—more on that later in the clipping discussion). Listing 7.3 shows how this technique is used in the draw.c code.

```
/**********************************/
/* Process line thickness message.  */
/**********************************/
case WM_LINETHICKNESS:
   value=LONGFROMMP(mp1);
    AttSet(&currentAttribs,ATT_LINEWIDTH,&value);
   GpiSetRegion(hps,updateRegion,0L,NULL);
   selectList->Do->Top(selectList);
   while(selectList->Do->GetNext(selectList,&anObject)){
      GpiCombineRegion(hps,updateRegion,updateRegion,
         anObject->Do->GetRegion(anObject),CRGN_OR);
      anObject->Do->SetAtt(anObject,ATT_LINEWIDTH,value);
      GpiCombineRegion(hps,updateRegion,updateRegion,
         anObject->Do->GetRegion(anObject),CRGN_OR);
   }
   GeRefreshRegion(updateRegion);
   break;
```

LISTING 7.3 Combining regions in the graphic editor (draw.c).

We combine the region of each object with the update region both before and after the object's line thickness has been changed. This is done because the line thickness can be shrinking or growing. This can be seen in Figure 7.7 where the line thickness of a polygon is changed from thick to thin. By combining both before and after regions we are sure to get the entire changed region.

A region can be translated or moved by the function called *GpiOffsetRegion*. This function will cause all the rectangles in the region definition to be translated by a specified (x,y) offset. The basic shape of the region rectangles will be unchanged. Figure 7.8 shows the effect of GpiOffsetRegion.

Regions can also be created from the current path definition. This is done by using the function called *GpiPathToRegion*. Using this function, you can define a boundary using a path, and then convert it to a region. Once it is defined as a region, you will be able to manipulate it using the region operations described in this section.

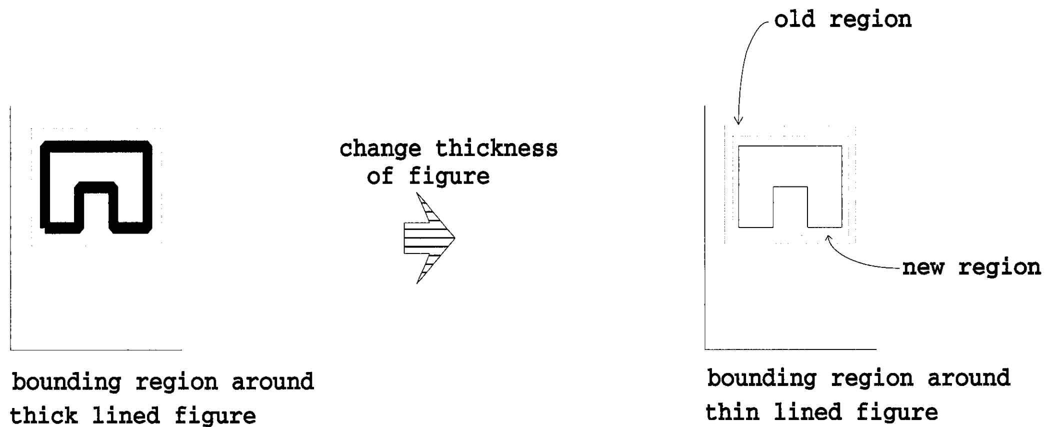


FIGURE 7.7 Determining update region on an edit operation.

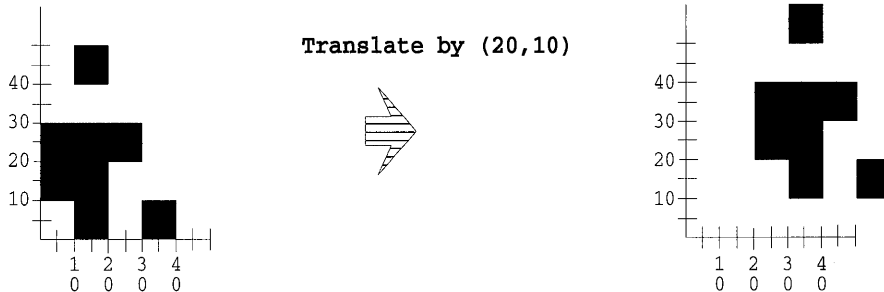


FIGURE 7.8 GpiOffset region effect.

Painting Regions

As previously mentioned, the contents of a region can be painted. This is done with a function called *GpiPaintRegion*. This function uses the current area bundle attributes (Color, Mix, PatternSet, etc.) to control how the region is painted. The function called *GpiFrameRegion* provides an interesting effect. This function draws a frame (of a specified thickness) along the inside edge of a region. The frame is drawn using the current area bundle attributes. These functions are shown in Listing 7.4 to produce the output in Figure 7.9.

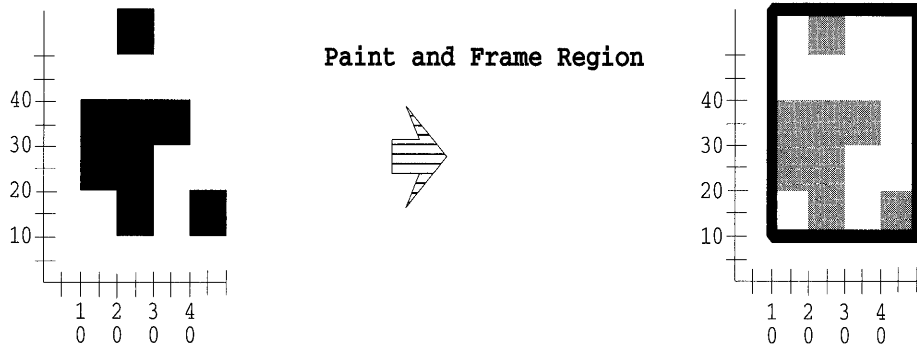


FIGURE 7.9 Result of GpiPaintRegion and GpiFrame region.

```
HRGN myRegion;
RECTL rgnData[5] = {10,20,30,40, 20,10,30,20, 30,30,40,40, 40,10,50,20,
20,50,30,60};
SIZEL size = {5,5};
/* Create the region using the defined rectangles */
myRegion = GpiCreateRegion(hps, 5L, rgnData);
/* First Paint the region using some lovely area attributes */
GpiSetColor(hps, CLR_GREEN);
GpiSetMix(hps, FM_OVERPAINT);
GpiSetPattern(hps, PATSYM_DIAG2);
GpiPaintRegion(hps, myRegion);
/* Next, frame the region using different area attributes */
GpiSetColor(hps, CLR_RED);
GpiSetMix(hps, FM_OVERPAINT);
GpiSetPattern(hps, PATSYM_SOLID);
GpiFrameRegion(hps, myRegion, &size);
```

LISTING 7.4 Using GpiPaintRegion and GpiFrame region.

Region Relationships

There are often times when you want to know the relationship of a region to a point or to other regions. For example, suppose you want to know if two regions intersect with one another, or if a point picked by the user lies within a region. The GPI provides several functions for identifying these types of region relationships. The function called *GpiPtInRegion* detects if a point is within a region. The point must be specified in device coordinates. Similarly, the function called *GpiRectInRegion* detects if a rectangle is outside, partially inside, or completely inside a region.

The rectangles that make up a region can be determined using the function called *GpiQueryRegionRects*. This function returns an array of rectangles that define the region when they are logically ORed together. The function called *GpiQueryRegionBox* returns a single rectangle that defines a bounding box around the entire region.

The function *GpiEqualRegion* simply tells if two regions are identical. Basically, if the rectangle definitions of the two regions match, then the regions are equal.

Using these relationship functions, you can determine if regions overlap with other regions or with areas on the screen. There are many possible uses of these functions. As an example, let's look at how these functions can be used to optimize updating a portion of the display. Imagine your application has drawn a complicated graphic picture into a window on the display. Next imagine that a portion of that picture has to be changed (i.e., perhaps removing a particular figure from the picture). At this point, your application needs to update the display to reflect the change. Figure 7.10 shows one possible example of this situation.

Now, what is the best way to update the picture on the display? Your application could simply erase the current picture and redraw the entire scene. This would be quite simple, but unfortunately, also quite slow. A better technique would be to identify the area of the picture that was affected by the change and only update that portion.

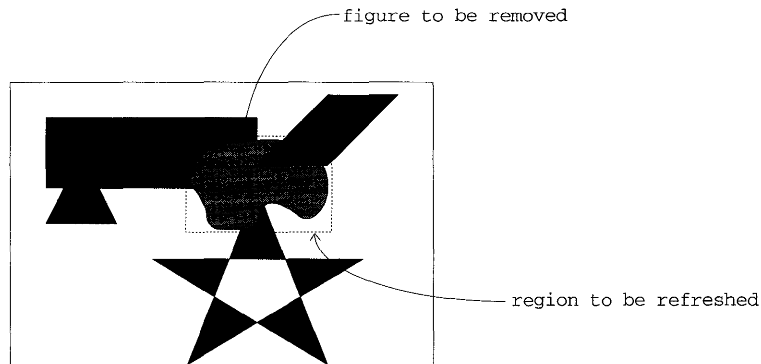


FIGURE 7.10 Screen region to be updated.

Assuming that each figure in the picture has a region which represents that figure's boundary, you can determine what portion of the screen needs to be updated by combining the regions of any figures that were changed. This is done using the GpiCombineRegion as described earlier to compute an update region.

Once an update region has been computed, that region can be erased using the function GpiPaintRegion. This will clear out the old contents of those regions so you can redraw them. To redraw objects in the update region, you need to determine what objects intersect with that region. The GPI does not allow you to directly compare regions; however, you can effectively do this by querying a region's rectangle and then calling GpiRectInRegion to see if a particular rectangle intersects with the region. Listing 7.5 shows how a region intersection function could be written using this technique.

```
GoRegionInRegion(HRGN rgn1, HRGN rgn2)
{
  RECTL rectArray[100];
  LONG detect;
  int i, partOut, partIn;
  RGNTECT rgnrcControl;
  /* First aquire rectangles in first region */
  rgnrcControl.ircStart = 1;
  rgnrcControl.crc = 100;
  rgnrcControl.usDirection = RECTDIR_LFRT_TOPBOT;
  GpiQueryRegionRects(hps, rgn1, NULL, &rgnrcControl, &rectArray);
  /* Then compare to see if those rectangles intersect with the second re-
gion */
  partOut = partIn = FALSE;
  for(i=0; i<rgnrcControl.crcReturned; i++) {
    detect = GpiRectInRegion(hps, rgn2, &rectArray[i]);
    partOut = partOut || (detect==RRGN_OUTSIDE) ;
    partIn = partIn || (detect==RRGN_INSIDE);
    if (partOut && partIn || (detect==RRGN_PARTIAL))
        return RRGN_PARTIAL;
  }
  if (partOut)
     return RRGN_OUTSIDE;
  else
     return RRGN_INSIDE;
}
```

LISTING 7.5 A RegionInRegion comparison function.

With such an intersection function, your application could run through its list of display objects and only draw those objects whose regions intersect with the update region.

Using this technique has some problems, however, because even objects that just barely touch the update regions will still be entirely redrawn. This

can cause corruption of other portions of the picture because those objects that touch the update region may be redrawn over other objects that were not redrawn because they were outside the update region. The problem here is that we are not restricting the redraw to just the update region. The next section discusses the topic of clipping, which addresses just this problem.

CLIPPING

Suppose you are developing an Electronic Computer Aided Design (CAD) application that produces very dense layouts of electronic circuits. Since graphics drawn in this application are very dense, it may be somewhat slow to draw them. Next, assume this application allows the user to interactively update or edit portions of the circuit layout. Since the draw times are slow, you may want to limit the redraw of the picture to just the portion of the figure that changed. This will make the screen updates much quicker which is very desirable for an interactive application.

The idea of limiting the area of the display or output device that is updated during a draw operation is known as *clipping*. Clipping essentially relieves you from having to worry about accidentally drawing over portions of the screen that you don't want touched. Figure 7.11 shows how a clip area can restrict drawing output.

By establishing a clipping area, your application can then draw anywhere it wants to, yet only the portion of the drawing that falls into the clipping area will actually be drawn. The clipping area can be defined using either a path or a region. The GPI provides a variety of clipping mechanisms that you can use in your applications. Figure 7.12 shows where in the viewing pipeline the GPI provides mechanisms for clipping drawing output.

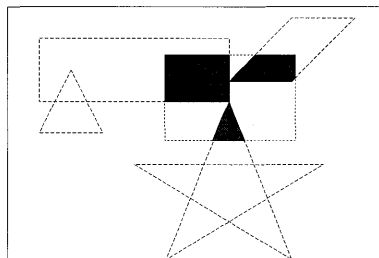


FIGURE 7.11 Restricting display output using a clip region.

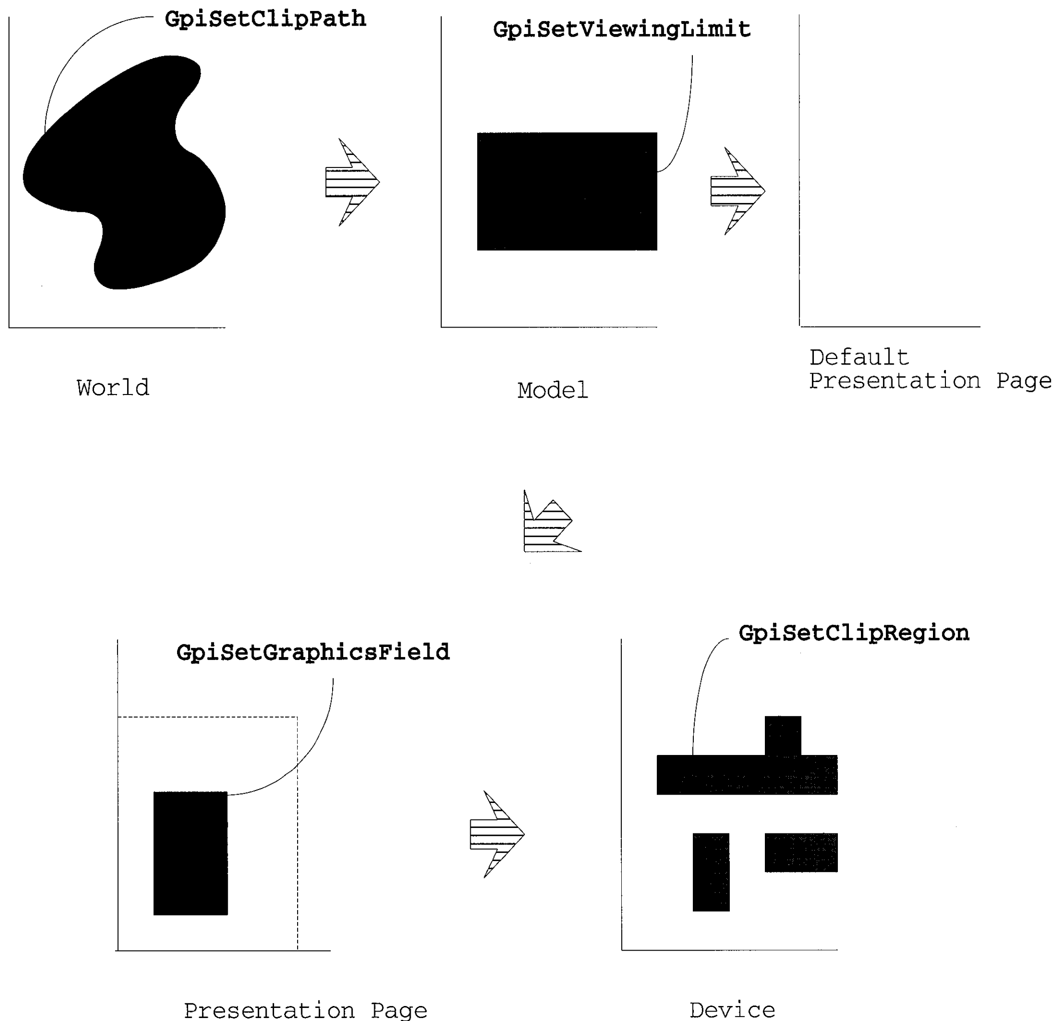


FIGURE 7.12 Clipping mechanisms in the viewing pipeline.

As you can see, there are a number of places that output can be clipped. Each stage provides its own set of unique clipping capabilities. The following sections will explore each of the stages in more detail. Table 7.3 provides an overview of the clipping related functions provided by the GPI.

TABLE 7.3 Clipping functions

Function	Description
GpiSetClipPath	Defines the current clip path.
GpiSetClipRegion	Sets the current clip region.
GpiExcludeClipRectangle	Excludes the given rectangle from the current clip region.
GpiIntersectClipRectangle	Intersects the given rectangle with the current clip region.
GpiOffsetClipRegion	Translates the position of the current clip region.
GpiPtVisible	Checks to see if a point is clipped or not.
GpiRectVisible	Checks to see if a rectangle is clipped or not.
GpiQueryClipBox	Returns the tightest rectangle around all clipping definitions.
GpiQueryClipRegion	Returns the current clip region.

World Space Clipping—Using Clip Paths

The first opportunity for doing clipping is in the world coordinate space. This can be done through the use of something the GPI calls a *clip path*. The general idea here is to take a path definition that you have created and convert it into the current clip path. Then, everything you subsequently draw will be clipped to the interior of the path definition.

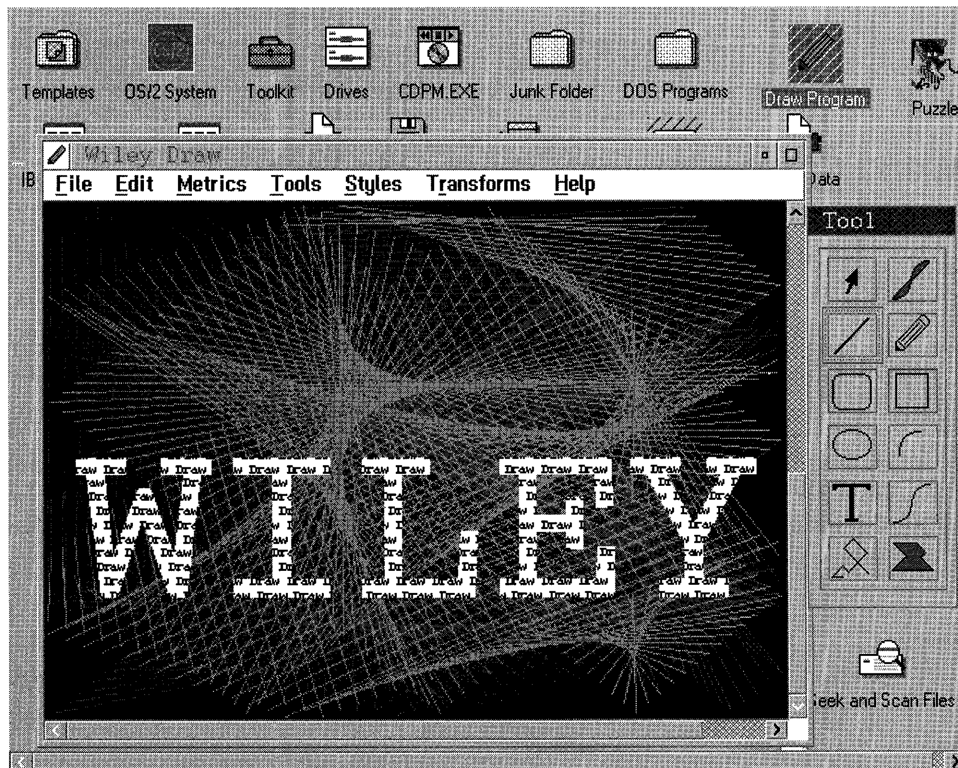
Since paths can be all kinds of shapes, you can produce some pretty interesting clipping effects when using them. That's mainly what clip paths should be used for, interesting clip patterns. For basic clipping (like simple rectangles) there are easier and faster mechanisms. We'll talk more about those later, but for now let's look at path clipping!

The function called *GpiSetClipPath* will convert the current path definition to a clipping area. Since paths can take on many shapes (including arcs, splines, and even outline text) the clipping capability of using paths is very extensive. This function has two parameters of interest: The *lPath* parameter lets you control whether the clip path is to be reset to infinity (no clipping) or intersected with the current path definition, and the *lOptions* parameter con-

trols two things, how the interior of the clip path is determined and how the new path is to be intersected with the current clip path.

The interior of the clip path can be determined using SCP_ALTERNATE or SCP_WINDING modes (see Determining the Interior of an Area in Chapter 3 for more discussion on this). How the new path is intersected is really just a mirror of how the lPath parameter was set (SCP_RESET or SCP_AND). The introduction screen of the graphics editor (Screen 7.1) shows several good examples of the effect that can be produced using path clipping.

My first reaction to this introduction screen was "Hey, that's cool! I wonder how he did that." The answer to the question is actually quite simple. This screen was produced by using clip paths in just a few different ways. The first thing you notice is the outlined drawing of the text "WILEY". This was done by drawing the text into a path definition and then outlining the path using GpiOutlinePath. Next, the "WILEY" text was again



SCREEN 7.1 Graphic editor introduction screen.

drawn into a path definition. The function GpiSetClipPath was then called twice: once to reset the clip path to infinity, and again to define the clipping area to the path containing the "WILEY" text. This allowed us to write the text "draw" over the entire screen. Since the clipping path had been established, however, only those areas inside the path were shown. The rest were clipped automatically by the GPI and discarded. Listing 7.6 shows this part of the introduction screen construction.

```
/****************************************************/
/* Process the animate message.  This will only occur */
/* once after the first WM_PAINT message.         */
/****************************************************/
case WM_ANIMATE:
   /************************************************/
   /* Make the window client area a white background. */
   /************************************************/
   pointsArray[0].x=0;
   pointsArray[0].y=0;
   pointsArray[1].x=WinQuerySysValue(HWND_DESKTOP,SV_CXSCREEN);
   pointsArray[1].y=WinQuerySysValue(HWND_DESKTOP,SV_CYSCREEN);
   GpiBitBlt(hps,hps,2L,pointsArray,ROP_ONE,BBO_AND);
   /***********************************************/
   /* Get size of window and convert in world units. */
   /***********************************************/
   WinQueryWindowRect(hwnd,&rcl);
   GpiConvert(hps,CVTC_DEVICE,CVTC_WORLD,2L,(POINTL *)&rcl);
   xTotal=rcl.xRight-rcl.xLeft;
   yTotal=rcl.yTop-rcl.yBottom;
   /****************************************/
   /* Get a font and adjust its point size. */
   /****************************************/
   lLcid=AddFont("Courier Bold", 0);
   pointSize=115;
   SetPtSize(hps,lLcid,pointSize);
   /*************************************/
   /* Find size of text box for clipping. */
   /*************************************/
   clipTextPoint.x=0;
   clipTextPoint.y=0;
   GpiConvert(hps,CVTC_DEFAULTPAGE,CVTC_WORLD,1L,&clipTextPoint);
   GpiMove(hps,&clipTextPoint);
   GpiQueryTextBox(hps,strlen(clip),clip,TXTBOX_COUNT,textPoints);
   ys=textPoints[0].y-textPoints[1].y;
   xs=textPoints[2].x-textPoints[1].x;
   /**********************************************/
   /* Find the start point so the text is centered. */
   /**********************************************/
   clipTextPoint.x=xTotal-xs;
   clipTextPoint.y=yTotal-ys;
```

LISTING 7.6 Producing the graphic editor introduction screen—part 1 (draw.c).

```
   clipTextPoint.x/=2;
   clipTextPoint.y/=2;
   /************************************/
   /* Draw an outline of the clip area. */
   /************************************/
   GpiMove(hps,&clipTextPoint);
   GpiBeginPath(hps,1L);
   GpiCharString(hps,sizeof(clip),clip);
   GpiEndPath(hps);
   GpiOutlinePath(hps,1,0);
   /**********************************************************/
   /* Create the path again and define as the clipPath so small  */
   /* text can be drawn in it.                                   */
   /**********************************************************/
   GpiMove(hps,&clipTextPoint);
   GpiBeginPath(hps,1L);
   GpiCharString(hps,strlen(clip),clip);
   GpiEndPath(hps);
   GpiSetClipPath(hps,0L,SCP_RESET);
   GpiSetClipPath(hps,1L,SCP_AND);
   /*********************************************/
   /* Change font and set its size to be small. */
   /*********************************************/
   lLcid=AddFont("Courier",0);
   pointSize=6;
   SetPtSize(hps,lLcid,pointSize);
   GpiSetCharSet(hps,0);
   /******************************************************/
   /* Find out size of string to draw in clip path. Note      */
   /* that the string is repeating 5 characters and           */
   /* shifted 1 character each time we start a new line.      */
   /******************************************************/
   GpiMove(hps,&clipTextPoint);
   GpiQueryTextBox(hps,5,clipPattern,TXTBOX_COUNT,textPoints);
   worldHeight=textPoints[0].y-textPoints[1].y;
   strWidth=textPoints[2].x-textPoints[1].x;
   xs=xs/strWidth;    // Number of times needed for width.
   ys=ys/worldHeight; // Number of times needed for height.
   smallPoint.x=clipTextPoint.x;
   smallPoint.y=clipTextPoint.y;
   /*********************************************/
   /* Draw the small text inside the clip path. */
   /*********************************************/
    for(linesUp=0;linesUp<=ys;linesUp++){
      for(words=0;words<=xs;words++)
         GpiCharString(hps,5,clipPattern+(linesUp % 4));
      smallPoint.y+=worldHeight;
   GpiMove(hps,&smallPoint);
}
```

LISTING 7.6 (Continued).

The second part of the screen is the string art. This was produced in a similar manner except, instead of having the clip path set to the interior of the "WILEY" text, here we set the clip path to be everything else but the interior

of the text. How did we do this? The answer is not obvious at first, but is actually quite simple. The solution was to take advantage of the clipping path interior detection option SCP_ALTERNATE. This option determines the interior by passing a line from any point toward infinity. If there are an odd number of path boundary crossings, the point is inside the clip path; if there are an even number of path boundary crossings, the point is outside the clip path. Well, if the path were set to just the "WILEY" text, then obviously just the text internals would be inside the clip path (as in the first part of the screen). To get the opposite effect, we merely draw another box in the path definition that surrounds the entire area that we want the string art to appear in (in this case the whole display). This is done with a simple GpiBox call that starts at the lower-left corner and goes to the upper-right corner of the output area. Now, since the box covers the entire output area, everything is inside the path definition, except the interior of the text (i.e., points on the interior of the text now have an even number of boundary crossings and are thus *outside* the path). Once the clip path is set, the string art can be drawn and it will show up everywhere except inside the "WILEY" text. Listing 7.7 shows the code that draws the second part of the introduction screen.

```
/*****************************************/
/* Get a font and adjust its point size. */
/*****************************************/
lLcid=AddFont("Courier Bold",0);
pointSize=115;
SetPtSize(hps,lLcid,pointSize);
/*******************************************************/
/* Set the clip path again so lines can be drawn around. */
/* the first clip path.                                  */
/*******************************************************/
GpiBeginPath(hps,1L);
startPoint.x=rcl.xLeft;
startPoint.y=rcl.yBottom;
GpiMove(hps,(POINTL *)&startPoint);
startPoint.x=rcl.xRight;
startPoint.y=rcl.yTop;
GpiBox(hps,DRO_OUTLINE,&startPoint,0L,0L);
GpiMove(hps, &clipTextPoint);
GpiCharString(hps,strlen(clip),clip);
GpiEndPath(hps);
GpiSetClipPath(hps,0L,SCP_RESET);
GpiSetClipPath(hps,1L,SCP_AND);
GpiBitBlt(hps,hps,2L,pointsArray,ROP_ZERO,BBO_AND);
```

LISTING 7.7 Promoting the graphic editor introduction screen—part 2 (draw.c).

```
/****************************************************/
/* Initialize start point and offsets for string art. */
/****************************************************/
startPoint.x=xTotal/2;
startPoint.y=yTotal/5;
endPoint.x=xTotal/4;
endPoint.y=yTotal/6;
xStartOff=xTotal/-60;
yStartOff=yTotal/50;
xEndOff=xTotal/40;
yEndOff=yTotal/-70;
/*******************************/
/* Do string art for 4 colors. */
/*******************************/
for(color=1;color<=3;color++){
   GpiSetColor(hps,color);
   for(linesUp=0;linesUp<=280;linesUp++){
      if((startPoint.x+xStartOff)>xTotal)xStartOff*=-1
   if((startPoint.x+xStartOff)<1)xStartOff*=-1;
   if((startPoint.y+yStartOff)>yTotal)yStartOff*=-1
   if((startPoint.y+yStartOff)<1)yStartOff*=-1;
    if((endPoint.x+xEndOff)>xTotal)xEndOff*=-1;
   if((endPoint.x+xEndOff)<1)xEndOff*=-1;
   if((endPoint.y+yEndOff)>yTotal)yEndOff*=-1;
   if((endPoint.y+yEndOff)<1)yEndOff*=-1;
   startPoint.x+=xStartOff;
   startPoint.y+=yStartOff;
   endPoint.x+=xEndOff;
   endPoint.y+=yEndOff;
   GpiMove(hps,&startPoint);
   GpiLine(hps,&endPoint);
   }
}
DosSleep(3000);   // Wait 3 seconds.
GpiSetClipPath(hps,0L,SCP_RESET);
return(MRESULT)TRUE;
```

LISTING 7.7 (Continued).

Model Space Clipping

The second opportunity for doing clipping is in the model coordinate space. Clipping in this space is defined with a single rectangular shape called the *viewing limit*. The viewing limit is used to restrict drawing of a picture as it passes through model space to only those portions that fall inside the rectangle.

The viewing limit defaults to infinity(i.e., no model space clipping). You can change the viewing limit using a function called *GpiSetViewingLimits*. This function accepts a rectangle whose boundaries are, of course, specified

in world coordinates. The boundaries of the rectangle are defined to be inside the viewing limit; therefore, points on them will not be clipped.

Page Space Clipping

Clipping can also be done as pictures pass through the page space. Here again, clipping is defined using a single rectangular shape. This clipping boundary is called the *graphics field*. It functions very much like the viewing limit, except it works in the page space. This function is basically an opportunity to perform the same type of clipping after the picture construction is complete.

By default, no clipping will be performed. You can change the graphics field using a function called *GpiSetGraphicsField*. This function accepts a rectangle whose boundaries are specified in page coordinates. The boundaries of this rectangle are also inclusive so points on them will not be clipped.

Device Space Clipping

Finally, we get to clipping in device space. Aside from clip paths, device clipping is probably the most interesting of the lot. Clipping in device space is done using a series of rectangular shapes that are collectively known as the *clip region*. If you recall from earlier in this chapter, a region is simply a collection of rectangular shapes that have been logically combined. The clip region is simply a particular region that has been chosen to be used for clipping.

As you can see from the previous section, clipping with paths is very powerful and can be used to produce many interesting effects. However, clip paths are also difficult to work with and are quite slow. Regions, on the other hand, are quite easy to work with and their use in clipping is much faster. The function called *GpiSetClipRegion* selects a specified region as the current clip region. All subsequent drawing calls will be clipped to the interior of the region.

The clip region operates slightly differently than the clip path. The clip region merely refers to a region definition that is currently selected. Think of it as a pointer to one of the current region definitions which can be changed by calling the GpiSetClipRegion function. Regions can be selected, and then later deselected, as the clip region without being destroyed (as occurs with clip paths). Also, the GpiSetClipRegion function does not combine the new

region with the old clip region (that can be accomplished through other region functions).

As mentioned earlier, our graphic editor makes heavy use of regions. Basically, all editing operations use clip regions for updates to the display. We considered using clip paths for this, but concluded that for what we were doing they would be more cumbersome to work with than clip regions and probably much slower.

Keep in mind that many of the region manipulation functions cannot be performed on the currently selected clip region. Because of this, you will want to perform most region manipulations before you select that region as the current clipping region.

Other Clipping Related Operations

The remaining clip functions exist to support the previously described clipping definitions. The function called *GpiExcludeClipRectangle* will update the current clipping region such that it excludes the rectangle specified on the call. In other words, the clip region is updated such that things drawn inside the specified rectangle will be clipped and will not be sent to the output device. Anything that is drawn outside the specified rectangle will be clipped according to the clipping definition as it existed prior to this call. The rectangle to be excluded is specified in world coordinates.

Another way to further restrict the clipping region is by using the function called *GpiIntersectClipRectangle*. This function updates the current clipping region such that things drawn outside the specified rectangle will be clipped and not sent to the output device. Anything that is drawn inside the specified rectangle will be clipped according to the clipping definition as it existed prior to this call. Again, the rectangle to intersect is specified in world coordinates. This is shown in Figure 7.13.

One last function for altering the current clip region is called *GpiOffsetClipRegion*. This function simply moves (or translates) the current clip region definition by a specified offset. The offset contains X and Y translation values that are specified in world coordinates.

If you're looking to find out information about the current clip region, there are two functions that can help. The function called *GpiQueryClipRegion* will return the handle of the currently selected clip region. With this, you can call GpiQueryRegionRects to determine the rectangles that

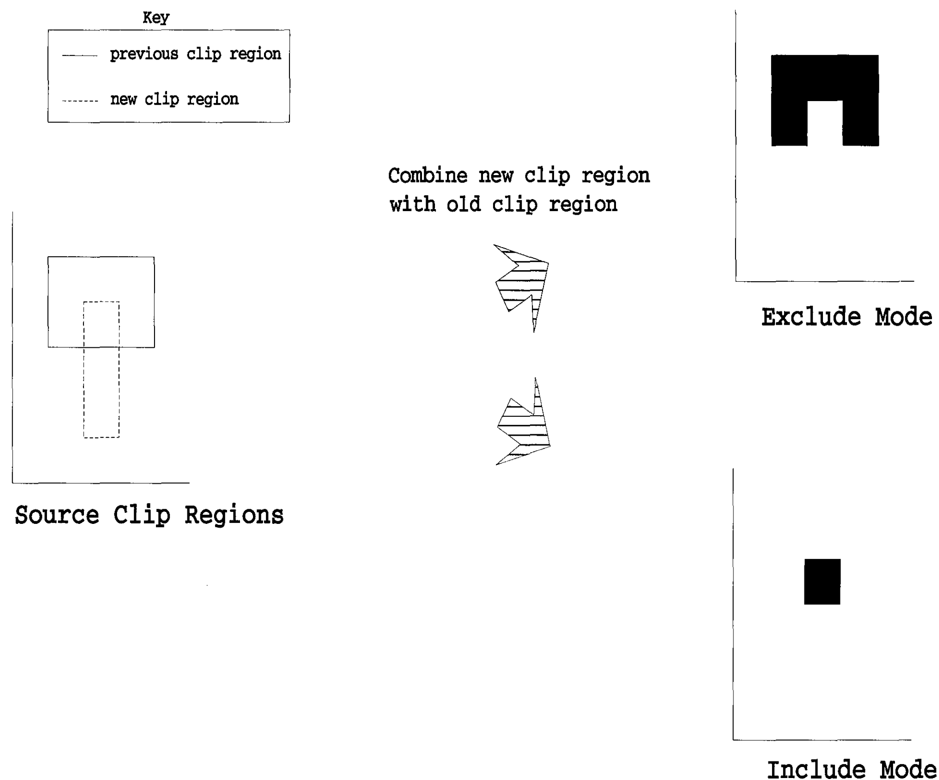


FIGURE 7.13 Modifying current clip region definition.

define that region. Note that you cannot call GpiQueryRegionRects while the region is currently selected as the clip region. You must first deselect it and then query it.

A second function called *GpiQueryClipBox* is useful for determining the bounding rectangle around the intersection of the current clipping definitions. This function computes the tightest bounding rectangle and returns it in world coordinates. Note that this function includes all the clipping definitions (i.e., clip path, viewing limit, graphics field, and clip region). Figure 7.14 shows an example of how the clip box is determined.

Once clipping definitions have been set, you may want to know whether a particular item is visible. One way to find the answer would be to query the clip region, deselect the clip region, check to see if the item is inside the

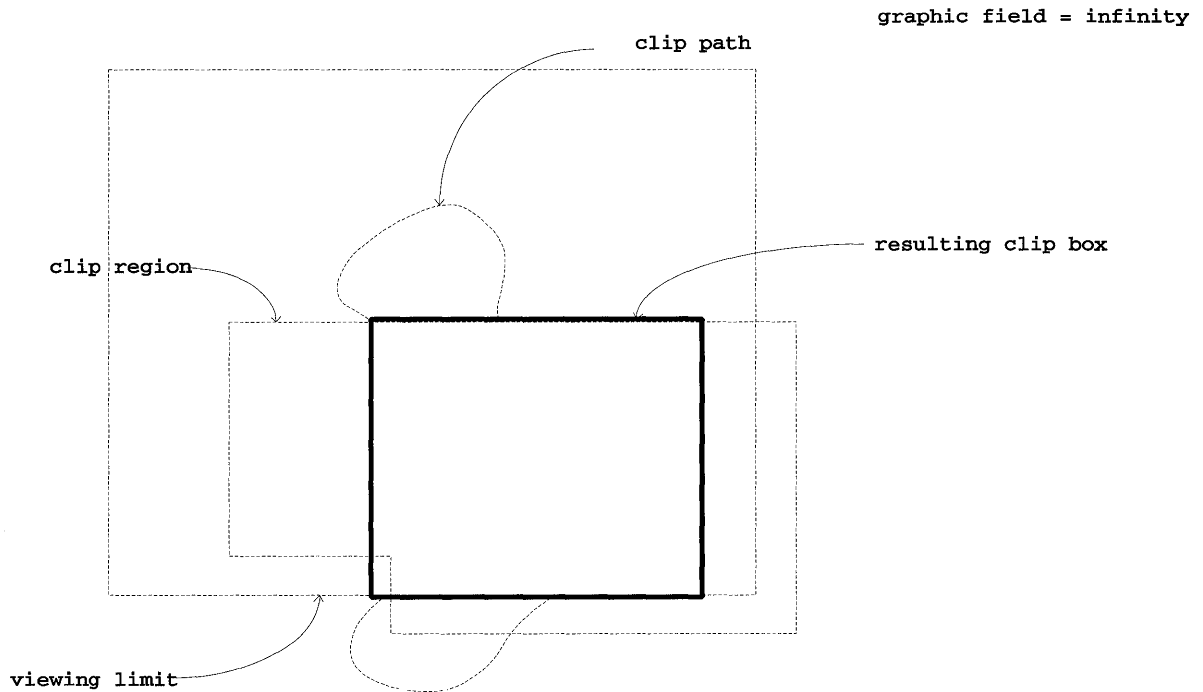


FIGURE 7.14 Determining clip box boundaries.

region (i.e., using GpiPtInRegion or GpiRectInRegion), and finally reselecting the clip region. What a pain—and it gets even worse if you want to check the effects of the other clipping definitions. Well, fortunately the GPI provides a couple of functions that help greatly. The function *GpiPtVisible* will check to see if the specified point is within the clipping area. If it is, that point will be part of the area of the output device that gets updated. The point is specified in world coordinates. The function includes all clipping definitions in its calculation.

The function called *GpiRectVisible* works in a similar fashion except that it accepts a rectangular region instead of a point. The rectangle boundaries are specified in world coordinates.

Boundary Accumulation

By now you should have a basic understanding of what paths and regions are, and also some idea of how they are used in clipping. One of the nice

things about paths is that their boundaries are computed automatically as you call GPI drawing primitives. You don't have to keep track of the boundaries, the GPI does it for you. This can be really handy, especially when the drawings are complicated and include curves, rotations, and the like. You may be wondering if the GPI can also automatically compute the boundaries of a region. Well, the answer is no (at least not directly). The GPI does, however, provide a partial solution through a mechanism called *Boundary Accumulation*. This mechanism allows you to determine the bounding rectangle that surrounds a collection of GPI drawing primitives. Figure 7.15 shows the boundary information collected around a series of drawing primitives.

You start the process of boundary accumulation using the function called *GpiSetDrawControl*. This function can be used to turn on several drawing options. The one we are interested in is DCTL_BOUNDARY. This option controls whether boundary accumulation information is to be gathered during subsequent drawing operations. If this option is set to DCTL_ON, then boundary accumulation information will be collected. The default value is DCTL_OFF.

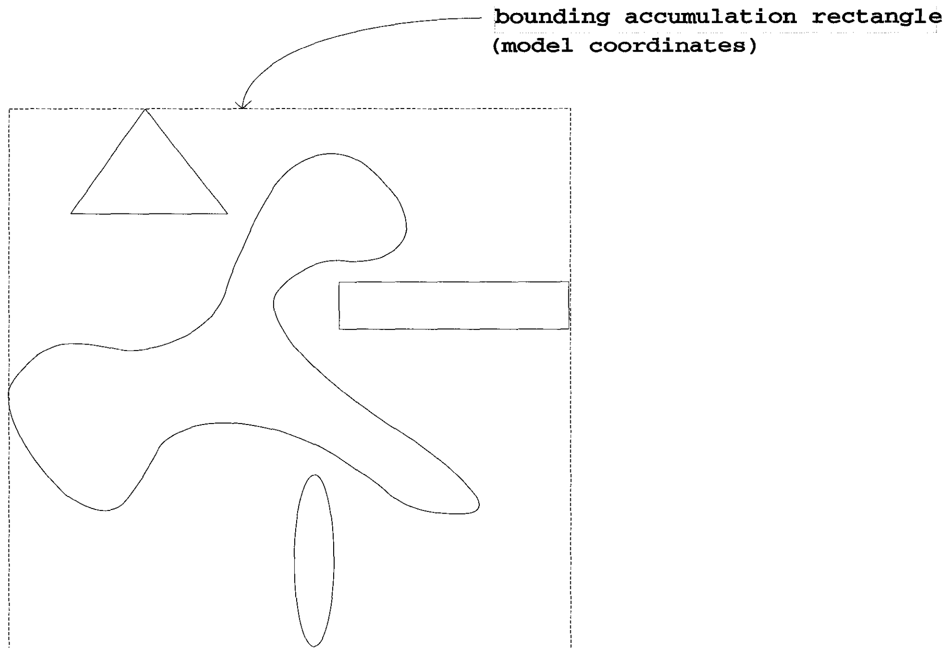


FIGURE 7.15 Gathering boundary information.

Once boundary accumulation has been turned on, the function called *GpiResetBoundaryData* can be used to initialize the boundary data to null. After this, all GPI drawing operations will update the current boundary accumulation definition.

Once you have performed all the desired operations, you can retrieve the boundary information by using the function called *GpiQueryBoundaryData*. This function returns the coordinates of the rectangle that bounds the GPI drawing primitives you issued. Be careful, the coordinates are returned in **model** coordinate space so if you need them in world or presentation page coordinates, you will have to convert them.

Our graphic editor uses boundary accumulation to determine the bounding box around each object. One of the uses of this boundary information is used to draw the dashed box around the object when it is selected. Listing 7.8 shows how we collect the boundary information on each object as it is drawn.

```
/*********************************************************/
/* Gather Boundary data (in model coordinates)           */
/*********************************************************/
GpiResetBoundaryData(hps);
GpiSetDrawControl(hps, DCTL_BOUNDARY, DCTL_ON);
GpiSetDrawControl(hps, DCTL_DISPLAY, DCTL_OFF);
GpiDrawSegment(hps, self->Segment);
GpiQueryBoundaryData(hps, &boundary);
GpiSetDrawControl(hps, DCTL_BOUNDARY, DCTL_OFF);
GpiSetDrawControl(hps, DCTL_DISPLAY, DCTL_ON);
if (self->State == GOS_SELECTED) {
   /*********************************************************/
   /* If object is selected, reopen the segment and add an    */
   /* element to display a dashed bounding rectangle around   */
   /* the outside of the object. Use the boundary information */
   /* just gathered to do this.                               */
   /* Insert this element into the segment before the model   */
   /* xform matrix is changed. This is done because we want   */
   /* the boundary to be affected by the current model xform  */
   /* but not the object specific model xform.                */
   /*********************************************************/
   GpiOpenSegment(hps, self->Segment);
   GpiSetElementPointer(hps, 4L);  // insert beform model xform is changed
   GpiBeginElement(hps, GO_ELEMENT_TYPE, "Bounding Select Rectangle");
   point.x = boundary.xLeft;
   point.y = boundary.yBottom;
   GpiMove(hps, &point);
   point.x = boundary.xRight;
   point.y = boundary.yTop;
```

LISTING 7.8 Gathering boundary information in the graphic editor (object.c).

```
    GpiSetLineType(hps, LINETYPE_DOT);
    GpiSetColor(hps, CLR_BLACK);
    GpiSetTag(hps, 0L);          // make boundary unpickable
    GpiBox(hps, DRO_OUTLINE, &point, 0L, 0L);
    GpiEndElement(hps);
    GpiPop(hps, 3L);
    GpiCloseSegment(hps);
}
```

LISTING 7.8 (Continued).

Note that boundary accumulation can be performed even though the objects are not actually being displayed.

CORRELATION

One issue that comes up with any interactive graphics program is that of getting input from the user. The Presentation Manager has many standard facilities that your program can use for this purpose. For example, pull-down menus are handled with a resource file and messages. Likewise, keystrokes, hot-keys, and scroll bars all send messages to the main window procedure of your application so that it can handle these activities. But, how does an application process input (such as mouse activity) where the user is trying to interact directly with the graphic drawn in the client window?

Well, certainly one way is for the application to simply receive the mouse movement and mouse button events. Then, based on the coordinates returned, the application can scan through some sort of internal data structure it maintains to identify what graphic object is being selected by the user. In fact, if the application were to maintain a region definition for each graphic object it drew, it could merely loop through the list of regions using the GpiPtInRegion function to determine what object the mouse was over. Our graphic editor could easily have used this technique to determine when objects are selected by the user. We didn't, however, because the GPI provides another mechanism that is even simpler and more powerful. This mechanism is called *correlation* (also known as *picking*). Correlation can be performed on graphics primitives within retained or nonretained segments.

Correlation allows you to correlate a single point with a group of graphics primitives that you have drawn. Depending on how fancy you want to be, correlation will allow you to detect general areas on a drawing or specific items within the drawing. Figure 7.16 shows how you can use correlation to identify various granularities of picture elements within a drawing.

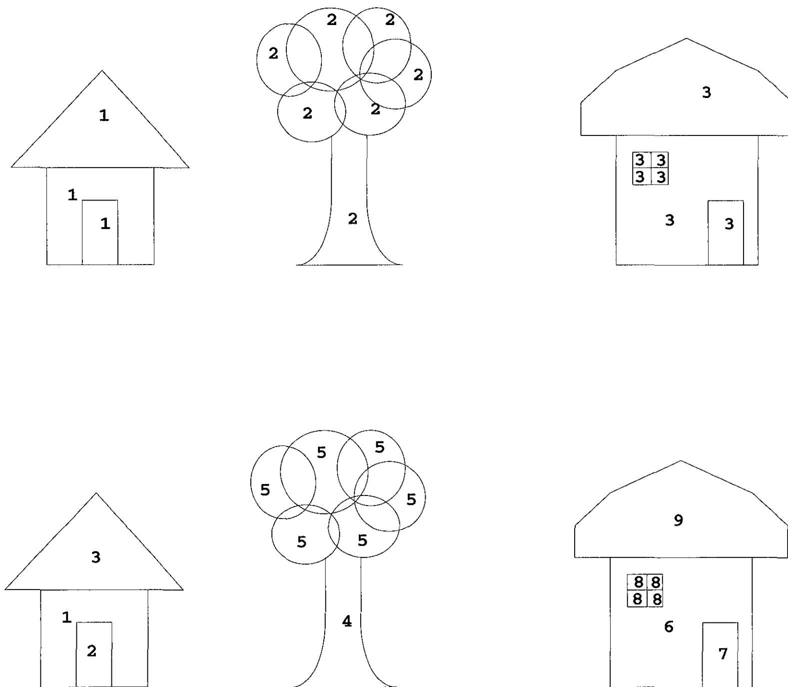


FIGURE 7.16 Varying degrees of correlation granularity using tags.

Essentially, each drawing primitive has an associated *tag* value. When you perform a correlation operation on a particular point, the GPI will determine if it intersects with (i.e., hits) any of the primitives that were drawn. The correlation operation returns the unique tag values of primitives that were found to be intersected. More than one primitive can be drawn using the same tag; therefore, multiple points on the picture may correlate to the same tag value. On the other hand, each drawing primitive can have its own unique tag to make it uniquely identifiable; therefore, if objects overlap, a single point may correlate to multiple tag values.

As we mentioned earlier, correlation is only provided within segments. The tag values are associated with drawing primitives by first setting the *current tag value* and then issuing the drawing commands. The current tag is set using the function called *GpiSetTag*. Once the tag has been set, all subsequent drawing primitives in the segment will take on that tag value. Later

correlation of points on those drawing primitives will return the same tag value. The current tag value can be changed multiple times within a segment. This allows you to have different portions of the picture with different tag values. Those items drawn with a tag value of zero will not be detected by any subsequent correlation operations. Listing 7.9 shows how multiple tag values can be assigned to various parts of the picture.

```
GpiOpenSegment(hps, MYSEG);
GpiSetTag(hps, 1L);
/* Draw the house with tag=1 */
GpiMove(hps, &housePt[0]);
GpiPolyLine(hps, 4L, &housePt[1]);
GpiSetTag(hps, 2L);
/* Draw the door with tag=2 */
GpiMove(hps, &doorPt[0]);
GpiBox(hps, DRO_OUTLINE, &doorPt[1], 0L, 0L);
GpiSetTag(hps, 3L);
/* Draw the roof with tag=3 */
GpiMove(hps, &roofPt[0]);
GpiPolyLine(hps, 3L, &roofPt[1]);
GpiCloseSegment(hps);
```

LISTING 7.9 Assigning tags to the drawing.

After the picture has been drawn, a point can be correlated against the picture using the GPI function *GpiCorrelateChain*. This function will return the data for any tagged primitive with which the point intersects. The returned data includes the tag value of the drawing primitive with which it was intersected, and the segment in which the drawing primitive was included. GpiCorrelateChain only examines segments that have the *chained* attribute set (this means that either they are included in the GPI's root segment chain or they are called by another segment that is). You can also specify that only those segments that are visible should be included in the correlation operation.

As mentioned earlier, a single point may correlate to several tagged drawing orders. The GPI refers to each of these correlation matches as *hits*. Thus, a single GpiCorrelateChain call can encounter several hits and return correlation data for each of them. Your application can specify the maximum number of hits it is willing to accept using the *lMaxHits* parameter. The hit data is then returned to your application as an array of segment/tag pairs. Because segments can contain calls to other segments, there is often a segment call hierarchy. Figure 7.17 shows an example of such a segment call hierarchy.

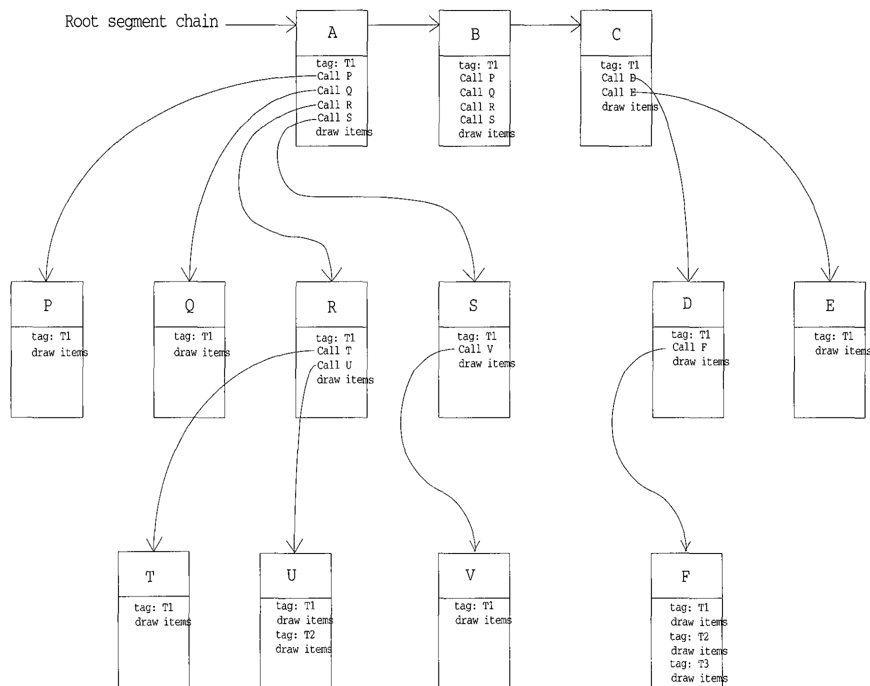
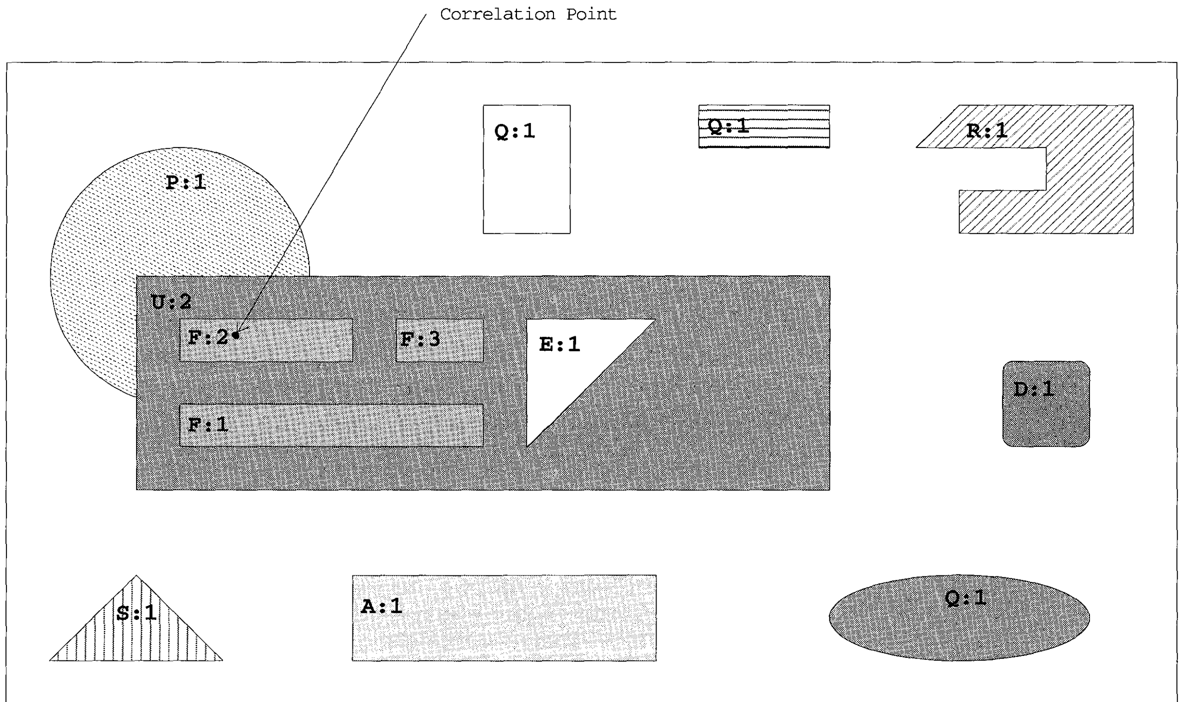


FIGURE 7.17 Segment call hierarchy.

Referring to Figure 7.17, if a hit is detected in segment T, you can instruct the GpiCorrelateChain function to report the segment call path. This is done using the *lMaxDepth* parameter. This parameter tells the GPI how far up the segment call tree should be reported for each hit. Therefore, if the lMaxDepth parameter were set to 3, a hit in segment T would report segment/tag pairs for segments T, R, and A.

Together, lMaxHits and lMaxDepth can be used to control how many hits are detected and how much segment hierarchy information is reported for each hit. The hits are reported in the reverse order they are encountered in the segment chain flow and the hierarchy information is reported from the point of detection on up. Figure 7.18 shows several examples of hit data that are returned from GpiCorrelateChain using various parameters of lMaxHits and lMaxDepth. The figure was drawn using the segment hierarchy shown in Figure 7.17. The point being correlated happens to hit drawing orders in segments F,U, and P.




```
Max Hits=1, Max Depth=1        Max Hits=1, MaxDepth=3
     F:2                            F:2
                                    D:1
                                    A:1

Max Hits=2, Max Depth=2        Max Hits=3, Max Depth=1
     F:2                            F:2
     D:1                            U:2
     U:2                            P:1
     R:1
```

FIGURE 7.18 Correlating multiple hits.

If you do not want to scan the entire segment chain, you can limit the search using the GPI function *GpiCorrelateFrom*. This function lets you specify the beginning and ending segments in the chain to be included in the

correlation operation. Other than this, the GpiCorrelateFrom function operates just as the GpiCorrelateChain function.

Finally, you can perform correlation against segments that are not in the root chain at all. This can be done with the GPI function called *GpiCorrelateSegment.* This function performs the correlation operation against the specified segment (and any that it calls). Again, the correlation operation is done basically the same as the GpiCorrelateChain function.

The GPI refers to the point that you are correlating against the picture as the *pick aperture*. In fact, the pick aperture does not have to be a point at all—your application can define it to be a larger size if it wishes. A larger pick aperture means that a larger area around the point will be checked for intersection with tagged drawing primitives. The GPI function *GpiSetPickApertureSize* is used to define the size of this area. The center of the pick aperture is defined using the GPI function called *GpiSetPickAperturePosition.*

These functions are typically used to understand input that came from the user via some sort of pointing device. Using correlation you can determine if the user was trying to interact with a portion of the picture. By setting a larger aperture, the pointing device essentially becomes larger and makes it easier to pick picture items (which can be especially nice for those users who are a little less steady with the mouse).

Consider, for a moment, what your application would have to do if the correlation functions were not available. Your application would need to manage its own correlation lookup table which would need an entry for each drawing primitive that was issued. Associated with each primitive would need to be the boundary information of the primitive, the segment it was drawn in, and the tag value that you assigned it. Then, when a point or region was to be correlated, you would need to scan the table to locate any primitives that intersected with it. Simply scanning a linear table would be awfully slow; therefore, you would probably want to organize the table for more efficient scanning. In addition, you would want to provide some sort of linking between table elements in order to determine call hierarchy between elements in one segment to elements in another segment....as you can see, the complexity rapidly grows as you start adding the various features we've previously discussed. The end result is that correlation can give your application a lot of flexibility in dealing with user input and save you a lot of work.

Correlation in the Graphic Editor

In the graphic editor, we use correlation for all interaction between the user and the objects drawn to the display. We take special care to tag the segments in a way that allows us to identify what object is being interacted with and what portions of that object are being manipulated. The following paragraphs describe how the editor lays out segments and how it uses tags to facilitate end-user editing operations.

The editor places each graphic object that the user creates in its own segment. Each object has both an in-memory data structure (i.e., an object) and a retained segment associated with it. The in-memory object holds the ID of the segment as one of its fields. Plus, the segment itself holds a pointer back to the in-memory object in one of its comment fields (a comment field can be used to hold anything, so we use it to hold an object pointer). This means that given one structure, we can get to the other structure fairly easily and quickly.

As object segments are drawn, all pickable portions of the segment will be drawn with a tag value of one or higher (this guarantees that they can be picked). Special points on the object, such as editing handles, will be drawn with tag values greater than one. Editing points on an object begin with a tag value of two and go up incrementally till the last point is reached. Listing 7.10 shows the function we use to draw edit handles for all multipoint objects.

```
/***********************************************************************/
/*                                                                     */
/*                          Draw Handles                               */
/*                                                                     */
/* This function draws the edit handles of the object. An Edit handle is*/
/* drawn at each point of the object.                                  */
/*                                                                     */
/***********************************************************************/
 void GoDrawHandles(GOBJ self)
{
   int i;
   for (i=0; i<self->PointCount; i++) {
      GpiSetTag(hps, i+2);              // Tags start at 2 and go up
      GpiMarker(hps, &((*self->Points)[i]));
   } /* endfor */
}
```

LISTING 7.10 Assigning correlation tags to edit handles in the graphic editor (object.c).

When a pointer event (i.e., mouse event) is received from the user, we correlate that point against all the segments in the chain. If the point intersects with a picture, the hit information returned to us will tell us which segment was hit. Since we store a pointer to our object in a comment field at the beginning of the segment, we can quickly get a pointer back to the object with which the segment is associated. Listing 7.11 shows how we use correlation to find the segment that was hit and, in turn, determine which object was picked.

```
/***************************************************************************/
/*                                                                         */
/*                             Pick                                        */
/*                                                                         */
/* This function performs a GPI correlate operation to determine if the    */
/* specified point is over any objects. If any objects are found, their    */
/* handles are stored in the given object handle array. The function       */
/* returns the number of 'hit' objects returned. The tag id of the hit     */
/* object is also returned in the given tag array.                         */
/*                                                                         */
/***************************************************************************/
#define MAX_PICKS 15                       /* Maximum pick depth of 15 items*/
int GoPick(POINTL point, GOBJ objArr[], LONG tagArr[], int arrSize)
{
   LONG corrSegTags[MAX_PICKS*2];   // Segment/Tag pair array used in correlation
   LONG hits=0;                     // Number of hits found during correlation
   GOBJ hitObj;                     // The graphic object that was hit in correla-
tion
   LONG hitTag;                     // The tag within the segment that was hit
   LONG hitSeg;                     // The segment that was hit in correlation
   int  i;                          // general loop index
   LONG maxHits;                    // Maximum number of allowable hits
   extern GOL displayList;          // The graphic object display list
   extern BOOL fastState;           // Flag if fast correlate is to be used.
   POINTL pagePoint;                // an X,Y point in page coordinates.
   RECTL boundary;                  // boundary to do fast correlation within
   maxHits = MIN(arrSize, MAX_PICKS);
   if (fastState && (maxHits==1))
{
/******************************************************************************/
/* Use quick scan method to locate objects being picked. This technique      */
/* scans the display list to find potential matches and then uses            */
/* correlation on those objects to determine if a true hit.                  */
/* This is done since correlating the entire chain can be slow.              */
/******************************************************************************/
displayList->Do->Bottom(displayList);
while (displayList->Do->GetPrev(displayList, &hitObj))
{
/* Examine each object in display list, checking for hits */
```

LISTING 7.11 Using correlation to identify objects in the graphic editor (object.c).

```
GpiQueryRegionBox(hps, hitObj->Do->GetRegion(hitObj), &boundary);
if (GpiPtInRegion(hps, hitObj->Do->GetRegion(hitObj), &point) == PRGN_INSIDE) {
/* Found a potential hit, use correlate to tell for sure */
pagePoint = point;
GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &pagePoint);
hits = GpiCorrelateSegment(hps, hitObj->Do->GetSeg(hitObj),
   PICKSEL_VISIBLE, &pagePoint, maxHits, 1L, corrSegTags);
hits = MIN(hits, maxHits);
if (hits>0) {
/* Got a hit! record object and tag that were hit */
for (i=0; i<hits; i++) {
   hitSeg = corrSegTags[i*2];
   /* Determine hit Segment        */
   hitTag = corrSegTags[i*2+1];
   /* Determine hit Tag ID         */
   /* transfer to object and tag arrays */
   objArr[i] = hitObj;
   tagArr[i] = hitTag;
} /* endfor */
break;
/* Get out of scan loop */
} /* endif hits>0 */
} /* endif PtInRegion */
} /* endwhile GetPrev */
} else
{
/*********************************************************************/
/* Use normal correlate chain technique to locate objects being picked */
/*********************************************************************/
pagePoint = point;
GpiConvert(hps, CVTC_DEVICE, CVTC_PAGE, 1L, &pagePoint);
hits = GpiCorrelateChain(hps, PICKSEL_VISIBLE, &pagePoint, maxHits, 1L, corrSeg-
Tags);
hits = MIN(hits, maxHits);
for (i=0; i<hits; i++) {hitSeg = corrSegTags[i*2];/* Determine hit Segment*/
hitTag = corrSegTags[i*2+1];  /* Determine hit Tag ID*/
                                    /*  Determine hit Object by     */
GpiSetEditMode(hps, SEGEM_INSERT);  /*  extracting it from segment  */
GpiOpenSegment(hps, hitSeg);        /*  comment field (always the   */
GpiSetElementPointer(hps, 2L);      /*  2nd element in the segment) */
GpiQueryElement(hps, 2L, sizeof(GOBJ), (PBYTE)&hitObj);
GpiCloseSegment(hps);               /* transfer to object and tag arrays */
objArr[i] = hitObj;
tagArr[i] = hitTag;
} /* endfor */
} /* endif */
if (hits<0) hits=0;
return hits;
}
```

LISTING 7.11 (Continued).

Along with identifying the segment that was hit, we also get information about what part of the segment was intersected. If the tag value is one then we know it was a basic part of the object. If it is greater than one, then we

know the user is trying to interact with an editing handle on the object. Listing 7.12 shows a simplified flow of how we accept and process mouse events (see the function GoMultiPointProcessEvents in Object.C for a more detailed example).

```
/* Handle incomming mouse messages */
switch (msg) {
   case WM_BUTTON2DOWN:
      /* Button pressed, See if we are over an editing handle */
      ptrPos.x = MOUSEMSG(&msg)->x;
      ptrPos.y = MOUSEMSG(&msg)->y;
      lastPtrPos = ptrPos;
      hits = GoPick(ptrPos, &object, &tag, 1);
      if (hits<1)
         return UNHANDLED;
      if (tag>1) {
         /* Yes we are over a handle so we will begin editing the  */
         /* object it belongs to. Record the edit handle and place */
         /* the object in XOR_DRAW mode */
            editObj->EditPoint = tag-1;
            editObj->DrawMode = XOR_DRAW;
            return BEGIN_EDIT;
         } else
            return UNHANDLED;
         break;
      case WM_MOUSEMOVE:
      /* if we are editing a handle pt, move it with mouse */
         if (editObj->EditPoint!=0) {
            if (memcmp(&lastPtrPos, &ptrPos, sizeof(ptrPos)) == 0)
               /* if ptr position hasn't changed then ignore message*/
               break;
            editObj->Do->Draw(editObj);       /* XOR previous object pos */
            ptrPos.x = MOUSEMSG(&msg)->x;
            ptrPos.y = MOUSEMSG(&msg)->y;
            GpiSetModelTransformMatrix(hps, 9L,
               &editObj->XformMtx, TRANSFORM_REPLACE);
            GpiConvert(hps, CVTC_DEVICE, CVTC_WORLD, 1L, &ptrPos);
            (*editObj->Points)[editObj->EditPoint-1] = ptrPos;
               /* Updates the point */
            editObj->Do->Draw(editObj);       /* XOR new object pos      */
         } else
            return UNHANDLED;
         break;
      case WM_BUTTON2UP:
         /* if we are editing a handle pt, we just finished. */
         if (editObj->EditPoint!=0) {
            editObj->EditPoint=0;             /* clear editing point     */
            editObj->Do->Draw(editObj);       /* XOR previous object pos */
            editObj->DrawMode = FULL_DRAW;
            editObj->UpdateRqd = TRUE;
            return END_EDIT;           /* let caller know were finished */
```

LISTING 7.12 Identifying edit handles using correlation tags.

```
      } else
         return UNHANDLED;
      break;
   default:
      return UNHANDLED;
   } /* endswitch */
      return HANDLED;
```

LISTING 7.12 (Continued).

Due to the way we choose tag values, correlation really simplifies the process of editing an object.

In the early versions of the graphic editor, we noticed that performance degraded as our drawings grew larger. In fact, as drawings grew larger than 50 objects, the performance was so objectionable that we knew something had to be done. So, we searched for what was slowing the program down. We found that two things, correlation and GpiDrawChain, were significant performance inhibitors.

This was somewhat disturbing since correlation and retained segment store were two of the main things we were featuring in our application. So, we searched to see if there were things we could do to improve performance, while still making use of these wonderful things. There were two things that we did that really helped us buy back the performance we wanted. Both of them were related to the fact that we maintain a boundary region for each graphical object in the editor. This boundary region is stored in device coordinates and, as mentioned earlier in this chapter, is used for device space clipping.

The first change we made was to reduce the size of the chain that we draw. Instead of doing a blind GpiDrawChain which covers all segments in the chain, we modified our code to skip through the objects in our display list and draw them only if their boundary region intersected with the current device clipping region. This is where the idea of a GpiRectInRegion function came about. Since we maintain a boundary region for group objects as well as primitive objects, we can bypass potentially huge numbers of segments with a single test that would otherwise have been drawn.

The second change was similar in nature but applied to the correlation activity. Again, rather than blindly applying a GpiCorrelateChain operation on mouse button input, we became more selective. Instead, we again scanned the objects in the display list and tested to see if the mouse pointer was in the boundary region of the object. If it was, then we applied a GpiCorrelateSegment just to that object to see if there was a hit; if not, we proceeded

to the next object in the display list. Since the correlate function is fairly expensive, we were able to reduce our use of it to only those areas where we felt the potential of a hit was fairly high. Again, we reaped the benefit of skipping unneccesary correlation on potentially large numbers of segments within an uninvolved group object.

Thus, maintaining boundary regions for our objects really turned out to be a triple performance treat (clipping, drawing, and correlation). If you are using retained segment storage, you may find a similar technique useful too!

This chapter has covered a variety of GPI topics. Most of them have been related to clipping in the various coordinate spaces of the viewing pipeline. Paths and regions, in particular, are common structures used for clipping. Path clipping is generally more flexible, while region clipping is much faster.

We've discussed definitions of regions through the process of boundary accumulation. In addition, we examined the important topic of correlating user input with your graphic pictures. This is certainly something that most people will run into, especially given the interactive nature of today's applications. We've also talked about some of the limitiations you will run into regarding correlation and performance and provided some alternatives you might find useful.

In Chapter 8, we discuss printing, which will allow you to move all these wonderful graphics to an output device.

CHAPTER 8
Printing Graphics

A common desire of many businesses is to share the printer resources they own. Most programmers who haven't worked with printer output may think this task is easy. But almost any programmer that has dealt with outputting printer data in an environment like DOS will probably tell you that it can be a nightmare. There are several reasons for this. First, it is almost always desirable to allow printing to occur while you still allow the user to perform other activities with their computer. Second, printers of different types have different control codes and functional characteristics that programmers need to understand so they can produce the best-fit output for the specific printer. Third, multiple print jobs may exist for a printer at any one time; hence, you need a way to pace these jobs through the system and manage the correct printer setup. In an environment like DOS where there are few rules and programs tend to step on each other easily, these problems become difficult to deal with.

OS/2, of course, provides an architecture for sharing resources between applications. This already begins to make the task of writing printer support easier. But beyond this, OS/2's Presentation Manager also provides an architecture and functions that help promote device independence. Because of this, you will see that we can write a Presentation Manager program that will allow the user to dynamically pick a locally known printer and direct output to this printer with very little extra programming effort. Before this is

shown, a higher-level view of how print data flows through an OS/2 system is appropriate.

There are several software components that exist in OS/2 that contribute to the control print data. Knowing a few of these key terms and how they relate to each other gives a lot of insight as to how printing works and what features exist in OS/2 printing. These terms are *print queues*, *printer names*, *presentation drivers*, and *port names*. Following is a list of descriptions of these terms, their basic function, and how they relate to each other. As you read these definitions, refer to Figure 8.1 to better see how these terms fit into the printer setup scheme for OS/2.

Print queue: This component is a holding place for print jobs before the print data is routed to the actual printer via a printer presentation driver. Multiple print queues can exist in an OS/2 system which allows for the grouping of jobs by some common characteristic. For instance, a print queue may exist for jobs that are to be printed portrait, while another queue may exist for jobs to be printed landscape. Or perhaps a queue may exist for a certain type of form for a printer. Hence, print jobs that need this form can be grouped in this print queue and held until the printer is set up with the appropriate form. A print queue has a name like LPT1Q and a default print queue processor name like PMPRINT. (The queue processor is the component that takes the print data from the print queue and passes it to the correct presentation driver for printing.) A print queue also has one or more printer names along with a default presentation driver associated with it. The default presentation driver must be one that is associated with the printer name,

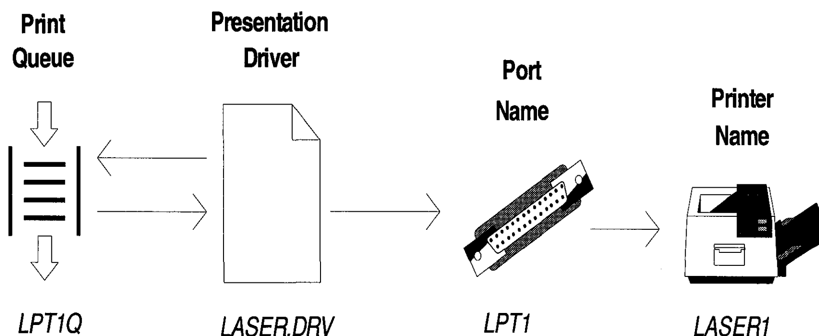


FIGURE 8.1 Printer terminology.

but does not have to be the same as the default presentation driver used by the printer name. Multiple print queues can all reference the same device; this is called *printer sharing*. A single print queue can also be connected to multiple devices; this is called *printer pooling*.

Printer name: This term is a descriptive name given to a physical printer and is used for printer setup. For instance, LASER1 may be the printer name given to a laser printer logically attached to the system. Each printer name also has a port name and a default print queue associated with it. This port name is used by the system to locate the printer device driver that outputs the actual print data to the physical printer. Besides the port and queue name, the printer name is also associated with one or more presentation drivers (one of them being a default).

Presentation driver: This component is responsible for converting information from an application to a format suitable for the physical output device. Hence, this component is key in providing device independence in the OS/2 environment. For instance, the presentation driver takes information about the application's presentation space and the GPI orders directed to it, and generates a printer data stream to produce the desired output. A presentation driver can also be passed information about a print job so it won't convert application data. This is called *raw print mode*. When using raw print mode, the application needs to know exactly how to drive the physical printer. Raw print mode is not encouraged or shown in this book. Because of the way the presentation driver interfaces with the OS/2 Spool Queue Manager, it is divided into two logical parts (however, the presentation driver is typically one physical component). The first part of the presentation driver writes a print file to be passed to the OS/2 Spooler to be queued, while the second part takes a print file from the Spooler and directs the output to the correct printer. In the case of direct printing, which is not shown in the book, the print data is routed directly to the second part of the presentation driver.

Port name: This term is a descriptive name used by the system to locate and route printer data to the correct printer device driver. The device driver is the component that actually communicates to the physical printer.

As implied by the previous descriptions, items like printer name and port name are actually used in the setup process of the OS/2 system and can be

queried by your program so you can pick and choose the printer support that best fits your needs.

Figure 8.2 shows an example of printer data flow in the OS/2 system for a typical Presentation Manager application. As you can see, the Presentation Manager application must create a presentation space and associate it with a valid device context, which it must also create. In the case of outputting graphics data to a display, this device context would be a window device context. But for a printer, the application must create either a queued or non-queued printer device context and associate the presentation space with it. In order to create the printer device context, however, the application must query the system for information about valid printer names, queues, and presentation drivers. (More detail about how to query the system for this type of information will be shown later.) Once the presentation space and printer device context are associated, the application must issue a *start document* device escape sequence to the device context to signal the start of the print job. Then the Presentation Manager application issues the desired GPI orders to the presentation space to produce the desired printer output. Once the graphic orders have been issued, the application must send an *end document* device escape code to the device context to signal the end of the print job. Once the application has done this, it can disassociate the presentation space from the printer device context, destroy the printer device context, and either destroy the presentation space or associate it with another device context if desired. From the Presentation Manager application's point of view, printing is complete. All that has been done from an OS/2 system's point of view, however, is to communicate print job information to a printer presentation driver. That is to say that even though the Presentation Manager application is done with printing from its perspective and can perform other activities, the print data for the print job is most likely in the OS/2 Spooler and has not yet physically printed. (The management of the queued print data is asynchronous to the application that issued the GPI calls to create the print data.)

Depending on the job information passed to the printer presentation driver, the presentation driver will either use the OS/2 Spooler to create a print spool file and place it in the correct print queue, or reroute the printer job data to itself. (If the print job was non-queued, the printer presentation driver will route the data back to itself for immediate printing.) If the print data was sent to a print queue, a queue processor such as `PMPRINT` will

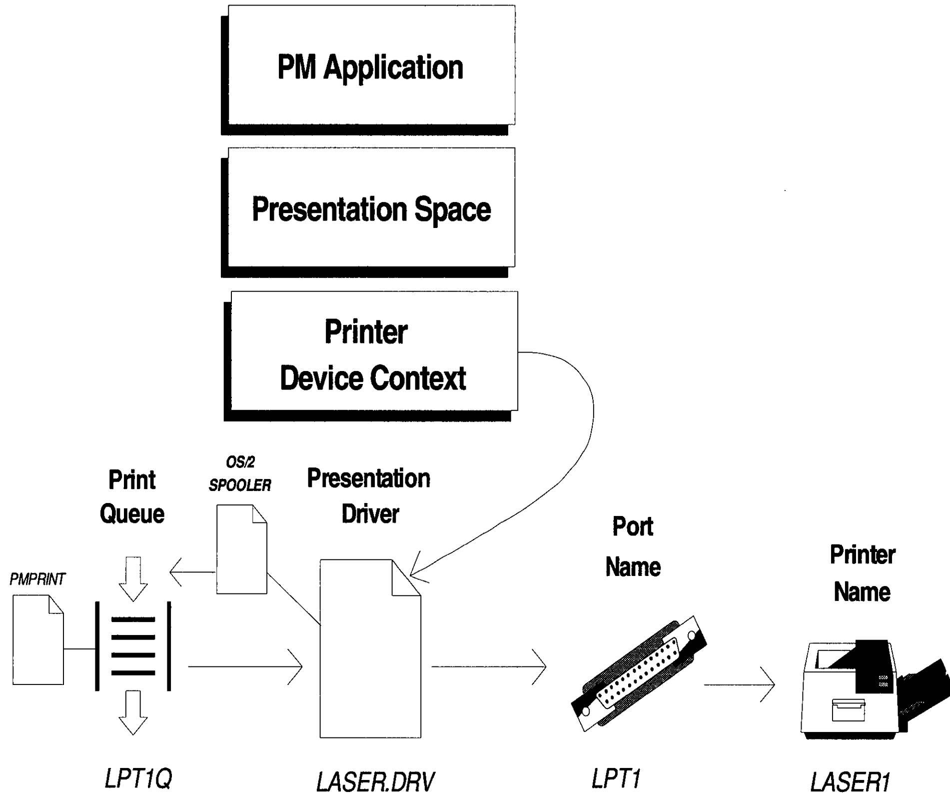


FIGURE 8.2 Presentation manager printer data flow.

take the print file from the print queue at the appropriate time and route it back to the correct printer presentation driver. The printer presentation driver will then take the printer data stream and route it to the correct device driver for printing via the port name.

The data flow just given simplifies the basic path used by Presentation Manager applications because it represents a single vertical path through the system. If you consider, however, the combination of ways a user can configure printers with OS/2 and the flexibility and device independence given by this scheme, you can begin to see how OS/2 is managing a relatively complex problem. Another typical example of an OS/2 printer configuration is shown in Figure 8.3, which depicts a more complex printer setup but still

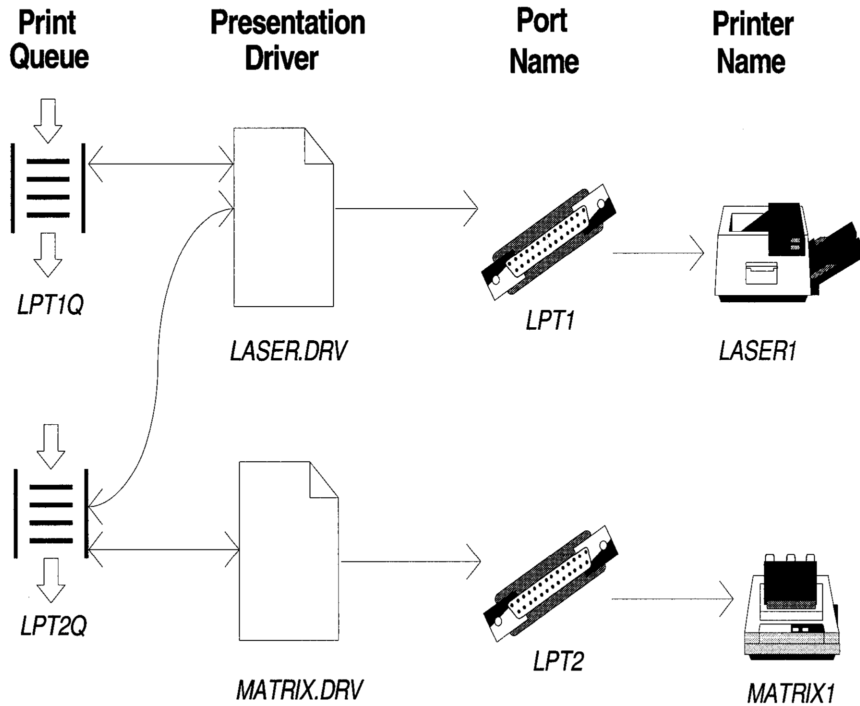


FIGURE 8.3 Complex printer setup.

doesn't show details like how a printer may have multiple presentation drivers. What is important to note, however, is that even though an OS/2 printer configuration can be fairly complex, the data flow between components in OS/2 is consistent and managed outside of the Presentation Manager application.

To help you discover the different print resources available on an OS/2 system, there is a utility program called `PRINTERS.EXE` found on the diskette provided with this book. This program will query the system for information about printer names and printer queues, and writes this information to a file named `WSYSPRT.LST`. It will then display this file with the Browse utility that is also given with this book (`BROWSE.DLL`). Hence, to use the Printers utility, you must also have `BROWSE.DLL` placed in a path where the system can find it. You will also find an icon on the book diskette

called PRINTERS.ICO that you may use to reference this program when using the OS/2 desktop.

Listing 8.1 shows an example of the output produced by the Printers program. To understand what this listing means and to discover how you can query a system for this type of information, we need to briefly review the highlights of the Printers program source code. The source code for the Printers program is shown in Listing 8.2.

```
SYSTEM PRINTER INFORMATION
Date and Time 10/22/1994  20:45:14.9
Printer Names:
   Printer IBM4029(
   HPLaserJ
Printer Path Information:
   Printer -->  LPT2;IBMNULL;Printer;;45;
      Where:Port name -->  LPT2
   Driver names.
   Device names -->  IBMNULL
   Queue names -->  PrinterIBM4029( -->  FILE;
      PSCRIPT.IBM 4029 (39 Fonts 600 Dpi),
      LASERJET.HP LaserJet IIIP;IBM4029(;;45;
         Where:Port name -->  FILE Driver names.
   Device names -->  PSCRIPT.IBM 4029 (39 Fonts 600 Dpi),
      LASERJET.HP LaserJet IIIP
   Queue names -->  IBM4029(HPLaserJ -->  LPT1;
      LASERJET.HP LaserJet IIIP,
      PSCRIPT.IBM 4029 (39 Fonts 600 Dpi);HPLaserJ;;45;
         Where:Port name -->  LPT1 Driver names.
   Device names -->  LASERJET.HP LaserJet IIIP,
      PSCRIPT.IBM 4029 (39 Fonts 600 Dpi)
   Queue names -->  HPLaserJQueue Names: Printer IBM4029( HPLaserJQueue
   Default Presentation Driver: Printer -->  IBMNULL; IBM4029( -->
      PSCRIPT.IBM 4029 (39 Fonts 600 Dpi);
      HPLaserJ -->  LASERJET.HP LaserJet IIIP;
```

LISTING 8.1 Printer utility sample output.

As you can see in Listing 8.2, the Printers program is a fairly small Presentation Manager program. In fact, you may notice that this program doesn't even have a header file or resource file of its own. It does, however, create a window of no dimension. By creating this window, the program has its own window procedure to send messages to even though the window itself is invisible. The window procedure is where the Printers program queries the system for print data, formats the data and writes it to a file, and communicates with the Browse utility about displaying the file. The place where the program provides most of this function is on the WM_CREATE window message for the window with no dimension created. The function is provided here because this message occurs only once right away. The first

thing done during the processing of the `WM_CREATE` message is to initialize some data areas. Then a file is created and opened and the title of the listing is written to it. The OS/2 system is then queried for the current time and date, which is formatted and written to this file. Finally, the subtitle that says printer names are to follow is written and the query function that allows us to discover the needed printer information is used. This function is named *PrfQueryProfileString* and will be used multiple times to extract all the desired printer and queue information.

The PrfQueryProfileString function is used to obtain profile information from either system or user applications that exist. The first parameter of this function is the handle of the initialization file that contains the desired information. In the case shown in Listing 8.2, the handle `HINI_PROFILE` is used, which informs this function to search both system and user profiles. The second parameter for this function is a pointer to a string which is the name of the application for which profile data is being requested for. As you can see from Listing 8.2, the application name we want to use for obtaining printer name information is `PM_SPOOLER_PRINTER`. (All names starting with `PM_` are reserved names.) The third parameter is also a pointer to a string that is a key name used to obtain specific profile data. This string depends on the application profile data being requested and varies from application to application. If this parameter is `NULL`, as shown for querying printer names, then a list of key names for the application is returned. This list is actually a contiguous series of 0 terminated strings with the last string in the list having a second 0 placed after it. Hence, the key names returned by using `NULL` for this parameter are the printer names known by the system. The fourth parameter for this function is a pointer to a default string returned to the application if no key names are found. As you can see, `NULL` can be used here as well. The last two parameters for the PrfQueryProfileString function are a pointer to a buffer to hold the string returned by this function, and the maximum size string that this function can place in this buffer.

Once all the key names for the `PM_SPOOLER_PRINTER` application name have been returned, the Printers program parses the returned list and writes out each key name (printer name). The application then writes out a subtitle indicating that printer path information is to follow. This path information is found by using the same PrfQueryProfileString function just shown, but this time using the printer names that were just returned for the key name parameter. When this is done, only a single string is returned

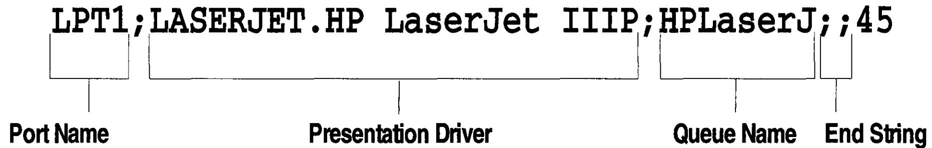


FIGURE 8.4 Parts of the printer path string.

which contains several pieces of information: a port name, one or more presentation driver names, and one or more queue names. These three types of data are separated by semicolons in the returned string. Two consecutive semicolons signal the end of these three types of data and the start of parameter data. (Parameter data is not important to this discussion and is not discussed in this book). Within these sections, items such as presentation drivers or queue names are separated by commas. If more than one presentation driver or queue name is given, the first one given is considered the default. Finally, the presentation driver may include a device name. If present, this device name is located after a dot in the presentation driver name. Figure 8.4 shows an example of a printer path string and what the different parts of this string represent.

The sections of the Printers program that deal with the queue information are almost identical to that shown for printer names, but the application name parameter for the PrfQueryProfileString function changes to a name that applies to queues. To get the queue names for a system, an application name of `PM_SPOOLER_QUEUE` is used with the key name parameter being `NULL`. To get the default presentation driver for each queue name, use the queue names just found as key names with an application name of `PM_SPOOLER_QUEUE_DD`. The presentation driver returned by this function may also have a device name attached to it (just like those queried via printer names). If this is the case, the presentation driver name and device name are separated with a dot.

Finally, the Printers program closes the file just created and passes the filename and window handle to a function called OutWindow. This function is part of the Browse utility DLL that will create a separate thread of execution and have all the logic to create a browse window and display the file's content. When the Browse utility DLL is complete, it will send a message

called WM_ENDBROWSE back to this window procedure to signal its end. This window procedure will then post a message to itself to quit.

You may also want to use the PrfQueryProfileString function to determine which queue processors are available. As said before, the queue processor is the component that takes the print data stream off the queue and passes it to the presentation driver for print. In the process of doing this, the queue processor can also affect the way output is printed. For instance, the queue processor can affect items like number of copies, scaling, and clipping. To query for queue processor names, use the application name PM_SPOOLER_QP with the PrfQueryProfileString function.

```
/************************************************************************/
/* PMPRINT Program.                                                     */
/* Copyright (c) 1994 John Wiley & Sons, Inc.  All rights reserved.     */
/* Reproduction or translation of this work beyond that permitted in    */
/* Section 117 of the 1976 United States Copyright Act without the      */
/* express written permisson of the copyright owner is unlawful.        */
/* Request for further information should be addressed to the          */
/* Permission Department, John Wiley & Sons, Inc.  The purchaser may    */
/* make back-up copies for his/her own use only and not for distribution */
/* or resale.  The Publisher assumes no responsibility for error,       */
/* omissions, or damages, caused by the use of these programs of from   */
/* the use on the information contained herein.                         */
/************************************************************************/
   #define INCL_DOSDATETIME
   #define INCL_DOSFILEMGR
   #define INCL_DOS
   #define INCL_DOSPROCESS
   #define INCL_WIN
   #define INCL_GPI
   #define INCL_DOSDEVICES
   #define INCL_PM
   #define OPEN_FILE 0x01
   #define CREATE_FILE 0X10
   #define FILE_ARCHIVE 0x20
   #define DASD_FLAG 0
   #define INHERIT 0x80
   #define WRITE_THRU 0
   #define FAIL_FLAG 0
   #define SHARE_FLAG 0x10
   #define ACCESS_FLAG 0x02
   #define FILE_SIZE 0L
   #define EABUF 0
   #include <os2.h>
   #include <stdio.h>
   #include <math.h>
   #include <stdlib.h>
   #include <string.h>
   #include <pmerr.h>
```

LISTING 8.2 Printer program.

```
   #include "browse.h"
   /*********************/
   /* Global Variables. */
   /*********************/
   HAB hab;              /* PM print anchor block handle. */
   HMQ hmq;              /* PM print message queue handle. */
   QMSG qmsg;            /* PM print message. */
   HWND hwndWindow;      /* Window handle. */
   /********************************************/
   /* Window Procedure Entry Point Definitions. */
   /********************************************/
   FNWP ClientWndProc;
   FNWP ProductDlgProc;
   /*******************/
   /* Main Procedure. */
   /*******************/
   main()
   {
   hab=WinInitialize(0);
   hmq=WinCreateMsgQueue(hab, 0); WinRegisterClass(hab, "PRINTERS",
      ClientWndProc,CS_SIZEREDRAW, 0);hwndWindow=WinCreateWindow(
      HWND_DESKTOP, "PRINTERS", "", WS_VISIBLE, 0, 0, 0, 0,
      NULLHANDLE, HWND_TOP, 10, NULL, NULL);
   /*************************************/
   /* Get PM messages and dispatch them. */
   /*************************************/
   while(WinGetMsg(hab, &qmsg, 0, 0, 0)){
      WinDispatchMsg(hab, &qmsg);
      }
   /***************************/
   /* Free resources obtained. */
   /***************************/
   WinDestroyWindow(hwndWindow);
   WinDestroyMsgQueue(hmq);
   WinTerminate(hab);
   return 0;
   }
/***************************/
/* Client window procedure. */
/***************************/
MRESULT EXPENTRY ClientWndProc(HWND hwnd, ULONG msg,
   MPARAM mp1, MPARAM mp2)
   {
DATETIME dateTime;     /* Time and date data structure. */
CHAR infoString[200]; /* Buffer for data about printer names and queues.*/
CHAR nameString[200]; /* Buffer for printer names and queues. */
INT aIndex=0;          /* Index into arrays. */
CHAR fileName[]="WSYSPRT.LST"; /* File name to hold printer data. */
CHAR pszFullFile[CCHMAXPATH] = "*.LST";
CHAR pszTitle[]= "System Printer Information";
HFILE FileHandle;      /* Handle of file to write print data to. */
ULONG wrote, action;   /* Variables used to open output file. */
CHAR fileData[200];    /* Buffer to format output data in. */
PCHAR pChar;
```

LISTING 8.2 (Continued).

```
PSZ workString=NULL;  /* Work string pointer.    */
PSZ port=NULL;        /* Pointer to port name.   */
PSZ driver=NULL;      /* Pointer to driver name. */
PSZ device=NULL;      /* Pointer to device name. */
PSZ queue=NULL;       /* Pointer to queue name.  */
switch(msg){
/*********************************/
/* Process window create message. */
/*********************************/
case WM_CREATE:
   memset(&nameString,0,sizeof(nameString));
   memset(&infoString,0,sizeof(infoString));
   /****************************************/
   /* Open file that was typed or selected. */
   /****************************************/
   action=2; DosOpen(fileName,
      &FileHandle, &action, FILE_SIZE, FILE_ARCHIVE,
      OPEN_FILE | CREATE_FILE,
      DASD_FLAG | INHERIT | WRITE_THRU |
      FAIL_FLAG | SHARE_FLAG | ACCESS_FLAG,
      EABUF);DosSetFileSize(FileHandle, FILE_SIZE);
   aIndex=0;
/**********************************/
/* Put out title for this listing. */
/**********************************/
sprintf(fileData, "SYSTEM PRINTER INFORMATION \n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/**********************************************/
/* Format and put out date and time information.*/
/**********************************************/
DosGetDateTime(&dateTime);
sprintf(fileData,
   "Date and Time %d/%d/%d  %d:%d:%d.%d \n\n",
   dateTime.month,
   dateTime.day,
   dateTime.year,
   dateTime.hours,
   dateTime.minutes,
   dateTime.seconds,
   dateTime.hundredths
   );
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/*****************************************/
/* Put out title for printer name section. */
/*****************************************/
sprintf(fileData, "Printer Names: \n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/**********************/
/* Get printer names . */
/**********************/
```

LISTING 8.2 (Continued).

```
PrfQueryProfileString(HINI_PROFILE,
   (PSZ)"PM_SPOOLER_PRINTER", NULL, NULL,
   (PSZ)nameString, (LONG) sizeof(nameString));
/************************************/
/* Put out each printer name found. */
/************************************/
aIndex=0;
while(nameString[aIndex]!=0){
   sprintf(fileData,
   "   ");
   strcat(fileData, &nameString[aIndex]);
   strcat(fileData, "\n");
   DosWrite(FileHandle, (PVOID) fileData,
      strlen(fileData), &wrote);
   while(nameString[aIndex]!=0)aIndex++;
   aIndex++;
   }
/*******************************************/
/* Put out title for printer path section. */
/*******************************************/
sprintf(fileData, "\nPrinter Path Information: \n");
DosWrite(FileHandle, (PVOID) fileData,strlen(fileData), &wrote);
/*************************************************************/
/* Get and put out path information for each printer name found. */
/*************************************************************/
aIndex=0;
while(nameString[aIndex]!=0){
   PrfQueryProfileString(HINI_PROFILE,
   (PSZ)"PM_SPOOLER_PRINTER",
   (PSZ)&nameString[aIndex],
   NULL,
   (PSZ)infoString, (LONG) sizeof(infoString));
/**************************************/
/* Put out white space for this line. */
/**************************************/
sprintf(fileData, "   ");
/**********************************************/
/* Put out printer name for this path data. */
/**********************************************/
strcat(fileData, &nameString[aIndex]);
strcat(fileData, " -->  ");
/*********************/
/* Put out path data. */
/*********************/
strcat(fileData, infoString);
strcat(fileData, "\n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/*************************************************/
/* Parse the returned string into basic parts. */
/*************************************************/
workString=&infoString[0];
pChar=strchr(workString, ';'); /* Find first semicolon.              */
```

LISTING 8.2 (Continued).

```
port=workString;         /* Point port string.                          */
*pChar='\0';             /* Terminate string  by replacing ; with 0. */
workString=++pChar;      /* Point to start of next section.             */
pChar=strchr(workString, ';'); /* Find next semicolon.                 */
driver=workString;       /* Point presentation   driver string.         */
*pChar='\0';             /* Terminate string  by replacing ; with 0. */
workString=++pChar;      /* Point to start of next section.             */
pChar=strchr(workString, ';'); /* Find next semicolon.                 */
queue=workString;        /* Point to queue string.                      */
*pChar='\0';             /* Terminate string by replacing ; with 0.  */
/***************************************/
/* Put out white space for this line. */
/***************************************/
sprintf(fileData, "     Where:\n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/**********************/
/* Put out port data. */
/**********************/
sprintf(fileData, "        Port name -->  ");
strcat(fileData, port);
strcat(fileData, "\n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/*************************************/
/* Put out driver.device name data. */
/*************************************/
sprintf(fileData, "        Driver names.Device names -->  ");
strcat(fileData, driver);
strcat(fileData, "\n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/***********************/
/* Put out queue data. */
/***********************/
sprintf(fileData, "        Queue names -->  ");
strcat(fileData, queue);
strcat(fileData, "\n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
while(nameString[aIndex]!=0)aIndex++;
aIndex++;
}
/*******************************************/
/* Put out title for queue name section. */
/*******************************************/
sprintf(fileData, "\nQueue Names: \n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/*********************/
/* Get queue names . */
/*********************/
```

LISTING 8.2 (Continued).

```
PrfQueryProfileString(HINI_PROFILE,
   (PSZ)"PM_SPOOLER_QUEUE", NULL, NULL,
   (PSZ)nameString, (LONG) sizeof(nameString));
/**********************************/
/* Put out each queue name found. */
/**********************************/
aIndex=0;
while(nameString[aIndex]!=0){
   sprintf(fileData, "   ");
   strcat(fileData, &nameString[aIndex]);
   strcat(fileData, "\n");
   DosWrite(FileHandle, (PVOID) fileData,
      strlen(fileData), &wrote);
   while(nameString[aIndex]!=0)aIndex++;
   aIndex++;
}
/*****************************************/
/* Put out title for queue data section. */
/*****************************************/
sprintf(fileData, "\nQueue Default Presentation Driver: \n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
/******************************************************************/
/* Get and put out the default presentation driver for each queue. */
/******************************************************************/
aIndex=0;
while(nameString[aIndex]!=0){
   PrfQueryProfileString(HINI_PROFILE,
      (PSZ)"PM_SPOOLER_QUEUE_DD",
      (PSZ)&nameString[aIndex], NULL,
      (PSZ)infoString, (LONG) sizeof(infoString));
/**************************************/
/* Put out white space for this line. */
/**************************************/
sprintf(fileData, "   ");
/********************************************************/
/* Put out queue name for this default presentation driver. */
/********************************************************/
strcat(fileData, &nameString[aIndex]);
strcat(fileData, " -->  ");
/***************************************/
/* Put out default presentation driver. */
/***************************************/
strcat(fileData, infoString);
strcat(fileData, "\n");
DosWrite(FileHandle, (PVOID) fileData,
   strlen(fileData), &wrote);
while(nameString[aIndex]!=0)aIndex++;
aIndex++;
}
DosClose(FileHandle);
OutWindow(hwnd,fileName);
return(MRESULT)FALSE;
```

LISTING 8.2 (Continued).

```
/*******************************/
/* Process END Browse message. */
/*******************************/
case WM_ENDBROWSE:
   WinPostMsg(hwnd, WM_QUIT, NULL, NULL);
   return(MRESULT)FALSE;
}
/***************************************************/
/* Let default routine process message and return. */
/***************************************************/
return WinDefWindowProc(hwnd, msg, mp1, mp2);
}
```

LISTING 8.2 (Continued).

Now that you've seen how to gather printer information from the OS/2 system, we can show you how to use this information to create a printer device context and perform printing for our graphics editor application. Listing 8.3 shows the logic used in the graphics editor program to perform printing. Because all the print logic for the graphics editor application is contained in the Print pull-down menu item and a small printer name selection dialog, only those two pieces of program logic are shown. Note that the Print pull-down menu item logic is found in `DRAW.C` and the printer name selection dialog logic is found in `FUNCS.C`.

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,
   MPARAM mp1,MPARAM mp2)
   {
   switch(msg)
{
/**********************************/
/* Process pull-down menu items. */
/**********************************/
case WM_COMMAND:
   switch (COMMANDMSG(&msg)->cmd){
      /************************************/
      /* Process Print pull-down option. */
      /************************************/
      case IDM_PRINT:
         if(WinDlgBox(HWND_DESKTOP,hwnd,PrintDlgProc,
            0,IDD_PRNT,&printer)){
         memset(printerPath,0,sizeof(printerPath));
         /**************************************/
         /* Get printer data for printer name. */
         /**************************************/
         PrfQueryProfileString(HINI_PROFILE,
            (PSZ)"PM_SPOOLER_PRINTER",(PSZ)printer,
            NULL,(PSZ)printerPath,(LONG)sizeof(printerPath));
```

LISTING 8.3 Draw program print routine.

```
   /**********************************************/
   /* Parse the returned string into basic parts. */
   /**********************************************/
   stringPointer=&printerPath[0];
   charPointer=strchr(stringPointer, ';'); // Find first semicolon.
   portPointer=stringPointer;    // Point port string.
   *charPointer='\0';         // Terminate string by replacing ;
   stringPointer=++charPointer;  // Point to start of next section.
   charPointer=strchr(stringPointer, ';'); // Find next semicolon.
   driverPointer=stringPointer;  // Point presentation driver string.
   *charPointer='\0';            // Terminate string by replacing ;
   stringPointer=++charPointer;  // Point to start of next section.
   charPointer=strchr(stringPointer, ';'); // Find next semicolon.
   queuePointer=stringPointer;   // Point to queue string.
   *charPointer='\0';            // Terminate string by replacing ;
   /**************************/
   /* Get default queue name. */
   /**************************/
   stringPointer=queuePointer;   // Point to start of queue section.
   charPointer=strchr(stringPointer, ','); // Use default queue name.
   if(charPointer!=NULL) {*charPointer='\0'; // Terminate string.
   }
/**********************************/
/* Get default driver.device name. */
/**********************************/
stringPointer=driverPointer; // Point to start of driver section.
charPointer=strchr(stringPointer,','); // Use default driver.
if(charPointer!=NULL){*charPointer='\0'; // Terminate string.
}
/*********************************************/
/* Break driver.device name into two strings. */
/*********************************************/
stringPointer=driverPointer; // Point to start of driver section.
charPointer=strchr(stringPointer,'.'); // Find . and separate.
if(charPointer!=NULL){*charPointer='\0'; // Terminate string.
devicePointer=++charPointer;
}
/***********************************************/
/* Get length of memory needed for driver data. */
/***********************************************/
length=DevPostDeviceModes(hab,NULL,
   (PSZ)driverPointer,(PSZ)devicePointer,
   (PSZ)printer,DPDM_POSTJOBPROP);
/*******************************************************/
/* Allocate memory for driver data if everything is okay. */
/*******************************************************/
if(length!=DPDM_ERROR && length!=DPDM_NONE){
driveData=malloc(length);
/*******************************************************/
/* Initialize length field to 0 so we can test for change */
/* on return from DevPostDeviceModes function.         */
/*******************************************************/
```

LISTING 8.3 (Continued).

```
        ((PDRIVDATA)driveData)->cb=0;
        length=DevPostDeviceModes(hab,driveData,
           (PSZ)driverPointer,(PSZ)devicePointer,
           (PSZ)printer,DPDM_POSTJOBPROP);
        /*************************************************************/
        /* If the driver data has been updated then prepare for print, */
        /* else an operator cancel or other event has caused the print */
        /* setup to abort.                                             */
        /*************************************************************/
        if(!(((PDRIVDATA)driveData)->cb==0)){
           prtdopData.pszLogAddress=queuePointer;
           prtdopData.pszDriverName=driverPointer;
           prtdopData.pdriv=driveData;
           prtdopData.pszDataType="PM_Q_STD";
           prtdopData.pszComment="Wiley Graphics Editor Print";
           prtdopData.pszQueueProcName=NULL;
           prtdopData.pszQueueProcParams="COL=C XFM=0";
           prtdopData.pszSpoolerParams=NULL;
           prtdopData.pszNetworkParams=NULL;
           /*******************************************************/
           /* Get a device context, associate it, and print the PS. */
           /*******************************************************/
           phdc=DevOpenDC(hab,OD_QUEUED,"*",9L,
              (PDEVOPENDATA)&prtdopData,0L);
           if(!phdc==DEV_ERROR){
              GpiAssociate(hps,0L);  // Disassociate PS and window DC.
              GpiAssociate(hps,phdc); SetDefaultView(hps,1,ptlDefault);
              DevEscape(phdc,DEVESC_STARTDOC,sizeof(docName),
                 docName,&outCount,NULL);
              GpiSetDrawingMode(hpsPrint,DM_DRAW);
              GpiDrawChain(hps);
              outCount=2;
              DevEscape(phdc,DEVESC_ENDDOC,0L,
                 NULL,&outCount,(PBYTE)&jobID);
              SetDefaultView(hps,zoomFactor,ptlTranslate);
              GpiAssociate(hps,0L); // Disassociate the PS and print DC.
              GpiAssociate(hps,hdc);  // Connect back to window.
              DevCloseDC(phdc);     // Release the print DC.
              }
           }
           free(driveData);    // Free print driver data structure.
           }
        }
        break;
     }
  }
  /*************************************************/
  /* Let default routine process message and return. */
  /*************************************************/
  return WinDefWindowProc(hwnd,msg,mp1,mp2);
  }
  /**************************/
  /* Print dialog procedure. */
  /**************************/
```

LISTING 8.3 (Continued).

```
MRESULT EXPENTRY PrintDlgProc(HWND hwnd,ULONG msg,
   MPARAM mp1,MPARAM mp2)
   {
   static BYTE FAR *ptrPrinter;
   CHAR printerName[150];
   SHORT index;
   USHORT item;
   MRESULT mReturn;
   USHORT keyFlags;
   switch(msg){
      /*********************/
      /* Initialize dialog. */
      /*********************/
      case WM_INITDLG:
         ptrPrinter=PVOIDFROMMP(mp2);
         PrfQueryProfileString(HINI_PROFILE,
            (PSZ)"PM_SPOOLER_PRINTER",NULL,NULL,
            (PSZ)printerName,(LONG)sizeof(printerName));
         index=0;
         /****************************************/
         /* If no printer names then abort dialog. */
         /****************************************/
         if(printerName[index]==0)
            WinDismissDlg(hwnd,FALSE);

         /*************************************************/
         /* Else put names in list and select the first one. */
         /*************************************************/
         else{
            while(printerName[index]!=0){
               WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_INSERTITEM,
                  MPFROM2SHORT(LIT_SORTASCENDING,0),
                  (MPARAM)(&printerName[index]));
               while(printerName[index]!=0)index++;
               index++;
               }
               WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_SELECTITEM,
                  MPFROMSHORT(0),(MPARAM)TRUE);
               WinSetFocus(HWND_DESKTOP,WinWindowFromID(hwnd,IDD_PRTLIST));
               }
            return(MRESULT)1L;
         case WM_CONTROL:
            switch(SHORT2FROMMP(mp1)){
            /************************************/
            /* Check for list box notifications. */
            /************************************/
            case LN_ENTER:
               mReturn=WinSendDlgItemMsg(hwnd,IDD_PRTLIST,
                  LM_QUERYSELECTION,MPFROMSHORT(LIT_FIRST),
                  (MPARAM)NULL);
               item=(SHORT)mReturn;
               WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_QUERYITEMTEXT,
```

LISTING 8.3 (Continued).

```
                MPFROM2SHORT(item,25),(MPARAM)ptrPrinter);
            WinDismissDlg(hwnd,TRUE);
            return(MRESULT)TRUE;
         case LN_SELECT:
            mReturn=WinSendDlgItemMsg(hwnd,IDD_PRTLIST,
               LM_QUERYSELECTION, MPFROMSHORT(LIT_FIRST),
               (MPARAM)NULL);item=(SHORT)mReturn;
            WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_QUERYITEMTEXT,
               MPFROM2SHORT(item,25),(MPARAM)ptrPrinter);
            return(MRESULT)TRUE;
         }
      break;
   /********************************/
   /* Process key stroke message. */
   /********************************/
   case WM_CHAR:
      keyFlags=(USHORT)SHORT1FROMMP(mp1);
      if(keyFlags & KC_VIRTUALKEY){
         switch (SHORT2FROMMP(mp2)){
   case VK_ENTER:     // Check for enter key.
   case VK_NEWLINE:   // Check for newline key.
   mReturn=WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_QUERYSELECTION,
      MPFROMSHORT(LIT_FIRST),(MPARAM)NULL);
   item=(SHORT)mReturn;
   WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_QUERYITEMTEXT,
      MPFROM2SHORT(item,25),(MPARAM)ptrPrinter);
   WinDismissDlg(hwnd,TRUE);
   return(MRESULT)TRUE;
   }
}
break;
/**************************/
/* Process notifications. */
/**************************/
case WM_COMMAND:
   switch(COMMANDMSG(&msg)->cmd){
      case DID_OK:
         mReturn=WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_QUERYSELECTION,
            MPFROMSHORT(LIT_FIRST),(MPARAM)NULL);
         item=(SHORT)mReturn;
         WinSendDlgItemMsg(hwnd,IDD_PRTLIST,LM_QUERYITEMTEXT,
            MPFROM2SHORT(item,25),(MPARAM)ptrPrinter);
         WinDismissDlg(hwnd,TRUE);
         return(MRESULT)TRUE;
      case DID_CANCEL:
         WinDismissDlg(hwnd,FALSE);
         return(MRESULT)TRUE;
      }
      break;
   }
return WinDefDlgProc(hwnd, msg, mp1, mp2);
}
```

LISTING 8.3 (Continued).

As you can see in Listing 8.3, the first thing the Print pull-down menu item does is start a dialog procedure called PrintDlgProc (located at the end of Listing 8.3). This dialog procedure is designed to allow the user to select one of the printer names that is known to the OS/2 system and return it to the Print pull-down menu item routine. By inspecting this PrintDlgProc procedure, you can see that it does this by first building a list box and then filling it in with all the known printer names. It builds this list when it receives a WM_INTDLG message. The printer names for the list box are gathered the very same way they were found by the Printer program. That is, the PrfQueryProfileString function is used with a handle of HINI_PROFILE (so both system and user profiles are searched), an application name of PM_SPOOLER_PRINTER (so the OS/2 Spooler will return printer information), and a key name of NULL (so all printer names will be returned). After this function is called, the dialog procedure tests to see if any printer names were returned by the function. If no printer names were found, then the dialog is dismissed with a FALSE indicator so the Print pull-down menu item will not continue the print process. If there are printer names, however, these names are parsed from the buffer that was filled in by the PrfQueryProfileString function and inserted into the list box in ascending order. The other messages that are processed in the PrintDlgProc procedures manage the list selection logic and button processing. When the OK button is clicked, the first highlighted list item is located, and then placed in the location requested by the Print pull-down menu item. The dialog is then dismissed with a TRUE indicator. If the user clicks the cancel button, then the dialog is dismissed with a FALSE indicator so the Print pull-down menu item will not continue the print process.

If the Print pull-down menu item receives a TRUE indicator from the PrintDlgProc, it assumes printing will occur and disconnects the graphics editor presentation space from the window device context. Next it clears out a buffer area to store printer information to be stored and then queries for the printer information with the printer name just returned from the PrintDlgProc procedure. As you recall from the Printers program, when a printer name is used as the key name with the PrfQueryProfileString function, a long string with embedded semicolons and periods is returned. This string can then be parsed to find several pieces of information for the printer name. The next section of logic in the Print pull-down menu item parses this long returned string into its simplest parts so they can be used to create a device

context for the printer. In particular, this string is parsed to obtain the default queue name, default presentation driver name, and default device name. Once these names have been parsed, a function called *DevPostDeviceModes* is used to obtain driver data.

The DevPostDeviceModes function has several parameters. The first is just a valid anchor-block handle. The second parameter is a pointer to driver data. If this second parameter is `NULL`, this function will return the length needed by the presentation driver to generate driver data. In fact, our first call of the DevPostDeviceModes function uses `NULL` for the second parameter so we can allocate the correct amount of space for driver data. The third parameter is the presentation driver name with which we want to interface. The fourth parameter is a pointer to a string that names the type of device to which we will be sending our graphics data. The fifth parameter is a pointer to a string that has the name of the device we want our graphics data to. And finally, the sixth parameter is a value for an option. The option parameter has defined values of `DPDM_POSTJOBPROP` and `DPDM_QUERYJOBPROP`. If `DPDM_POSTJOBPROP` is specified as an option and the pointer to the driver data is not `NULL`, the presentation driver will display a dialog for the user to work with. This dialog allows the user to select printer options specific to the device to which the print data will be routed. (Pretty cool stuff!) And on return from the dialog displayed by the presentation manager, the correct driver data is filled in. If `DPDM_QUERYJOBPROP` is used as an option and the pointer to the device data is not `NULL`, then the presentation driver will fill the driver data structure with established default values and no dialog will be displayed. As you can see in Listing 8.3, our graphics editor application uses the `DPDM_POSTJOBPROP` option so another dialog will appear.

Once driver data has been established, the Print pull-down menu item logic fills in a data structure required by a function called *DevOpenDC*. You have probably already seen and used the DevOpenDC function, but if you've only used it to obtain a memory device context, most of the parameters associated with the function were fairly boring (0). To create a printer device context, however, the parameters for the DevOpenDC function become much more interesting. Like the DevPostDeviceModes function , the first parameter for the DevOpenDC function is a valid anchor-block handle. The second parameter is a value that indicates the type of device context to be opened. Several definitions exist for values that can be assigned to this

parameter. These definitions are OD_QUEUED, OD_DIRECT, OD_INFO, OD_METAFILE, OD_METAFILE_NOQUERY, and OD_MEMORY. For our device context, we want our output to be queued, therefore the value we use for this parameter is OD_QUEUED. The third parameter is a pointer to a string that identifies the device information held in the initialization file. When this string is "*", like in our graphics editor application, no information is taken from the initialization file. Instead, all data is obtained from the device context data area. The fourth parameter for the DevOpenDC function is a count of the number of items that are present in the open device context data area. Parameter five is a pointer to the open device context data area. The last parameter of the DevOpenDC function is a handle to a device context used when the type of device context being opened is OD_MEMORY. This handle is to a device context with bitmaps that are to be used with this device context. In our case, this parameter is zero.

The fifth parameter to the DevOpenDC function, a pointer to an open device context data area, is where the print job information that we've been accumulating is used. As you can see from the prtdopData structure used in our example, this structure is actually an array of pointers. As implied by the fourth parameter of the DevOpenDC function, the entire array of pointers is not required to satisfy the DevOpenDC function. Our graphics editor application does, however, fill in the entire open device context data area structure.

The first field of the open device context data area structure, pszLogAddress, is used to specify the logical address of the device to which data is to be communicated. In the case of a device context type of OD_QUEUED, this is a queue name such as "LPT1Q". If the device context type is OD_DIRECT, the logical address would refer to a port name such as LPT1. In the graphics editor application, the logical address is the name of the default queue found with the PrfQueryProfileString function.

The second field of the open device context data area structure, pszDriverName, is used to specify the presentation driver name with which the presentation space wants to communicate. Again, this driver name for our example application is the default presentation driver name found for the printer name by using the PrfQueryProfileString function.

The third field of the open device context structure is used to specify presentation driver data. This data is used by the presentation driver to help set up the print job for a particular printer. This presentation driver data can

be established in a few different ways. For instance, for a driver name that does not have a dot qualified device name, you can use NULL for this field. If the driver name does have a dot qualified device name, then a pointer to presentation driver data is required. If you want to specify default printer and job properties for your output, you can build a DRIVDATA structure and initialize the following fields in the following way (note that *X* and *devicename* are variable):

```
X.cb=sizeof(DRIVDATA);
X.lVersion=0;
strcpy(X.szDeviceName, devicename);
X.abGeneralData[0]=0;
```

The graphics editor application builds and passes driver data to the presentation driver by using the DevPostDeviceModes function with an option of DPDM_POSTJOBPROP. When this is done, the presentation driver presents the user with a dialog with which to interact. By doing this, the presentation driver builds the driver data that fulfills the needs of the user and returns it to our application. Then, when our application uses the DevOpenDC function, it points to this driver data as an input parameter to the function.

The fourth field of the open device context data area structure, pszDataType, is used to specify the type of print file data that is queued when a device context type of OD_QUEUED is being used. If the device context type is not OD_QUEUED, then NULL can be used for this field. Normal strings pointed to by this field are "PM_Q_STD" or "PM_Q_RAW". "PM_Q_STD" is standard print spool file format which is designed to promote device independence and is the recommended format. The "PM_Q_RAW" file format is device specific; hence, output generated with this format will probably not print on any other printer than the one for which the output was created. Other strings may exist for other queue processors.

The fifth field of the open device context data area structure, pszComment, is used to attach some descriptive text to the print job. The Print Manager will display this text along with other job information when the job is in the queue.

The sixth field of the open device context data area structure, pszQueueProcName, is used to specify the queue processor to be used to remove the print file from the queue and route the data back to the presentation driver. By specifying NULL, like our application does, the default

queue processor is used. The default queue processor is most likely named "PMPRINT" or is compatible with this queue processor.

For some jobs, such as those going to a plotter, special requirements or restrictions in how output is constructed may be needed. Because of the location of the queue processor in the system, it has the ability to alter or control the flow of data back to the presentation driver. This is where a special queue processor may be used to solve some of these special requirements. In these cases, a special queue processor that has been installed on the system may be specified.

Because queue processors can alter and control the flow of data back to the presentation driver, it is also common to be able to pass parameters to the queue processor so you can control some of their processing. Because different queue processors can have different options, the parameters that can be passed to them may also be different. Field seven of the open device context data area structure, pszQueueProcParams, is the means by which your application can pass parameters to the queue processor. This field points to a string that contains the parameters specific to the queue processor you will be using. In our example, which defaults to the PMPRINT queue processor, "COL=C XFM=0" is specified. As you might guess from this string, the format of passing parameters to a queue processor is by using a keyword followed by a value. In the case of our graphics editor, COL=C specifies that color or shades of grey should be used as opposed to monochrome (COL=M). XFM=0 specifies that the drawing should be printed real size. If XFM=1 were used, then two more keyword parameters called FIT= and ARE= could be used to alter the outputs position and size. To get more detail about all the parameters available for the PMPRINT queue processor, refer to IBM technical reference material. But, one last parameter worth mentioning here (even though our graphics editor doesn't use it) is the COP= parameter. As you may have guessed, the COP= parameter allows your program to specify the number of copies you'd like the queue processor to print.

Field eight of the open device context data area structure, pszSpoolerParams, is similar to field seven. This field, however, allows you to specify parameters to the OS/2 Spooler. These parameters are also passed in a string where there are keywords followed by a value. Again, you should refer to IBM technical reference material to get a complete list of spooler parameters and descriptions, but the two worth noting here are FORM= and PRTY=.

These two parameters allow you to specify a valid form name as returned by the DevQueryHardcopyCaps function and set the priority of the print file in relationship to others in the queue. (Priority values range from 00–99 where 99 is the highest priority.) Our example application doesn't send any parameters to the spooler, hence, we get default values. In the case of forms, the current form is used and the default priority is a value of 50.

The last field of the open device context data area structure, pszNetworkParams, is used to specify network parameters. The graphics editor application passes a NULL for this field; hence, no parameters are passed.

After the graphics editor application creates a printer device context, it associates the presentation space used to produce the drawing with the printer device context. Next, the function called SetDefaultView (which is found in `FUNC.C`) is used to ensure the drawing is not zoomed and the picture origin is at the correct location. Once the zoom and translate transform for the drawing are known to be correct, a start document escape sequence is sent to the printer presentation driver with the *DevEscape* function.

The DevEscape function allows the application to send a special code, called an escape code, to the presentation driver to communicate job information. A standard set of escape codes are available and our application uses two of them. These two standard escape codes are defined as `DEVESC_STARTDOC` and `DEVESC_ENDDOC` and are used to signal the start and end of our print job to the presentation driver. The parameters for the DevEscape function are fairly straightforward. The first parameter is a handle to the device context. This allows the function to find the correct presentation driver to which to route the escape code. The second parameter of the DevEscape function is the escape code value. Parameter three is a count value in bytes of the amount of data being passed with the escape code, and parameter four is a pointer to the data. In the case of sending a start document escape code, this is the length of the name and the name our application is assigning to the document. If you look at job information for the printer to which this output is being routed, you will see the document name listed or under the print job icon. The fifth parameter is a pointer to a value that indicates the number of bytes of data associated with the sixth parameter, which is a pointer to a data buffer. The data associated with parameters five and six may be for input to or output from the presentation driver. In the case of sending a end of document escape code, the presentation driver can return a USHORT value that is a print job identifier. Hence, parameter five

can be pointing to a value of two and parameter six can be a pointer to a USHORT variable. (If parameter five were pointing to a value of zero, then no job identifier would be returned.)

After the start document escape has been set to the presentation driver, our application generates the drawing by issuing all the GPI functions required to create the drawing. Because our graphics editor uses retained graphics, all we need to do is draw the chain with the GpiDrawChain function and our drawing is complete. TOO EASY! Now all our application has to do is send another device escape code to tell the presentation driver that we are done printing the document. This is done by using the `DEVESC_ENDDOC` escape code. From the system point of view, our application is done printing the document. Getting the output onto paper is now the responsibility of OS/2!

When the application is done printing, all it needs to do is release the resources it acquired for the printing task, reapply the zoom and scroll transforms for window device context, and reassociate our drawing presentation space with the window device context. As you can see from Listing 8.3, all of these steps are fairly straightforward.

CHAPTER 9

OS/2 MetaFiles

By now you should have a fairly good idea of how to use the GPI to create your own graphical product. If you want to allow picture interchange between your product and other commerially available products, however, you need to be able to read and write the other products' graphical output format. For instance, our WLY file format is our own creation. Because of this, the only product capable of reading our WLY files is our own. Of course, if some product developer was just dying to read or write our file format, they could try to reverse engineer the data stream or perhaps ask us to provide a specification (or in our case, buy our book!). Reverse engineering a data stream, however, does not always reveal all the fine points needed to use a data stream correctly; even though receiving a specification from another product group is much safer, it typically does not limit the originator of the data stream from enhancing it. Hence, your product can become obsolete in the area of importing other product data streams as they enhance their data stream.

So, what can be done? Well, expecting other products not to enhance their data streams is not a very safe bet. Therefore, common ground or a more standard output file format is desirable. This does not mean to imply that a standard file format won't be enhanced. But typically a standard file format is enhanced in a way that makes it easier with which to stay compatible. This is the point of the OS/2 MetaFile. As you will soon see, OS/2 has

functions that make it easy to produce this standard file format, as well as to read it.

Our graphic editor program actually exports two types of standard graphic information. One standard file format is called Tag Image File Format (TIFF). (We use specification revision 5.0.) TIFF is used to store high-quality image data (pixel information) to a file. By doing this, other products can reproduce these high-quality images. It was the export to TIFF feature of our graphic editor that was used to produce all of our screen and window captures for this book. As you can tell, given a utility program to read this TIFF data and then write the image data to a high-quality printer can produce excellent results.

The second standard file format our editor exports to is the OS/2 MetaFile. An OS/2 MetaFile is designed to be device independent. It can contain image data similar to TIFF but it can also save drawing commands. Because of this, a MetaFile may be much smaller in size than a TIFF file which captures only image data. Also, because the MetaFile does record graphic order information, manipulating a MetaFile graphic is much easier to do. The only manipulation of a MetaFile you will see in this book, however, is how we scale a complete MetaFile to fit our MetaFile Viewer window! But, because we do export picture information to the OS/2 MetaFile format, you can create drawings with our graphic editor and import them into other graphic editors to use.

Basically, an OS/2 MetaFile is a recording of all the graphic orders and environmental data for a presentation space. Because of this, you will soon see that the same graphic segment format and graphic orders discussed eariler in this book are contained within the MetaFile. In addition to this graphic segment and order data, the MetaFile also records other environmental information that relates to the presentation space. This information is needed so when the MetaFile is played back into presentation space, the receiving presentation space can be set up in the very same way as the one used during the recording. By doing this, the drawing saved in the MetaFile can be reproduced just like the original.

Before we jump into the OS/2 functions that allow us to create and use OS/2 MetaFiles, a conceptual view of how MetaFiles are created and played back into a presentation space is helpful. Figure 9.1 shows a conceptual view of how MetaFile data can be thought of in the OS/2 enviornment. The MetaFile is actually a data stream that is in system memory. You should

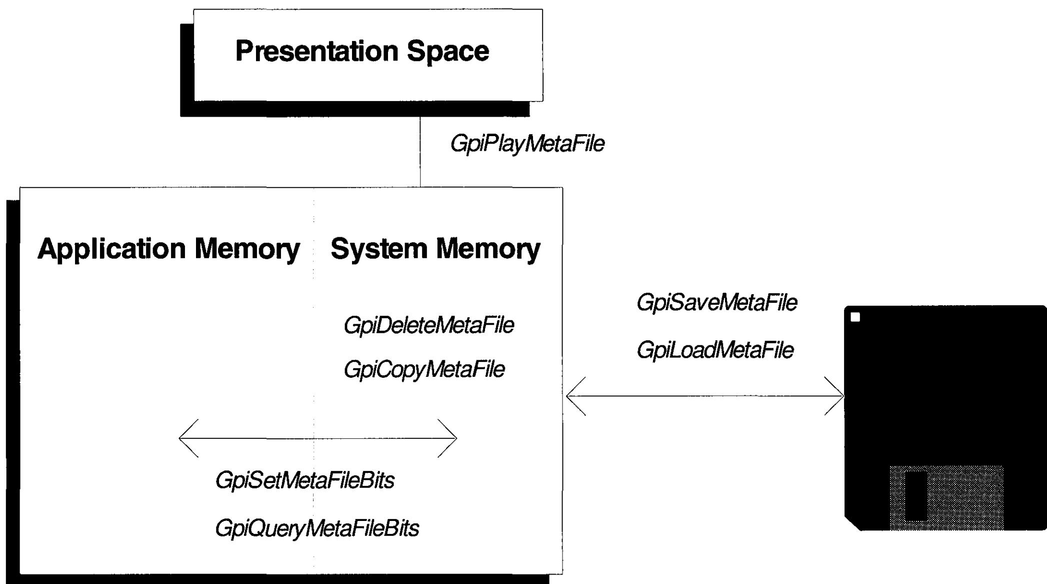


FIGURE 9.1 OS/2 MetaFile conceptual view.

note that we make a destinction between system memory and application memory. (Application memory is allocated and owned by your application.) When the MetaFile exists in system memory, it can be played to a presentation space, copied to application memory, or written to disk. Conversely, to get a MetaFile in system memory, it can be loaded from a file, copied from application memory, or created from a presentation space.

Of course when we think of transporting MetaFiles to other systems, we think of MetaFiles as just another file that is saved on your PC disk drive. (And as just indicated, a MetaFile can take a file form!) OS/2, however, works with a MetaFile just as if it were a device such as a printer or window. Therefore, the first step in creating an OS/2 MetaFile is to create a MetaFile device context. To do this, you need to specify either `OD_MetaFile` or `OD_MetaFile_NOQUERY` in the device type field of the DevOpenDC GPI function. (If you use `OD_MetaFile_NOQUERY`, performance could be better when recording but none of the query attribute GPI functions are supported.) The device data that is associated with the DevOpenDC function is also simple. All you really need to specify for device data is that the driver name is DISPLAY. If you want the MetaFile to have a description, fill in the pszComment field.

Before you actually associate your MetaFile device context with your presentation space, you should insure that the presentation space environmental atrributes and resources are set up the way you want. This is because a MetaFile records a snapshot of these environmental attributes and resources before it records drawing orders and then uses these attributes and resources for the remainder of the recording. (Actually, the OS/2 MetaFile will record many of the attribute and resource changes as escape orders; however, they should still be avoided because other products may fail to recognize these escape orders.) For instance, you should not change your presentation page units or dimensions after you associate the MetaFile device context with your presentation space. You should also establish your color tables before associating the device context. Other restrictions will be listed later, but for now let's finish the logic that creates the MetaFile and puts it to disk.

Once you associate your presentation space with the MetaFile device context, you can start issuing GPI drawing functions. But, as we saw earlier in our book, the destination of the drawing information is still dependent on the drawing mode. (Recall draw mode is set with the GpiSetDrawMode function and can be set to DRAW, RETAIN, or DRAWANDRETAIN.) If the draw mode is DRAW, then all drawing is directed to the MetaFile. If the draw mode is RETAIN, however, the drawing goes to graphic segment store until a GpiDrawChain, GpiDrawFrom, or GpiDrawSegment function is used. Once a GpiDrawChain, GpiDrawFrom, or GpiDrawSegment function is used, the drawing is directed to the MetaFile. If the draw mode is DRAWANDRETAIN, then drawing goes to both graphic segment store and the MetaFile.

When you are done drawing to the MetaFile, you can disassociate the MetaFile device context from your presentation space and close the device context. You may think you are done with your MetaFile recording but you're not! When you close the MetaFile device context, OS/2 will return a handle to a MetaFile. At this point in time, your MetaFile is in memory but not on PC disk. To place the memory MetaFile to disk, you can use a function called *GpiSaveMetaFile*. The GpiSaveMetaFile function will accept a MetaFile handle and a filename for your memory MetaFile and then automatically create the MetaFile on disk for you! Again, too easy. Listing 9.1 shows the logic used by the graphic editor to save OS/2 MetaFiles for this book.

```
MRESULT EXPENTRY ClientWndProc(HWND hwnd,ULONG msg,MPARAM mp1,MPARAM mp2){
   switch(msg){
      /*********************************/
      /* Process pull-down menu items. */
      /*********************************/
      case WM_COMMAND:
      switch (COMMANDMSG(&msg)->cmd){
         /*********************************************/
         /* Process export metafile pull-down option. */
         /*********************************************/
         case IDM_MET:
            memset(&fileDialog,0,sizeof(FILEDLG));
            /*************************************/
            /* Initialize file dialog structure. */
            /*************************************/
            fileDialog.cbSize=sizeof(FILEDLG);
            fileDialog.fl=FDS_HELPBUTTON | FDS_CENTER | FDS_OPEN_DIALOG;
            fileDialog.pszTitle=metOpen;
            strcpy(fileDialog.szFullFile,metaFile);
            /****************************************/
            /* Display the dialog and get the file. */
            /****************************************/
            hwndFileDlg=WinFileDlg(HWND_DESKTOP,hwndFrame,&fileDialog);
            if(fileDialog.lReturn==DID_CANCEL)return(MRESULT)TRUE;
            strcpy(metaFile,fileDialog.szFullFile);
            dopData.pszLogAddress=NULL;
            dopData.pszDriverName="DISPLAY";
            dopData.pdriv=NULL;
            dopData.pszDataType=NULL;
            dopData.pszComment="John Wiley & Sons, Inc. OS/2 GPI Book.";
            dopData.pszQueueProcName=NULL;
            dopData.pszQueueProcParams=NULL;
            dopData.pszSpoolerParams=NULL;
            dopData.pszNetworkParams=NULL;
            GpiAssociate(hps,0L);
            metaDc=DevOpenDC(hab,OD_METAFILE,"*",9L,(PDEVOPENDATA)
               &dopData,0L);
            GpiAssociate(hps,metaDc);
            GpiDrawChain(hps);
            GpiAssociate(hps,0L);
            GpiAssociate(hps,hdc);
            hmf=DevCloseDC(metaDc);
            DosDelete(fileDialog.szFullFile);
            GpiSaveMetaFile(hmf,fileDialog.szFullFile);
            break;
         /*************************************/
         /* Default processing for WM_COMMAND. */
         /*************************************/
         default:
            return WinDefWindowProc(hwnd,msg,mp1,mp2);
         }
      break;
   }
}
```

LISTING 9.1 Draw MetaFile recording routine.

As you can see in Listing 9.1, the graphic editor first sets up a structure required by the WinFileDlg function and then uses the WinFileDlg function to get a filename from the user. If the user cancels out of WinFileDlg function, this menu item just returns; therefore, no MetaFile is created. If the user does provide a valid filename, a device open data structure is initialized for the DevOpenDC function. As you can see, the only fields that are really used in this data structure are the pszDriverName and pszComment fields. When we look at the MetaFile content later, you will notice that the pszComment field is used as part of the MetaFile description. Right before the DevOpenDC function is used to open a MetaFile device context, the graphic editor presentation space is disassociated from its window device context. After the DevOpenDC function, the MetaFile device context is associated with the graphic editor presentation space. Then the GpiDrawChain function is used to produced the drawing.

After the drawing is produced using the MetaFile device context, the window device context is reassociated with the graphic editor presentation space and the MetaFile device context is closed. Then a DosDelete function is done to insure the filename the user gave us does not exist when we do the GpiSaveMetaFile function. After we do the GpiSaveMetaFile function, the MetaFile is saved to disk and the memory MetaFile is automatically deleted. At this point, other applications or our own application can use this MetaFile saved on disk. (Note that if you wanted to delete a memory MetaFile without writing it to disk, you can use a function called *GpiDeleteMetaFile*.)

If you want to make a copy of a MetaFile that exists on disk, you could obviously just copy the file. But if you want to copy MetaFiles or parts of MetaFiles that exist in memory, you must use other GPI functions. Copying memory MetaFile data to and from application memory can be done with functions called *GpiSetMetaFileBits* and *GpiQueryMetaFileBits*. The GpiSetMetaFileBits is used to transfer MetaFile data from an application buffer to a memory MetaFile, whereas, the GpiQueryMetaFileBits can be used to get data from a memory MetaFile to an application buffer. Of course, once you know the internals of a MetaFile, you may uses these functions to do some fairly complex manipulation of MetaFile data, but a simple example of how these functions can be used is to copy an existing memory MetaFile to a new memory MetaFile.

To create a new empty memory MetaFile, just open and then immediately close a MetaFile device context. Once you have an empty memory

MetaFile, you can use the GpiQueryMetaFileBits function to get the existing memory MetaFile data into your application buffer. After you have the MetaFile data in your application buffer, you can transfer the MetaFile data to the empty memory MetaFile by using the GpiSetMetaFileBits function. Note that with the GpiSetMetaFileBits and GpiQueryMetaFileBits functions you can transfer part of a MetaFile. This is made possible because the parameters to these functions include lengths, offsets, and pointers. To find the length of an existing memory MetaFile to do the neccessay memory management, use a function called *GpiQueryMetaFileLength*.

Now you can see that copying parts of a memory MetaFile isn't too tough. But in the the case where you want to make a copy of a complete memory MetaFile, you can use a function called *GpiCopyMetaFile*. This GPI function will create a whole new copy of the memory MetaFile in one function call and return you a new handle to the new copy. It doesn't get much simpler than that!

So far, we've discussed how you can create a MetaFile and save it to disk. We've also discussed how you can retrieve and save parts of memory MetaFiles to and from application memory. And finally, we've discussed how you can make a copy of an entire memory MetaFile. But what we haven't discussed is how to load a MetaFile from disk or play a memory MetaFile back to a presentation space. So let's start this discussion by seeing how an application can get a MetaFile saved on disk loaded back into a memory MetaFile form.

Getting a MetaFile from disk is really easy. Once you know the MetFile filename, you can use a function called *GpiLoadMetaFile*. This function will load the MetaFile from disk and create a memory MetaFile from it and then return a handle of the memory MetaFile to your application. At this point, this memory MetaFile is just like the memory MetaFile we discussed earlier. Once you have a memory MetaFile, you will eventually want to play its contents back into a presentation space so you can view the picture that it describes. The function that allows you to do this is called *GpiPlayMetaFile*.

The GpiPlayMetaFile function is not quite as straightforward as the other MetaFile functions we've discussed, but there is good reason for its extra complexity. Recall that when the MetaFile was originally created it recorded the presentation space environment and resource information. By doing this, the MetaFile is prepared to recreate the picture as close as possi-

ble to the original recording. Note, however, that the presentation space that you may be playing the MetaFile back into may have different environment and resource assignments than the MetaFile; hence, the extra complexity that is associated with the GpiPlayMetaFile function is a set of options that allow you to decide how to deal with these environment and resource differences. These options are passed to the GpiPlayMetaFile function in an array of options. This array can be up to nine elements in length and the toolkit provides definitions for both the array indexes and option (element) values. The following list shows each element in the option array and describes the possible values and their meanings that can be assigned to an option.

PMF_SEGBASE – Reserved. Must be 0.

PMF_LOADTYPE – This element specifies which transformations should be performed on the imported MetaFile.

- **LT_NOMODIFY or LT_DEFAULT** – This value specifies that the target presentation space viewing transform is to be used. Any change to the graphic field or default viewing transform found in the MetaFile will be ignored.
- **LT_ORIGINALVIEW** – This value specifies that the viewing transforms found in the MetaFile will be used during picture construction. The graphics field definition in the MetaFile is used to update the definition in the target presentation space.

PMF_RESOLVE – This element is reserved. Must be 0.

PMF_LCIDS – This element specifies how to deal with logical font definitions or bitmaps referenced by a local set identifier.

- **LC_NOLOAD** or **LC_DEFAULTS** – This value specifies that the local set identifiers for logical font definitions and bitmaps found in the MetaFile will be ignored. The local set identifiers in the target presentation space will be preserved and used for picture construction.
- **LC_LOADDISC** – This value specifies that the local set identifiers found in the MetaFile will be loaded into the target presentation space and will replace any identifiers with the same value.

PMF_RESET This element specifies whether the target presentation space should be reset with the page units and size specified in the MetaFile.

- **RES_NORESET** or **RES_DEFAULT** –This value specifies that no reset will be performed. If this value is used, you should insure that

the target presentation space page units are the same as those specified in the MetaFile.

RES_RESET – This value specifies that the target presentation space will be reset just as if it were newly created to have the page units and size as specified in the MetaFile.

PMF_SUPRESS This element specifies whether the MetaFile drawing orders will be played. This allows an application to first specify the **RES_RESET** option and then modify the presentation space environment and resources before the MetaFile is actually drawn.

SUP_NOSUPPRESS or **SUP_DEFAULT** –This value specifies that the picture defined in the MetaFile will be drawn.

SUP_SUPPRESS – This value specifies that the drawing will be suppressed; however, the target presentation space environment and resources will be established in accordance with the other options.

PMF_COLORTABLES This element specifies if the current color table in the target presentation space is to be replaced with the color table defined inthe MetaFile.

CTAB_NOMODIFY or **CTAB_DEFAULT** – This value specifies that the color table defined in the MetaFile will be ignored and the color table defined in the target presentation space will be used.

CTAB_REPLACE – This value specifies that the color table defined in the MetaFilewill replace the current color table defined in the target presentation space.

CTAB_REPLACEPALETTE – This value specifies that the target presentation pallet should be updated with the one defined in the MetaFile if it exists.

PMF_COLORREALIZEABLE–This element specifies whether the color table data contained in the MetaFile should be loaded with the LCOL_REALIZABLE option.

CREA_DOREALIZE – This value specifies that the color table should be loaded with the realizable option on.

CREA_NOREALIZE or **CREA_DEFAULT** – This value specifies that the color table should be loaded with the realizable option off.

PMF_DEFAULTS – This element specifies whether the drawing default found in the MetaFile should be used.

DDEF_IGNORE or **DDEF_DEFAULTS** – This value specifies that the drawing defaults found in the MetaFile should be ignored.

DDEF_LOADDISC – This value specifies that the drawing defaults found in the MetaFile should be used to replace those found in the target presentation space.

As you can see from the descriptions of the GpiPlayMetaFile function options, you can control the target presentation space environment and resources to meet your particular needs. An example of how you might use these options is to reproduce a picture described in a MetaFile to display as close to the original picture as possible. To do this, you would set the values in the options array as follows:

```
LT_ORIGINALVIEW
LC_LOADDISC
RES_RESET
SUP_NOSUPPRESS
CTAB_REPLACE
CREA_NOREALIZE
DDEF_LOADDISC
```

As it turns out, the MetaFile Viewer program that comes with this book does not reproduce the picture just as described by a given MetaFile. Instead, the MetaFile Viewer will scale a given MetaFile picture to fit to the size of the Viewer application window. Therefore, when you resize the MetaFile Viewer window, the picture will be scaled to fit the size of the client area. It scales in a way, however, that does not distort the final picture. To see how this is done, look at Listing 9.2.

```
/****************************/
/* Client window procedure. */
/****************************/
MRESULT EXPENTRY ClientWndProc(HWND hwnd, ULONG msg,
   MPARAM mp1, MPARAM mp2){
   static LONG metaOptions[ENTRIES];
   LONG segCount;
   CHAR title[]="MetaFile Listing File";
   CHAR fullFile[CCHMAXPATH]="*.LST";
   CHAR loadTitle[]="MetaFile To List";
   static CHAR metaFile[CCHMAXPATH]="*.MET";
   SIZEL sizl;
   static BOOL wetPaint=FALSE;
   LONG byteCount;
   LONG offset=0;
```

LISTING 9.2 MetaFile viewer program.

```
static CHAR *buffer,*metaFileData;
APIRET rc;
POINTL origin;
RECTL window,boundaryData;
MATRIXLF matrix;
MATRIXLF viewMatrix={0x10000,0L,0L,0L,0x10000,0L,0L,0L,1L};
FIXED scale[2];
POINTL bigWindow[4];
switch(msg){
/********************************/
/* Process pull-down menu items. */
/********************************/
case WM_COMMAND:
   switch (COMMANDMSG(&msg)->cmd){
/*****************************************************************/
/* Process load pull-down option.  If a MetaFile was passed as */
/* a command line parameter, commandParm will be TRUE and      */
/* MetaFileName will have been set.                            */
/*****************************************************************/
case IDM_VOPEN:
if(!commandParm){
   memset(&fileDlg,0,sizeof(FILEDLG));
   /************************************/
   /* Initialize file dialog structure. */
   /************************************/
   fileDlg.cbSize=sizeof(FILEDLG);
   fileDlg.fl=FDS_HELPBUTTON | FDS_CENTER | FDS_OPEN_DIALOG;
   fileDlg.pszTitle=loadTitle;
   strcpy(fileDlg.szFullFile,metaFile);
   /***************************************/
   /* Display the dialog and get the file. */
   /***************************************/
   hwndFileDlg=WinFileDlg(HWND_DESKTOP, hwndFrame,&fileDlg);
   strcpy(metaFileName,fileDlg.szFullFile);
   strcpy(metaFile,fileDlg.szFullFile);
   if(fileDlg.lReturn==DID_CANCEL)return(MRESULT)TRUE;
}
/*****************************************************/
/* Clear commandParm flag so this can be TRUE only once. */
/*****************************************************/
commandParm=FALSE;
if(metOn){          // If memory MetaFile existed delete it.
   metOn=FALSE;
   GpiDeleteMetaFile(hmf);
}
/*******************************/
/* Load the MetaFile from disk. */
/*******************************/
hmf=GpiLoadMetaFile(hab,metaFileName);
if(hmf!=GPI_ERROR){
   metOn=TRUE;      // Flag MetaFile loaded.
```

LISTING 9.2 (Continued).

```
/*********************************************/
/* Clear out window and play MetaFile to it. */
/*********************************************/
bigWindow[0].x=0;
bigWindow[0].y=0;
bigWindow[1].x=WinQuerySysValue(HWND_DESKTOP,SV_CXSCREEN);
bigWindow[1].y=WinQuerySysValue(HWND_DESKTOP,SV_CYSCREEN);
GpiBitBlt(hps,hps,2L,bigWindow,ROP_ONE,BBO_AND);
/****************************************************/
/* First set up options to get MetaFile page units. */
/****************************************************/
metaOptions[PMF_SEGBASE]=0L;
metaOptions[PMF_LOADTYPE]=LT_NOMODIFY;
metaOptions[PMF_RESOLVE]=0L;
metaOptions[PMF_LCIDS]=LC_LOADDISC;
metaOptions[PMF_RESET]=RES_RESET;
metaOptions[PMF_SUPPRESS]=SUP_SUPPRESS;
metaOptions[PMF_COLORTABLES]=CTAB_REPLACE;
metaOptions[PMF_COLORREALIZABLE]=CREA_NOREALIZE;
metaOptions[PMF_DEFAULTS]=DDEF_LOADDISC;
GpiPlayMetaFile(hps,hmf,ENTRIES,metaOptions,&segCount,
   sizeof(descr),descr);
/**********************************/
/* Query window size for scaling. */
/**********************************/
WinQueryWindowRect(hwnd,&window);
GpiConvert(hps,CVTC_DEVICE,CVTC_PAGE,2L,(PPOINTL)(&window));
/****************************************************/
/* Now modify options to get collect boundary data. */
/****************************************************/
metaOptions[PMF_RESET]=RES_NORESET;
metaOptions[PMF_SUPPRESS]=SUP_NOSUPPRESS;
GpiSetDrawControl(hps,DCTL_DISPLAY,DCTL_OFF);
GpiSetDrawControl(hps,DCTL_BOUNDARY,DCTL_ON);
GpiResetBoundaryData(hps);
GpiPlayMetaFile(hps,hmf,ENTRIES,metaOptions,
   &segCount,sizeof(descr),descr);
GpiQueryBoundaryData(hps,&boundaryData);
/********************************************************/
/* Calculate scale factors for both X and Y directions. */
/* Only use the smaller factor so drawing will fit      */
/* in the window.                                       */
/********************************************************/
scale[0]=((window.xRight-window.xLeft) * 0x10000) /
   boundaryData.xRight-boundaryData.xLeft);
scale[1]=((window.yTop-window.yBottom) * 0x10000) /
   (boundaryData.yTop-boundaryData.yBottom);
if(scale[0]<scale[1])scale[1]=scale[0];
else scale[0]=scale[1];
origin.x=boundaryData.xLeft;
origin.y=boundaryData.yBottom;
GpiResetPS(hps,GRES_ALL);
```

LISTING 9.2 (Continued).

```
      GpiScale(hps,&matrix,TRANSFORM_REPLACE,scale,&origin);
      origin.x=window.xLeft-boundaryData.xLeft;
      origin.y=window.yBottom-boundaryData.yBottom;
      GpiTranslate(hps,&matrix,TRANSFORM_ADD,&origin);
      GpiSetDefaultViewMatrix(hps,9L,&matrix,TRANSFORM_REPLACE);
      /************************************/
      /* Now draw so picture can be seen. */
      /************************************/
      GpiSetDrawControl(hps,DCTL_DISPLAY,DCTL_ON);
      GpiSetDrawControl(hps,DCTL_BOUNDARY,DCTL_OFF);
      GpiPlayMetaFile(hps,hmf,ENTRIES,metaOptions,
         &segCount,sizeof(descr),descr);
      GpiResetPS(hps,GRES_ALL);
      /***************************************************/
      /* Flag just displayed so WM_PAINT will not re-draw. */
      /***************************************************/
      wetPaint=TRUE;
   }
   /************************************/
   /* Enable menu item if not browsing. */
   /************************************/
   if(!browsing)
      WinSendMsg(WinWindowFromID(hwndFrame,FID_MENU),
         MM_SETITEMATTR,MPFROM2SHORT(IDM_VLIST,TRUE),
         MPFROM2SHORT(MIA_DISABLED,0));
      return(MRESULT)TRUE;
      }
   break;
   }
   /**************************************************/
   /* Let default routine process message and return. */
   /**************************************************/
   return WinDefWindowProc(hwnd,msg,mp1,mp2);
}
```

LISTING 9.2 (Continued).

Listing 9.2 is a code fragment from `VIEWMET.C` that deals with the Open MetaFile pull-down menu item. As you can see, the first thing done in this routine depends on a flag called commandParm. This flag indicates if a filename was passed when the MetaFile Viewer program was first started. If this flag is set TRUE, this logic assumes a valid filename already exists in the variable called MetaFileName and proceeds to work with this filename. (Note, because commandParm is set to FALSE every time through this routine and never gets reset to TRUE after startup, it can only be TRUE one time through this routine.) If commandParm is FALSE on entry to this routine, a file dialog is displayed to prompt the user for a MetaFile filename. If the user enters a valid filename, then the filename is copied to MetaFileName and the routine continues.

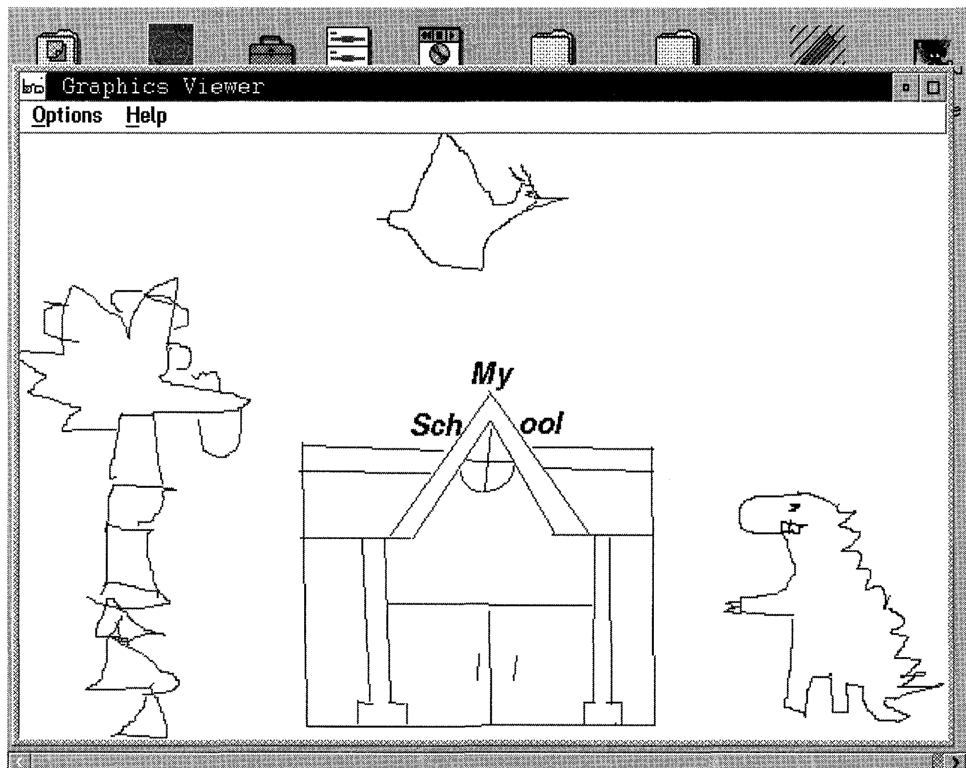
After a valid filename has been placed in MetaFileName, another flag called metOn is tested. This flag indicates if another MetaFile has already been opened by this program. If another MetaFile has been opened, then that MetaFile is deleted from memory with the GpiDeleteMetaFile function. Once the old MetaFile has been deleted, the new MetaFile is loaded into memory with the GpiLoadMetaFile function. If the new MetaFile has an error loading, this routine just falls through. In the usual good case, the metOn flag is set to TRUE and we get ready to display the picture described by the MetaFile.

In preparation for displaying the picture, this routine will use the GpiBitBlt function to force the MetaFile Viewer client area to be white (all bits on). Now, because we don't know the units defined in the MetaFile, we play the MetaFile with the GpiPlayMetaFile function using the `RES_RESET` and `SUP_SUPPRESS` options. (The other options set in the options array are set to reproduce the picture as defined in the MetaFile, but aren't really interesting until the next step!) Because the `RES_RESET` option is used, the target presentation space will take on the page units described in the MetaFile. But because the `SUP_SUPPRESS` option was also specified, the drawing will not really occur. Now that the presentation space has the correct units for the presentation page, the next time we play the MetaFile we can collect boundary data for the picture. By doing this, we can scale the picture to fit the current size of the client window. Before we collect boundary data, we use the WinQueryWindowRect function to get the current size of the client window. This size information is saved and will be used later for determining the correct scaling factor.

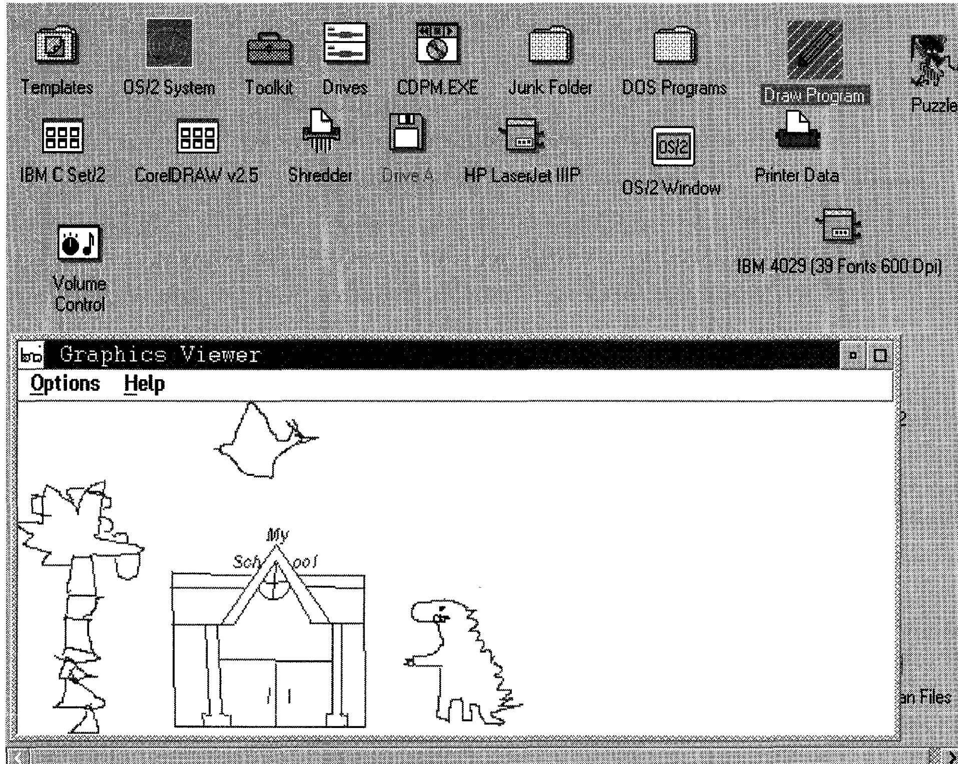
Before we play the MetaFile to collect boundary data, we change the MetaFilePlay options to `RES_NORESET` and `SUP_NOSUPPRESS`. We also change the current draw controls to collect boundary information and to not display output. By doing this, the MetaFile will play so we can collect the boundary information, but a picture will still not appear in our client area. Right before we play the MetaFile to collect boundary data, we issue the GpiResetBoundaryData function to clear out any previous boundary data information. Then, as soon as the MetaFile has been played, the boundary data is queried with the GpiQueryBoundaryData function . Once we have boundary information for our MetaFile picture, x and y scaling factors are calculated. The smaller of the two scaling factors is applied to both x and y directions so that the picture will scale inside the client area and not be dis-

torted. The boundary data is then used to determine an x and y translation point so the picture will be moved to the bottom left of the client area. But before the actually scale and translation tranformations are applied to the default viewing matrix, the presentation space is reset to clear out all previous transforms that may exist. Then, right before the MetaFile is played for the final time, the draw controls are changed again so boundary data isn't collected and so the drawing will be visible when played.

Now, when the GpiPlayMetaFile function is used, the MetaFile picture will appear in the window! After the play operation, the presentation space is reset again for the next time the picture needs to be redrawn. A flag called wetPaint is then set to inform the `WM_PAINT` routine that this picture has been drawn so it can skip redrawing it again right away. Finally, if we are not currently using the Browse utility to browse an existing MetaFile order structure, the List pull-down menu item is enabled so the user can view the MetaFile order content.



SCREEN 9.1 MetaFile picture large size.



SCREEN 9.2 MetaFile picture small size.

Although not shown in Listing 9.2, the WM_PAINT routine has similar logic as was just reviewed for redrawing the picture. In particular, if you resize the MetaFile Viewer application window, the WM_PAINT message will rescale the MetaFile picture to fit the new size window. To see what is meant by scaling to fit the MetaFile Viewer Window, look at Screens 9.1 and 9.2. These two screen captures are of the same MetaFile picture, but the MetaFile Viewer application window size is different.

Internal Stucture of a MetaFile

As indicated eariler, the internal structure of a MetaFile contains graphic segments and orders just like those discussed in Chapter 5. But the MetaFile encapsulates this segment and order information in data structures called *structured fields*. In fact, the entire content of a MetaFile is nothing more than a sequential set of structured fields. Besides segment and order information, a MetaFile also encapsulates a lot of other information in these

structured fields such as environment attributes, resources, descriptions, and so on.

The MetaFile structured field is composed of three parts. The first part of a MetaFile structured field is called the *header*. The header is also made up of several fields. The first field in the header is a length field. This length field is 2 bytes long and is the length of the entire structured field (including the length field). The second field of the header is a 3-byte identifier. This identifier uniquely identifies the type of the structured field. The third and fourth fields of the header are called flags and segment sequence number. The flags field is 1 byte in length, while the segment sequence number is 2 bytes. These two fields, flags and segment squence number, are always 0!

The second part of a MetaFile structured field is positional parameters. These positional parameters are optional and depend on the type of the structured field.

Finally, after the positional parameters, the structured field may have nonpositional parameters called *triplets*. A triplet consists of a 1-byte length field, a 1-byte identifier, and perhaps several self-defining parameters. Like before, the length field includes the entire length of the triplet including the length field and the identifier uniquely identifies the type of triplet within the structured field.

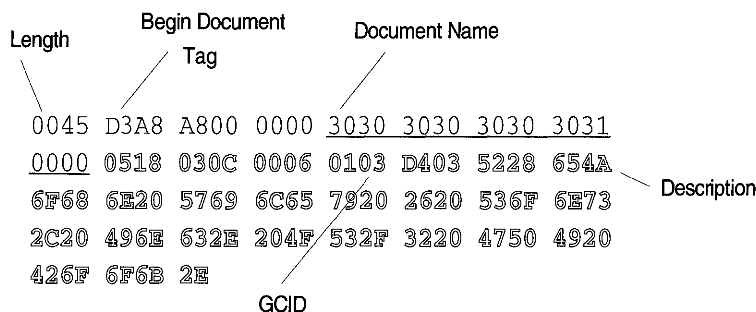


FIGURE 9.2 Begin document structure field.

With all that structure stuff said, let's look at an example of a MetaFile structured field so you can see the parts more clearly. Figure 9.2 shows a *Begin Document* structured field that was generated when we saved a drawing with our graphic editor to an OS/2 MetaFile. By studying Figure 9.2, you should be able to see the different parts of the structured field as previously described.

You've probably already figured out that in one of these structured field types, you can find the graphic order data that was dicussed in Chapter 5. You're right! A structure field type called *Graphics Data* can contain order data just as described earlier.

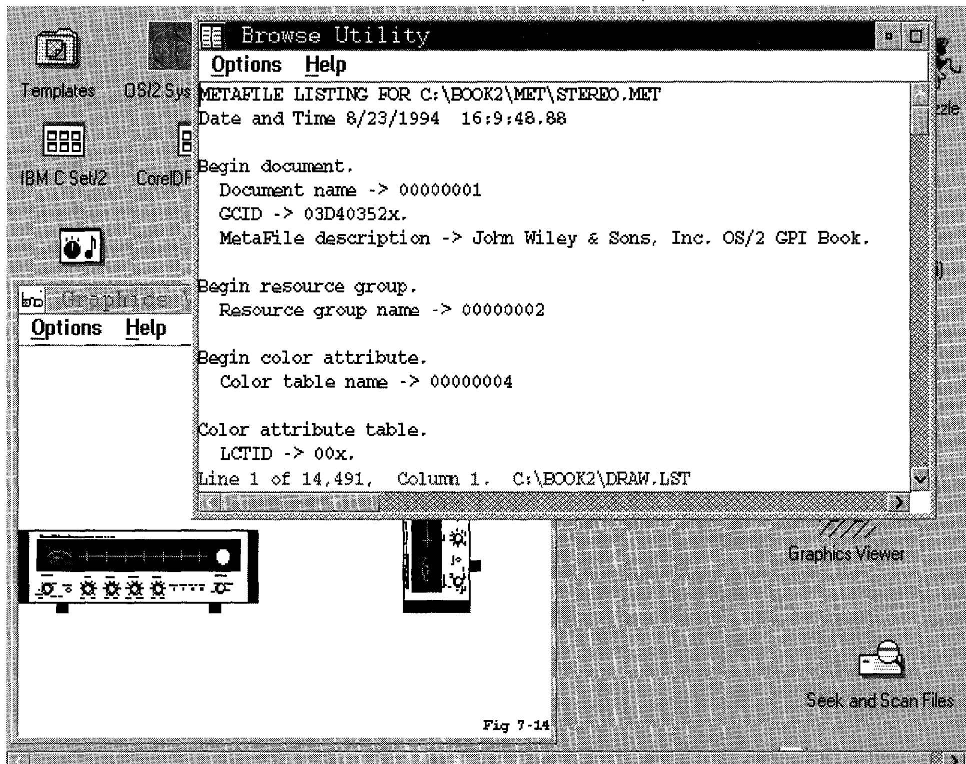
As already stated, a MetaFile is made up of several structured fields which are of a variety of types. But besides the example of the Begin Document structured field shown in Figure 9.2, this book does not show the definition of the other structured fields available. For structured field definitions, you should consult an IBM technical reference. What this book does provide, however, is a utility that will parse the content of the structured fields for a given OS/2 MetaFile and display the output with the Browse utility! Again, because this utility comes with the book, so does all the source code. By having this MetaFile parser available to use when developing the graphic editor, we could see how our graphic objects were being generated and more quickly understand this environment. Like the object viewer, this utility is a great debug aid. What makes this utility a bit different than the object viewer is that in the graphic editor you can see more of the presentation space environment information in addition to graphic order data. Also, you can see how other graphic editors produced their objects if they produce an OS/2 MetaFile!

MetaFile Parser

Like the graphic editor, the MetaFile Viewer program will also display graphic object information. The difference between these features, however, is that the MetaFile Viewer will parse the entire MetaFile structure, which also includes the graphic order data. To access the MetaFile Viewer's parser feature you need to use the List pull-down menu item. Note that the List menu item is only enabled when a MetaFile picture is being displayed and no other MetaFile listing is being browsed via this feature. Once you select the List menu item, a file dialog box will appear and prompt you for a

filename for where you would like the MetaFile listing data to be stored. After you have entered a valid filename, the MetaFile parser feature will create the output in the specified file and automatically start the Browse utility to display the output.

You may have noticed two other menu items named *Snapshot* and *Snapshot Length* in the MetaFile Viewer Options pull-down menu. If Snapshot is checked before you use the List menu item, then the parser feature will also dump hexadecimal data for each structured field it encounters. The amount of hexadecimal data that is dumped for each structured field depends on the length of the structured field and the value set with the Snapshot Length menu item. The amount of hexadecimal data dumped will either be the length of the structured field or the Snapshot Length value (whichever is smaller). Screen 9.3 shows an example of the Browse utility displaying some parsed MetaFile data. You should note, however, that the filename you chose to place the parsed output into can be copied and browsed with any text editor.



SCREEN 9.3 MetaFile data being browsed.

The actual MetaFile parsing function doesn't really happen in the main source code for the MetaFile Viewer (VIEWMET.C). Instead, the List pull-down menu item routine will pass several parameters to a function called *ParseMet* and it is this function that starts the parsing function. The parameters passed to ParseMet are our window handle, the output filename, the MetaFile filename, the snapshot flag, and the snapshot length.

The source code for the ParseMet function is found in a file called PARSEMET.C. Listing 9.3 shows a small fraction of the source code used to produce the MetaFile parser function. The main things missing from Listing 9.3 that are found in PARSEMET.C are several of the CASE statements used for parsing MetaFile structured fields. The only CASE statements shown in Listing 9.3 for parsing MetaFile structured fields are for the Begin Document, Grapic Data, and End Document structured fields. Be assured, however, that the other CASE statements are in the PARSEMET.C source code found on the book diskette!

There really aren't a whole lot of exciting GPI function calls used in the ParseMet function. Parsing MetaFile data is just plain tedious C logic which involves breaking apart each structured field and writing the meaning of the structured field out to a file for later viewing . Hence, not a lot of explanation of the ParseMet function is going to be given here.

There are, however, a couple of interesting points to be made about ParseMet.C. Look at Listing 9.3 and notice how the MetaFile data is made available to the ParseMet function. First the MetaFile is loaded into system memory with the GpiLoadMetaFile function. Then the length of the MetaFile data is determined with the GpiQueryMetaFileLength function, memory is allocated, and the MetaFile data is made available to our function by using the GpiQueryMetaFileBits function.

```
/**********************************/
/* Function Parse Graphic Segment. */
/**********************************/
BOOL ParseMet(HWND hwndcaller, char *filename, char *metaname,
   BOOL snapFlag, LONG snapLength){
   #include "ordertab.c" extern HAB hab;
   static CHAR fileData[256];
   DATETIME dateTime;
   HFILE fileHandle;
   ULONG wrote,action,rc;
   INT stringlen,i,tempLength,G=4,T,nameLength,tripType;
   BOOL indicator; HMF hmf=GPI_ERROR;
```

LISTING 9.3 MetaFile parser routines.

```
   LONG byteCount;
   LONG offset=0;
   ULONG identifier;
   static CHAR *buffer,*metData,*workPtr;
   USHORT fieldOffset;
   ULONG dataLen,edatalen,dataLen1,j,j1,j2,totalData,snapLines;
   BOOL edt=FALSE,firstPass;
   CHAR tempByte,byteLength;
   LONG maxcnt,linecnt,orderOffset,orderAdder,blankcnt;
   ERRORID errorCode;
   /****************************************/
   /* Start of output to listing function. */
   /****************************************/
   hmf=GpiLoadMetaFile(hab,metaname);
   byteCount=GpiQueryMetaFileLength(hmf);
   rc=DosAllocMem((PVOID)&buffer,(ULONG)byteCount,
      PAG_COMMIT | PAG_READ | PAG_WRITE);
   GpiQueryMetaFileBits(hmf,offset,byteCount,buffer);
   metData=buffer;
   fieldOffset=0;
   edt=FALSE;
   /*****************************************/
   /* Open file that was typed or selected. */
   /*****************************************/
   action=2;
   rc=DosOpen(filename,&fileHandle,&action,FILE_SIZE,
      FILE_NORMAL,1,0x40,EABUF);
/**********************************************************************/
/* If file was available, not found, or explicit open failed         */
/* then continue to create the list file, otherwise file not available.*/
/**********************************************************************/
if((rc==0) || (rc==2) || (rc==110)) {
   DosClose(fileHandle);
   DosOpen(filename,
      &fileHandle,&action,FILE_SIZE,FILE_ARCHIVE,
      OPEN_FILE | CREATE_FILE,
      DASD_FLAG | INHERIT | WRITE_THRU |
      FAIL_FLAG | SHARE_FLAG | ACCESS_FLAG,
      EABUF);
   DosSetFileSize(fileHandle,FILE_SIZE);
   /*****************************************************/
   /* Print out heading title and time info for listing. */
   /*****************************************************/
   stringlen=sprintf(fileData,"METAFILE LISTING FOR ");
   strcat(fileData,metaname);
   strcat(fileData,"\n");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   DosGetDateTime(&dateTime);
   stringlen=sprintf(fileData,"Date and Time %d/%d/%d  %d:%d:%d.%d \n",
      dateTime.month,
      dateTime.day,
      dateTime.year,
```

LISTING 9.3 (Continued).

```
      dateTime.hours,
      dateTime.minutes,
      dateTime.seconds,
      dateTime.hundredths
      );
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   while((fieldOffset<byteCount) && !edt){
      fieldOffset=(((USHORT)(*(metData+0)))*256) + (*(metData+1));
      identifier=(((ULONG)(*(metData+2)))*65536) +
         (((ULONG)(*(metData+3)))*256) + (*(metData+4));
      workPtr=metData;
   /************************************************************/
   /* Output up to snapLength bytes of data in structured field. */
   /************************************************************/
   if(snapFlag){
      sprintf(fileData,"\nSnap shot: ");
      DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
      i=0;
      snapLines=snapLength/40;
      if(snapLength % 40)snapLines++;
      while((i<snapLength) && (i<fieldOffset)){
   /*********************************************************/
   /* Put out up to number of possible snap lines of output.*/
   /*********************************************************/
   for(j2=0;j2<snapLines;j2++){
      /***************************************************/
      /* Put a space between output groups.  5 per line. */
      /***************************************************/
      for(j1=0;j1<5;j1++){
         /**********************************/
         /* Put 8 bytes in an output group. */
         /**********************************/
         for(j=0;j<8;j++){
            sprintf(fileData,"%02X",workPtr[i]);
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
            i++;
            if((i==fieldOffset) || (i==snapLength))break;
         } // End of j for loop.
         if((i==fieldOffset) || (i==snapLength))break;
         sprintf(fileData, " ");
         DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
      } // End of j1 for loop.
         if((i==fieldOffset) || (i==snapLength))break;
         sprintf(fileData, "\n            ");
         DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
      } // End of j2 for loop. }
   // End outside while.
}
switch(identifier){
/**********************************/
/* Begin document structured field. */
/**********************************/
```

LISTING 9.3 (Continued).

```
case BDT:
   sprintf(fileData,"\nBegin document.\n  Document name -> ");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   workPtr=workPtr+8;     // Point to document name field.
   strncpy(fileData,workPtr,8);
   fileData[8]=0;         // Make string into ASCIIZ string.
   strcat(fileData,"\n  GCID -> ");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   workPtr=workPtr+17;    // Point to GCID field and format to hex.
   for(i=0;i<4;i++)
   sprintf(&fileData[i*2],"%02X",workPtr[i]);
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   workPtr=workPtr+4;
   i=(int)*workPtr;
   *fileData=0;
   strcat(fileData,"x.\n  MetaFile description -> ");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   strncpy(fileData,workPtr+2,i-2);
   fileData[i-2]=0;       // Make string into ASCIIZ string.
   strcat(fileData,"\n");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   break;
/******************/
/* Graphics Data. */
/******************/
case GAD:
   sprintf(fileData,"\nGraphics data.\n");
   DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
   workPtr=workPtr+8;
   while(workPtr<(metData+fieldOffset)){
      if(workPtr[0]!='\x70'){
         sprintf(fileData,"  Not a segment!!.\n");
         DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         break;
         }
         sprintf(fileData,"\n  Segment ID %02X%02X%02X%02Xx.\n",
            workPtr[2],workPtr[3],workPtr[4],workPtr[5]);
         DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         if(workPtr[6]&'\x80'){
            sprintf(fileData,"  Invisible.\n");
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
         if(workPtr[6]&'\x40'){
            sprintf(fileData,"  Propagate invisibility.\n");
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
         if(workPtr[6]&'\x20'){
            sprintf(fileData,"  Detectable.\n");
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
         if(workPtr[6]&'\x10'){
            sprintf(fileData,"  Propagate detectability.\n");
```

LISTING 9.3 (Continued).

```
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
         if(workPtr[6]&'\x02'){sprintf(fileData,"  Dynamic.\n");
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
         if(workPtr[6]&'\x01'){
            sprintf(fileData,"  Fast chaining.\n");
            DosWrite(fileHandle,(PVOID) fileData,strlen(fileData),&wrote);
         }
         if(workPtr[7]&'\x80'){
            sprintf(fileData,"  Non-chained.\n");
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
         if(workPtr[7]&'\x10'){
            sprintf(fileData,"  Prolog.\n");
            DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
         }
      /**********************************************************/
      /* Due to a bug found in OS/2 2.1, we only parse the low     */
      /* order word of data length. The high order word was not    */
      /* set correctly in 2.1 but should be correct in Warp. In    */
      /* order to maintain compatibility with old MetaFiles, we    */
      /* ignore the high order word, hence only MetaFile with data */
      /* packets of less than 65536 bytes will parse correctly.    */
      /* That should be the vast majority of MetaFiles.            */
      /**********************************************************/
      dataLen=(((ULONG)(*(workPtr+8)))*256) + (*(workPtr+9));
      workPtr=workPtr+16;
      orderOffset=0;
      edatalen=dataLen;
      /***************/
      /* Parse data. */
      /***************/
      while(edatalen){
      /*************************************/
      /* Determine if this is a single byte */
      /* order.                             */
      /*************************************/
      if((workPtr[orderOffset]==0) || (workPtr[orderOffset]==255)){
         orderAdder=1;
         parseTab[workPtr[orderOffset]](fileHandle,
            workPtr+orderOffset,G);
      }
      /***********************************/
      /* Determine if this is an extended */
      /* order and print its data.        */
      /***********************************/
      if(workPtr[orderOffset]==254){
         orderAdder=(workPtr[orderOffset+2])*256;
         orderAdder=orderAdder+(workPtr[orderOffset+3]);
         orderAdder=orderAdder+4;
         parseTab[workPtr[orderOffset]](fileHandle,
```

LISTING 9.3 (Continued).

```
                workPtr+orderOffset, G);
          }
          /***********************************/
          /* Determine if this is a two      */
          /* byte order and print its data.  */
          /***********************************/
          if((workPtr[orderOffset]&8) && (workPtr[orderOffset]<=127)){
             orderAdder=2;
             parseTab[workPtr[orderOffset]](fileHandle,
                workPtr+orderOffset,G);
          }
          /******************************/
          /* Determine if this is a long */
          /* order and print its data.   */
          /******************************/
          if( (workPtr[orderOffset]!=0) &&
             (workPtr[orderOffset]!=255) &&
             (workPtr[orderOffset]!=254) &&
             ( !((workPtr[orderOffset]<=127) &&
             (workPtr[orderOffset]&8)) ) ) {
          orderAdder=workPtr[orderOffset+1]+2;
          if(edatalen<orderAdder){
             sprintf(fileData,"  DATA LENGTH PROBLEM!!\n");
             DosWrite(fileHandle,(PVOID)fileData,
                strlen(fileData),&wrote);
             break;
          }
          else parseTab[workPtr[orderOffset]](fileHandle,
             workPtr+orderOffset,G);
          }
          edatalen=edatalen-orderAdder;
          orderOffset=orderOffset+orderAdder;
       }
       workPtr=workPtr+dataLen;
    } /* endwhile */
    break;
    /********************/
    /* End of document. */
    /********************/
    case EDT:
       sprintf(fileData,"  \nEnd document.\n  Document name -> ");
       DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
       workPtr=workPtr+8;    // Point to document name field.
       strncpy(fileData,workPtr,8); fileData[8]=0;
          // Make string into ASCIIZ string.
       strcat(fileData,"\n");
       DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
       edt=TRUE;
       break;
    default:
       sprintf(fileData,"\nUnknown identifier -> %06Xx.\n",identifier);
       DosWrite(fileHandle,(PVOID)fileData,strlen(fileData),&wrote);
       break;
```

LISTING 9.3 (Continued).

```
} /* endswitch */
metData+=fieldOffset;
}
DosClose(fileHandle);
indicator=TRUE;
}
else
   indicator=FALSE;
DosFreeMem(buffer);
return indicator;
}
```

LISTING 9.3 (Continued).

You can also see from Listing 9.3 that structured fields are parsed until data is exhausted or the End Document structured field is encountered. (edt is a flag set when the End Document structured field is found. The End Document structured field should always be found before we run out of data!)

Finally, look at the logic inside the CASE statement for the Graphic Data structured field in Listing 9.3. Once we get past parsing the header and some of the parameters for the Graphic Data structured field, we get to a part that is the graphic order data. (This starts right after the block comment with text **Parse Data**.) The logic to parse the graphic order data is identical to that found in the graphic editor object parser. In fact, the functions pointed to by the parseTab are the very same functions used by the graphic editor. These order parsing functions are found in a file named PORDERS.C on the book diskette.

So, there you have it! Not only do you have an example of how to use MetaFiles and examples of the GPI functions to work with MetaFiles, but you also have a program that will let you view a MetaFile's internal structure as well as the picture described by the MetaFile. Gee, I love this GPI stuff!

APPENDIX A

GPI Functions

This Appendix is designed to give you a quick reference to all the GPI functions available in OS/2. The GPI functions are listed in alphabetical order and give the input/output parameter types for the function followed by its description. The definition of the parameter types for the functions are not given, but the type definitions are fairly intuitive by their name. If you need a definition for a parameter type, refer to an OS/2 technical reference.

return=**GpiAnimatePalette** (palette, format, startIndex, elements, dataArea);

(LONG) return	Number of remapped colors.
(HPAL) palette	Handle to palette.
(ULONG) format	Format of table entries.
(ULONG) startIndex	Starting index.
(ULONG) elements	Count of elements in data area.
(PULONG) dataArea	Pointer to user-defined element data.

This function changes the color values of animating indexes in a palette.

return=**GpiAssociate** (hps, hdc);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HDC) hdc	Handle to device context.

This function associates a presentation space with a device context. By specifying 0 for the device context parameter, the presentation space can be disassociated.

return=**GpiBeginArea** (hps, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(ULONG) options	Area options.

This function begins the construction of an area for a presentation space.

return=**GpiBeginElement** (hps, type, description);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Type to associate with the element.
(PSZ) description	Pointer to text description for the element.

This function marks the beginning of a user-defined element in graphic segment.

return=**GpiBeginPath** (hps, pathID);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) pathID	Path identifier.

This function marks the start of the beginning of a path. An indicator is returned indicating the success or failure of the function call.

return=**GpiBitBlt** (target, source, pointCount, pointArray, rop, options);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) target	Handle to target presentation space.
(HPS) source	Handle to source presentation space.
(LONG) pointCount	Number of points in point array.
(PPOINTL) pointArray	Pointer to point array.
(LONG) rop	Raster operation mix function.
(ULONG) options	Options.

This function copies bitmap image data from one rectangular area to another.

return=**GpiBox** (hps, control, cornerPoint, xRounding, yRounding);

(LONG) return	Number of correlation hits or indicator for success or failure.

(HPS) hps	Handle to presentation space.
(LONG) control	Outline and fill control.
(PPOINTL) cornerPoint	Pointer to the coordinate of the ending corner. (Opposite of corner from current position.)
(LONG) xRounding	Length of the x axis of an ellipse used for rounding each corner.
(LONG) yRounding	Length of the y axis of an ellipse used for rounding each corner.

This function draws a rectangle box with rounded corners at the current position.

return=**GpiCallSegmentMatrix** (hps,segmentID, elements, transform, options);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segmentID	Segment identifier to be called.
(LONG) elements	Number of elements in the transform matrix to examine.
(PMATRIXLF) transform	Pointer to instance transform matrix.
(LONG) options	Transformation options.

This function calls a graphics segment and applies an instance transform to it.

return=**GpiCharString** (hps, count, string);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of characters in string to draw.
(PCH) string	Pointer to characters to draw.

This function draws a text string.

return=**GpiCharStringAt** (hps, startPoint, count,string);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) startPoint	Pointer to a starting point for the string.

(LONG) count	Number of characters in string to draw.
(PCH) string	Pointer to characters to draw.

This function draws a text string at a given starting point.

return=**GpiCharStringPos** (hps, rectangle, options, count, string, increments);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to rectangle structure which defines the background for the text string.
(ULONG) options	Formatting options for the string.
(LONG) count	Number of characters in string to draw.
(PCH) string	Pointer to characters to draw.
(PLONG) increments	Pointer to an array of values used for character placement.

This function draws a text string with formatting options.

return=**GpiCharStringPosAt** (hps, startPoint, rectangle, options, count, string, increments);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) startPoint	Pointer to a starting point for the string.
(PRECTL) rectangle	Pointer to rectangle structure which defines the background for the text string.
(ULONG) options	Formatting options for the string.
(LONG) count	Number of characters in string to draw.
(PCH) string	Pointer to characters to draw.
(PLONG) increments	Pointer to an array of values used for character placement.

This function draws a text string with formatting options at a given position.

return=**GpiCloseFigure** (hps);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function closes a figure within a path.

return=**GpiCloseSegment** (hps);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function closes the current graphic segment.

return=**GpiCombineRegion** (hps, target, source1, source2, mode);

(LONG) return	Complexity of resulting region and indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) target	Handle to target region.
(HRGN) source1	Handle to first source region.
(HRGN) source2	Handle to second source region.
(LONG) mode	Method of combining regions.

This function combines two regions.

return=**GpiComment** (hps, length, comment);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) length	Length of comment string.
(PBYTE) comment	Pointer to comment string.

This function adds a comment to the current graphic segment.

return=**GpiConvert**(hps, source, target, count, points);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) source	Source coordinate space.
(LONG) target	Target coordinate space.
(LONG) count	Number of coordinate pairs in array.
(PPOINTL) points	Array of coordinate pairs.

This function converts coordinate pairs from one coordinate space to another coordinate space.

return=**GpiConvertWithMatrix** (hps, count, points, elements, transform);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of coordinate pairs in array.
(PPOINTL) points	Array of coordinate pairs.
(LONG) elements	Number of elements in transform matrix.
(PMATRIXLF) transform	Pointer to instance transform matrix.

This function converts coordinate pairs from one coordinate space to another coordinate space using a specified tranformation matrix.

return=**GpiCopyMetaFile** (metaFile);

(HMF)return	Handle of new MetaFile.
(HMF)metaFile	Handle of source MetaFile.

This function creates a new MetaFile and copies the source MetaFile contents to it.

return=**GpiCorrelateChain** (hps, type, pick, maxHits, maxDepth, segTag);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Type of segments on which to perform correlation.
(PPOINTL) pick	Pointer to pick position.
(LONG) maxHits	Maximum number of hit to be returned.
(LONG) maxDepth	Maximum number of segment and tag pairs to be returned by each hit.
(PLONG) segTag	Pointer to an array where segment identifiers and tags are returned.

This function performs correlation on a retained segment chain.

return=**GpiCorrelateFrom** (hps, first, last, type, pick, maxHits, maxDepth, segTag);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.

(LONG) first	First segment identifier in chain to start correlation.
(LONG) last	Last segment identifier in chain for correlation.
(LONG) type	Type of segments on which to perform correlation.
(PPOINTL) pick	Pointer to pick position.
(LONG) maxHits	Maximum number of hits to be returned.
(LONG) maxDepth	Maximum number of segment and tag pairs to be returned by each hit.
(PLONG) segTag	Pointer to an array where segment identifiers and tags are returned.

This function performs correlation on part of a retained segment chain.

return=**GpiCorrelateSegment** (hps, segment, type, pick, maxHits, maxDepth, segTag);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier on which to perform correlation.
(LONG) type	Type of segments on which to perform correlation.
(PPOINTL) pick	Pointer to pick position.
(LONG) maxHits	Maximum number of hits to be returned.
(LONG) maxDepth	Maximum number of segment and tag pairs to be returned by each hit.
(PLONG) segTag	Pointer to an array where segment identifiers and tags are returned.

This function performs correlation on a specified graphic segment.

return=**GpiCreateBitmap**(hps, header, options, initData, infoTable);

(HBITMAP) return	Handle to bitmap or error indicator.
(HPS) hps	Handle to presentation space.
(PBITMAPINFOHEADER) header	Pointer to bitmap header that

	describes the format of the bitmap to be created.
(ULONG) options	Options.
(PBYTE) initData	Pointer to bitmap initialization data.
(PBITMAPINFO2) infoTable	Pointer to table that defines the format of the initialization data.

This function creates a bitmap.

return=**GpiCreateLogColorTable** (hps, options, format, start, elements, table);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) format	Format of table entries.
(LONG) start	Starting index.
(LONG) elements	Count of elements in table.
(PLONG) table	Pointer to user table.

This function defines the entries of the logical color table.

return=**GpiCreateLogFont** (hps, name, lcid, fontAttr);

(LONG) return	Match indicator.
(HPS) hps	Handle to presentation space.
(PSTR8) name	Logical font name.
(LONG) lcid	Local identifier used to refer to the font.
(LONG) fontAttr	Pointer to attributes require of the font.

This function gives a logical definition for a font.

return=**GpiCreatePalette**(hab, options, format, elements, table);

(HPAL) return	Handle to palette or error indicator.
(HAB) hab	Handle to anchor-block.
(ULONG) options	Options.
(LONG) format	Format of table entries.
(LONG) elements	Count of elements in table.
(PLONG) table	Pointer to user table.

This function creates a color palette.

return=**GpiCreatePS**(hab, hdc, size, options);

(HPS) return	Handle to presentation space or error indicator.
(HAB) hab	Handle to anchor-block.
(HDC) hdc	Handle to device context to be associated with presentation space.
(PSIZEL) size	Pointer to size of presentation page.
(LONG) options	Options.

This function creates a presentation space.

return=**GpiCreateRegion** (hps, count, rectangles);

(HRGN) return	Handle to a region.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of rectangles for region creation.
(PRECTL) rectangles	Pointer to array of rectangles.

This function creates a region from a series of rectangles.

return=**GpiDeleteBitmap** (bitmap);

(BOOL) return	Indicator of success or failure.
(HBITMAP) bitmap	Handle to bitmap.

This function deletes a bitmap.

return=**GpiDeleteElement** (hps);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.

This function deletes the element pointed to by the element pointer.

return=**GpiDeleteElement** (hps, first, last);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.
(LONG) first	Number of first element to delete.
(LONG) last	Number of last element in the range to delete.

This function deletes the elements in a range.

return=**GpiDeleteElementsBetweenLabels** (hps, first, last);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.
(LONG) first	Label marking the start of the elements to be deleted.

(LONG) last — Label marking the end of the elements to be deleted.

This function deletes the elements between the labels in a graphic segment.

return=**GpiDeleteMetaFile** (metaFile);

(BOOL) return — Indicator of success or failure.
(HMF) metaFile — Handle to MetaFile.

This function deletes a MetaFile.

return=**GpiDeletePalette**(palette);

(BOOL) return — Indicator of success or failure.
(HPAL) palette — Handle to palette.

This function deletes a color palette.

return=**GpiDeleteSegment** (hps, segment);

(BOOL) return — Indicator of success or failure.
(HPS) hps — Handle to presentation space.
(LONG) segment — Segment identifier.

This function deletes a retained graphic segment.

return=**GpiDeleteSegments** (hps, first, last);

(BOOL) return — Indicator of success or failure.
(HPS) hps — Handle to presentation space.
(LONG) first — First segment identifier in a range.
(LONG) last — Last segment identifier in a range.

This function deletes graphic segments in an identifier range.

return=**GpiDeleteSetId** (hps, lcid);

(BOOL) return — Indicator of success or failure.
(HPS) hps — Handle to presentation space.
(LONG) lcid — Local identifier.

This function deletes a local identifier.

return=**GpiDestroyPS**(hps);

(BOOL) return — Indicator of success or failure.
(HPS) hps — Handle to presentation space.

This function destroys a presentation space.

return=**GpiDestroyRegion** (hps, region);

(BOOL) return — Indicator of success or failure.
(HPS) hps — Handle to presentation space.

(HRGN) region	Handle to region.

This function destroys a region.

return=**GpiDrawBits** (hps, bits, infoTable, count, points, rop, options);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to target presentation space.
(PVOID) bits	Pointer to bitmap bits.
(PBITMAPINFO2) infoTable	Pointer to table the defines the format of the bitmap data.
(LONG) count	Number of points in bitmap data.
(PPOINTL) points	Pointer to point array.
(LONG) rop	Raster operation mix function.
(ULONG) options	Options.

This function draws bitmap bits.

return=**GpiDrawChain** (hps);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.

This function draws the segment chain.

return=**GpiDrawDynamics** (hps);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.

This function redraws the dynamic segments in the segment chain.

return=**GpiDrawFrom** (hps, first, last);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.
(LONG) first	First segment identifier to draw in the segment chain.
(LONG) last	Last segment identifier to draw in the segment chain.

This function draws a section of the segment chain.

return=**GpiDrawSegment** (hps, segment);

(BOOL) return	Indicator of success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier to draw.

This function draws a specified graphic segment.

return=**GpiElement** (hps, type, description, length, data);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Type to be associated with the element.
(PSZ) description	Pointer to description string.
(LONG) length	Length of element data.
(PBYTE) data	Pointer to element data.

This function adds an element to the current segment.

return=**GpiEndArea** (hps);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function ends the construction of an area.

return=**GpiEndElement** (hps);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function ends the element started by a GpiBeginElement function.

return=**GpiEndPath** (hps);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function ends the path started by a GpiBeginPath function.

return=**GpiEqualReqion** (hps, region1, region2);

(LONG) return	Equality or error indicator.
(HPS) hps	Handle to presentation space.
(HRGN) region1	Handle to first region.
(HRGN) region2	Handle to second region.

This function compares if two regions are equal.

return=**GpiErase** (hps);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function clears the output area of the device context associated with the presentation space to CLR_BACKGROUND color.

return=**GpiErrorSegmentData** (hps, segment, context);

(BOOL) return	Byte offset, element number, or error indicator.
(HPS) hps	Handle to presentation space.
(PLONG) segment	Pointer to segment in which the error occured.
(PLONG) context	Pointer to context of the error.

This function gives information about the last error that occured during a segment drawing operation.

return=**GpiFillPath** (hps, path, options);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) path	Path identifier.
(LONG) options	Fill options.

This function draws the inside of a path with the area attribute.

return=**GpiFloodFill** (hps, options, color);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) options	Flood fill options.
(LONG) color	Color.

This function fills an area bound by a given color or while on a given color.

return=**GpiFrameRegion** (hps, region, thickness);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region.
(PSIZEL) thickness	Pointer to thickness of frame.

This function draws a frame inside a region with the current pattern attribute.

return=**GpiFullArc** (hps, control, multiplier);

(LONG) return	Number of correlation hits or indicator for success or failure.

(HPS) hps	Handle to presentation space.
(LONG) region	Outline and interior control.
(FIXED) multiplier	Multiplier.

This function draws a full arc with its center at the current position.

return=**GpiGetData** (hps, segment, offset, format, length, buffer);

(LONG) return	Length of data returned or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier from which to get data.
(PLONG) offset	Offset into segment to start retrieving data.
(LONG) format	Coordinate type required.
(LONG) length	Length of data buffer.
(PBYTE) buffer	Pointer to data buffer.

This function gets data from a specified graphics segment.

return=**GpiImage** (hps, format, size, length, image);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) format	Format of image data.
(PSIZEL) size	Image size in PELS.
(LONG) length	Length of image data.
(PBYTE) image	Pointer to image data.

This function draws a rectangular image with the current position being the top-left corner.

return=**GpiIntersectClipRectangle** (hps, rectangle);

(LONG) return	Complexity of clipping or error indicator.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to rectangle.

This function creates a new clip region to the intersection of a specified rectangle and the current clip region.

return=**GpiLabel** (hps, label);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) label	Label.

This function creates an element with the specified label.

return=**GpiLine** (hps, endPoint);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) endPoint	Pointer to end point.

This function draws a line from the current position to the specified end point.

return=**GpiLoadBitmap** (hps, resource, bitmapID, width, height);

(HBITMAP) return	Handle to bitmap or error indicator.
(HPS) hps	Handle to presentation space.
(HMODULE) resource	Handle to resources.
(ULONG) bitmapID	Identifier of bitmap in resource file.
(LONG) width	Width of bitmap.
(LONG) height	Height of bitmap.

This function creates and loads a bitmap from a resource.

return=**GpiLoadFonts** (hab, fileName);

(BOOL) return	Indicator for success or failure.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Pointer to filename that contains font.

This function loads one or more fonts from a specified resource file.

return=**GpiLoadMetaFile** (hab, fileName);

(HMFL) return	Handle to a MetaFile or error indicator.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Pointer to filename that contains MetaFile data.

This function loads data from a file into a MetaFile.

return=**GpiLoadPublicFonts** (hab, fileName);

(BOOL) return	Indicator for success or failure.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Pointer to filename that contains font.

This function loads one or more fonts from a specified resource file for public use.

return=**GpiMarker** (hps, point);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to point.

This function draws a marker at the specified point.

return=**GpiModifyPath** (hps, path, mode);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) path	Path identifier.
(LONG) mode	Modification required.

This function modifies the specified path.

return=**GpiMove** (hps, point);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to point.

This function moves the current position to the specified point.

return=**GpiOffsetClipRegion** (hps, displacement);

(LONG) return	Complexity of clipping or error indicator.
(HPS) hps	Handle to presentation space.
(PPOINTL) displacement	Pointer to displacement.

This function moves a clipping region by a specified displacement.

return=**GpiOffsetElementPointer** (hps, offset);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) offset	Offset.

This function moves the element pointer within a segment by the specified offset.

return=**GpiOffsetRegion** (hps, region, offset);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region to be moved.
(PPOINTL) offset	Offset.

This function moves a region by a specified offset.

return=**GpiOpenSegment** (hps, segment);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier.

This function opens a graphic segment with the specified identifier.

return=**GpiOutlinePath** (hps, path, options);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) path	Path identifier.
(LONG) options	Options.

This function draws the outline of a path.

return=**GpiPaintRegion** (hps, region);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle of region to paint.

This function paints a region using the current pattern attribute.

return=**GpiPartialArc** (hps, center, multiplier, startAngle, sweepAngle);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) center	Pointer to center point of the arc.
(FIXED) multiplier	Multiplier.
(FIXED) startAngle	Start angle.
(FIXED) sweepAngle	Sweep angle.

This function draws a line followed by an arc.

return=**GpiPathToRegion** (hps, path, options);

(HRGN) return	Handle of region or error indicator.
(HPS) hps	Handle to presentation space.

(LONG) path	Path identifier.
(ULONG) options	Options.

This function converts a path to a region.

return=**GpiPlayMetaFile** (hps, metaFile, count, options, segCount, count2, description);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HMF) metaFile	Handle to MetaFile to play.
(LONG) count	Count of elements in options array.
(PLONG) options	Pointer to options array.
(PLONG) segCount	Reserved.
(LONG) count2	Size of description.
(PSZ) description	Pointer to description.

This function plays a MetaFile into a presentation space.

return=**GpiPointArc** (hps, points);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) points	Pointer to end and intermediate points.

This function draws an arc using the current arc parameters through three points starting at the current position.

return=**GpiPolyFillet** (hps, count, points);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of points in array.
(PPOINTL) points	Pointer to points array.

This function draws a curve starting at the current position and the specified points.

return=**GpiPolyFilletSharp** (hps, count, points, sharpness);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of points in arrays.

(PPOINTL) points	Pointer to points array.
(PFIXED) sharpness	Pointer to sharpness values.

This function draws a fillet on a series of lines starting from the current position and supplied end points.

return=**GpiPolygons** (hps, count, polygons, options, model);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of polygons in array.
(PPOLYGON) polygons	Pointer to polygons array.
(LONG) options	Drawing options.
(LONG) model	Drawing model.

This function draws a set of closed polygons.

return=**GpiPolyLine** (hps, count, points);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of points in array.
(PPOINTL) points	Pointer to points array.

This function draws a series of lines starting at the current position and the specified end points.

return=**GpiPolyLineDisjoint** (hps, count, points);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of points in array.
(PPOINTL) points	Pointer to points array.

This function draws a series of disjoint lines starting at the current position and the specified end points.

return=**GpiPolyMarker** (hps, count, points);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of points in array.
(PPOINTL) points	Pointer to points array.

This function draws a series of markers at the specified points.

return=**GpiPop** (hps, count);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of attributes to be restored.

This function restores primitive attributes that have been saved on the stack.

return=**GpiPtInRegion** (hps, region, point);

(LONG) return	Inside or error indicator.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle of region.
(PPOINTL) point	Pointer to point to test.

This function tests for a point to be in a specified region.

return=**GpiPtVisible** (hps, point);

(LONG) return	Visibility or error indicator.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to point to test.

This function tests for a point to be visible within the clipping region of the device for the specified presentation space.

return=**GpiPutData** (hps, format, length, buffer);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) format	Coordinate type used.
(LONG) length	Length of data buffer.
(PBYTE) buffer	Pointer to data buffer.

This function puts graphic order data in the current segment.

return=**GpiQueryArcParams** (hps, arcParams);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PARCPARAMS) arcParams	Pointer to arc parameters.

This function returns the current arc parameters.

return=**GpiQueryAttrMode** (hps);

(LONG) return	Current attribute mode or indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function returns the current attribute mode.

return=**GpiQueryAttrs** (hps, type, mask, attributes);

(LONG) return	Current attribute mode or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Primitive type.
(ULONG) mask	Attribute mask.
(PBUNDLE) attributes	Pointer to attribute bundle.

This function returns the current attributes for a specified primitive type.

return=**GpiQueryBackColor** (hps);

(LONG) return	Background color.
(HPS) hps	Handle to presentation space.

This function returns the current background color.

return=**GpiQueryBitmapBits** (hps, start, count, buffer, infoTable);

(LONG) return	Scan lines returned or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) start	Starting scan line number.
(LONG) count	Number of scan lines to query.
(PBYTE) buffer	Buffer to receive bitmap data.
(PBITMAPINFO2) infoTable	Pointer to table that defines the format of the bitmap data.

This function transfers data from a bitmap to a specified buffer.

return=**GpiQueryBitmapDimension** (bitmap, dimension);

(BOOL) return	Indicator for success or failure.
(HBM) bitmap	Handle to bitmap.
(PSIZEL) dimension	Pointer to bitmap dimension.

This function retrieves the width and height of a specified bitmap.

return=**GpiQueryBitmapHandle** (hps, lcid);

(HBM) return	Handle to bitmap or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) lcid	Local identifier for bitmap.

This function returns the handle of the bitmap currently associated with the local identifier.

return=**GpiQueryBitmapInfoHeader** (bitmap, header);

(BOOL) return	Indicator for success or failure.
(HBITMAP) bitmap	Handle to bitmap to query.

(PBITMAPINFOHEADER2) header
Pointer to bitmap information header.
This function returns information about a specified bitmap.

return=**GpiQueryBitmapParameters** (bitmap, parameters);
(BOOL) return — Indicator for success or failure.
(HBITMAP) bitmap — Handle to bitmap to query.
(PBITMAPINFOHEADER) parameters
Pointer to bitmap information header.
This function returns parameters for a specified bitmap.

return=**GpiQueryBoundaryData** (hps, boundaryData);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PRECTL) boundaryData — Pointer boundary data.
This function returns boundary data.

return=**GpiQueryCharAngle** (hps, angle);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PGRADIENTL) angle — Pointer baseline angle.
This function returns the current character baseline angle.

return=**GpiQueryCharBox** (hps, boxSize);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PSIZEF) boxSize — Pointer to character box size.
This function returns the current character box attribute.

return=**GpiQueryCharBreakExtra** (hps, breakExtra);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PFIXED) breakExtra — Pointer to character break extra attribute value.
This function returns the current character break extra attribute.

return=**GpiQueryCharDirection** (hps);
(LONG) return — Character direction or error indicator.
(HPS) hps — Handle to presentation space.
This function returns the current character direction attribute.

return=**GpiQueryCharExtra** (hps, breakExtra);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PFIXED) extra	Pointer to character extra attribute value.

This function returns the current character extra attribute.

return=**GpiQueryCharMode** (hps);

(LONG) return	Character mode or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current character mode attribute.

return=**GpiQueryCharSet** (hps);

(LONG) return	Character set local identifier.
(HPS) hps	Handle to presentation space.

This function returns the current local identifier for the current font.

return=**GpiQueryCharShear** (hps, shear);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) shear	Pointer to shear vector point.

This function indicates the current character shear angle by returning a vector point.

return=**GpiQueryCharStringPos** (hps, options, length, string, incs, points);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) length	Length of string.
(PCH) string	Pointer to character string.
(PLONG) incs	Pointer to horizontal increment values.
(PPOINTL) points	Array of positions for character placement.

This function returns the positions at which a specified character string's characters would be placed if drawn.

return=**GpiQueryCharStringPosAt** (hps, start, options, length, string, incs, points);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) start	Starting position.
(ULONG) options	Options.
(LONG) length	Length of string.
(PCH) string	Pointer to character string.
(PLONG) incs	Pointer to horizontal increment values.
(PPOINTL) points	Array of positions for character placement.

This function returns the positions at which a specified character string's characters would be placed if drawn.

return=**GpiQueryClipBox** (hps, rectangle);

(LONG) return	Complexity of resulting region and indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to bounding rectangle.

This function returns the smallest rectangle possible that would include all the intersections of all the clipping definitions.

return=**GpiQueryClipRegion** (hps);

(HRGN) return	Handle to region or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the handle to the currently selected clip region.

return=**GpiQueryColor** (hps);

(LONG) return	Color attribute or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current color attribute value.

return=**GpiQueryColorData** (hps, count, colorInfo);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count	Number of elements in array.
(PLONG) colorInfo	Pointer to array to hold color information.

This function returns information about the current logical color table or the selected palette.

return=**GpiQueryColorIndex** (hps, options, rgb);

(LONG) return	Color index or error indicator.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) rgb	RGB color value.

This function returns the color index of the device color which is the closest match to the specified RGB color for the device associated with the presentation space.

return=**GpiQueryCp** (hps);

(ULONG) return	Code page or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the currently selected graphics code page.

return=**GpiQueryCurrentPosition** (hps, point);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to current position value.

This function returns the current position value.

return=**GpiQueryDefArcParams** (hps, params);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PARCPARAMS) params	Pointer to arc parameters.

This function returns the default arc parameter values.

return=**GpiQueryDefAttrs** (hps, type, mask, attributes);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Primitive type.
(ULONG) mask	Attribute mask.
(PBUNDLE) attributes	Pointer to attribute bundle.

This function returns the default attribute values for a specified primitive type.

return=**GpiQueryDefaultViewMatrix** (hps, elements, matrix);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

(LONG) elements — Number of matrix elements.
(PMATRIXLF) matrix — Pointer to transformation matrix.
This function returns the current default viewing transformaton matrix.

return=**GpiQueryDefCharBox** (hps, size);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PSIZEL) size — Pointer to character box size.
This function returns the size of the default graphics character box.

return=**GpiQueryDefTag** (hps, tag);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PLONG) tag — Pointer to tag identifier value.
This function returns the default value of the tag identifier.

return=**GpiQueryDefViewingLimits** (hps, limits);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PRECTL) limits — Pointer to default viewing limits.
This function returns the default value of the viewing limits.

return=**GpiQueryDevice** (hps);
(HDC) return — Handle to device context or error indicator.
(HPS) hps — Handle to presentation space.
This function returns the handle of the device context currently associated with the presentation space.

return=**GpiQueryDeviceBitmapFormats** (hps, elements, dataArray);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(LONG) elements — Number of elements in the data array.
(PLONG) dataArray — Pointer to the data array.
This function returns the supported formats of the bitmaps for the associated device context.

return=**GpiQueryDrawControl** (hps, control);
(LONG) return — Value of control or error indicator.
(HPS) hps — Handle to presentation space.
(LONG) control — Control.
This function returns the value for a specified drawing control.

return=**GpiQueryDrawingMode** (hps);

(LONG) return	Drawing mode or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current drawing mode.

return=**GpiQueryEditMode** (hps);

(LONG) return	Editing mode or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current editing mode.

return=**GpiQueryElement** (hps, offset, length, buffer);

(LONG) return	Number of bytes returned or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) offset	Starting byte offset into element.
(LONG) length	Maximum length of data that can be returned.
(PBYTE) buffer	Pointer to data buffer.

This function returns element data from the element being pointed to by the element pointer.

return=**GpiQueryElementPointer** (hps);

(LONG) return	Current element pointer value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current element pointer value.

return=**GpiQueryElementType** (hps, type, length, buffer);

(LONG) return	Number of bytes returned or error indicator.
(HPS) hps	Handle to presentation space.
(PLONG) type	Type of element.
(LONG) length	Maximum length of data that can be returned.
(PSZ) buffer	Pointer to data buffer to hold element description.

This function returns information about the element being pointed to by the element pointer.

return=**GpiQueryFaceString** (hps, family, faceAttrs, length, faceName);

(LONG) return	Length of compound face name or error indicator.
(HPS) hps	Handle to presentation space.
(PSZ) family	Pointer to font family name.
(PFACENAMEDESC) faceAttrs	Pointer to face name attributes.
(LONG) length	Maximum length of data that can be returned in faceName buffer.
(PSZ) faceName	Pointer to face name data buffer.

This function returns a compound face name for a font.

return=**GpiQueryFontAction** (hab, options);

(ULONG) return	Action or error indicator.
(HAB) hab	Handle to anchor-block.
(ULONG) options	Options.

This function returns if available fonts have been affected since the last time this function was used.

return=**GpiQueryFontFileDescription** (hab, fileName, count, fontDescs);

(LONG) return	Number of fonts for which details were not returned or error indicator.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Filename for font resource.
(PLONG) count	Pointer to the maximum number of family and face name pairs to be returned.
(PFFDESC) fontDescs	Pointer to array to put family and face name data.

This function returns all the family and face name information for a specified font resource file.

return=**GpiQueryFonts** (hps, options, faceName, fontCount, length, metrics);

(LONG) return	Number of fonts not returned or error indicator.
(HPS) hps	Handle to presentation space.
(ULONG) options	Enumeration options.
(PSZ) faceName	Face name of fonts to query.

(PLONG) fontCount	Pointer to number of fonts that application requires and on return, number of fonts returned.
(LONG) length	Length of each font metrics record.
(PFONTMETRICS) metrics	Pointer to font metric records.

This function returns font metrics for fonts of the specified face name.

return=**GpiQueryFullFontFileDescs** (habs, fileName, count, names, length);

(LONG) return	Number of fonts for which details were not returned or error indicator.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Pointer to font resource filename.
(PLONG) count	Pointer to number of fonts that application requires and on return, number of fonts returned.
(PVOID) names	Pointer to buffer where font family and face name pairs are returned.
(PLONG) length	Length of names buffer and on return, actual length needed to hold all family and face names.

This function returns family and face name information for fonts in a specified font resource file.

return=**GpiQueryGraphicsField** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to graphics field.

This function returns the graphics field rectangle in presentation page units.

return=**GpiQueryInitialSegmentAttrs** (hps, attribute);

(LONG) return	Current initial attribute value or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) attribute	Attribute to query.

This function returns the initial segment attribute value for a specified attribute.

return=**GpiQueryKerningPairs** (hps, elements, pairs);

(LONG) return	Number of kerning pairs returned or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) elements	Number of elements in pairs array.
(PKERNINGPAIRS) pairs	Pointer to array of kerning pairs.

This function returns kerning pair data for the current logical font.

return=**GpiQueryLineEnd** (hps);

(LONG) return	Line-end attribute value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current line-end attribute value.

return=**GpiQueryLineJoin** (hps);

(LONG) return	Line-join attribute value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current line-join attribute value.

return=**GpiQueryLineType** (hps);

(LONG) return	Line-type attribute value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current line-type attribute value.

return=**GpiQueryLineWidth** (hps);

(FIXED) return	Cosmetic line-width attribute value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current cosmetic line-width attribute value.

return=**GpiQueryLineWidthGeom** (hps);

(LONG) return	Geometric line-width attribute value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current geometric line-width attribute value.

return=**GpiQueryLogColorTable** (hps, options, start, elements, array);

(LONG) return	Number of elements returned or error indicator.
(HPS) hps	Handle to presentation space.

(ULONG) options — Options.
(LONG) start — Starting index for data to be returned.
(LONG) elements — Number of elements in array.
(PLONG) array — Pointer to color array.
This function returns the logical color table.

return=**GpiQueryLogicalFont** (hps, lcid, name, attributes, length);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(LONG) lcid — Local identifier.
(PSTR8) name — Pointer to logical font name.
(PFATTRS) attributes — Pointer to attributes of font.
(LONG) length — Maximum length of data that can be returned in attributes buffer.
This function returns information about a logical font.

return=**GpiQueryMarker** (hps);
(LONG) return — Marker symbol or error indicator.
(HPS) hps — Handle to presentation space.
This function returns the value of current marker symbol attribute.

return=**GpiQueryMarkerBox** (hps, size);
(BOOL) return — Indicator for success or failure.
(HPS) hps — Handle to presentation space.
(PSIZEF) size — Pointer to size of marker box.
This function returns the value of current marker-box attribute.

return=**GpiQueryMarkerSet** (hps);
(LONG) return — Marker-set local identifier or error indicator.
(HPS) hps — Handle to presentation space.
This function returns the current value of the marker-set attribute.

return=**GpiQueryMetaFileBits** (metaFile, offset, length, data);
(BOOL) return — Indicator for success or failure.
(HMF) metaFile — Handle to MetaFile.
(LONG) offset — Offset into MetaFile to start transfer.
(LONG) length — Length of MetaFile data to transfer.
(PBYTE) data — Pointer to data buffer.
This function transfer MetaFile data to a application data buffer.

return=**GpiQueryMetaFileLength** (metaFile);

(LONG) return	Length of MetaFile data or error indicator.
(HMF) metaFile	Handle to MetaFile.

This function returns the length of data in a specified MetaFile.

return=**GpiQueryMix** (hps);

(LONG) return	Value of foreground color-mixing mode or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current value of the foreground color-mixing mode.

return=**GpiQueryModelTransformMatrix** (hps, elements, matrix);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.

This function returns the current model transformation matrix.

return=**GpiQueryNearestColor** (hps, options, rgb);

(LONG) return	Nearest color or error indicator.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) rgb	Required RGB color.

This function returns the nearest color available on the currently associated device with a specified color.

return=**GpiQueryNumberSetIds** (hps);

(LONG) return	Number of used local identifiers or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the number of local identifiers in use.

return=**GpiQueryPageViewport** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to viewport rectangle.

This function returns the page viewport.

return=**GpiQueryPalette** (hps);

(HPAL) return	Handle to palette or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the handle of the currently selected palette.

return=**GpiQueryPaletteInfo** (palette, hps, options, start, elements, array);

(LONG) return	Number of elements returned or error indicator.
(HPAL) palette	Handle to palette.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) start	Starting index from which to retrieve data.
(LONG) elements	Number of elements to retrieve.
(PLONG) array	Pointer to array for data.

This function returns information for a palette.

return=**GpiQueryPattern** (hps);

(LONG) return	Pattern symbol value or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the current value for the pattern symbol.

return=**GpiQueryPatternRefPoint** (hps, point);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to pattern reference point.

This function returns the current pattern reference point.

return=**GpiQueryPatternSet** (hps);

(LONG) return	Pattern-set local identifier or error indicator.
(HPS) hps	Handle to presentation space.

This function returns the pattern-set local identifier.

return=**GpiQueryPel** (hps, point);

(LONG) return	Color index of pel or error indicator.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to pel.

This function returns the color of a pel at a position specified by a point in world coordinates.

return=**GpiQueryPickAperturePosition** (hps, point);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to pick aperture position.

This function returns the position of the center of the pick aperture.

return=**GpiQueryPickApertureSize** (hps, size);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PSIZEL) size	Pointer to pick aperture size.

This function returns the size of the pick aperture.

return=**GpiQueryPS** (hps, size);

(LONG) return	Presentation space option or error indicator.
(HPS) hps	Handle to presentation space.
(PSIZEL) size	Pointer to size of presentation page.

This function returns information about a presentation space.

return=**GpiQueryRealColors** (hps, options, start, elements, array);

(LONG) return	Number of elements returned or error indicator.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) start	Ordinal number of first color required.
(LONG) elements	Maximum number of elements to return.
(PLONG) array	Pointer to array for RGB values.

This function returns the RGB values of distinct colors available on the currently associated device.

return=**GpiQueryRegionBox** (hps, region, rectangle);

(LONG) return	Complexity of resulting region and indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region.
(PRECTL) rectangle	Pointer to bounding rectangle.

This function returns the bounding rectangle of a specified region.

return=**GpiQueryRegionRects** (hps, region, rectangle, control, array);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region.
(PRECTL) rectangle	Pointer to bounding rectangle.
(PRGNRECT) control	Pointer to processing control structure.
(PRECTL) array	Pointer to array of rectangles.

This function returns the rectangles that define a specified region.

return=**GpiQueryRGBColor** (hps, options, index);

(LONG) return	RGB value or error indicator.
(HPS) hps	Handle to presentation space.
(ULONG) options	Options.
(LONG) index	Color index.

This function returns the RGB value that results from a specified index on the currently associated device.

return=**GpiQuerySegmentAttrs** (hps, segment, attribute);

(LONG) return	Current attribute value or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier.
(LONG) attribute	Attribute to query.

This function returns the current attribute value for a specified graphic segment and attribute.

return=**GpiQuerySegmentNames** (hps, first, last, maximum, array);

(LONG) return	Number of identifiers returned or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) first	First segment identifier in a range.
(LONG) last	Last segment identifier in a range.
(LONG) maximum	Maximum number of Identifier to be returned.
(PLONG) array	Pointer to array of segment identifiers.

This function returns the segment identifers within a specified range.

return=**GpiQuerySegmentPriority** (hps, reference, order);

(LONG) return	Segment identifier or error indicator.
(HPS) hps	Handle to presentation space.
(LONG) reference	Reference segment identifier.
(LONG) order	Order indicating which segment to return.

This function returns the segment identifer of a graphic segment that is immediately before or after a specified reference segment.

return=**GpiQuerySegmentTransformMatrix** (hps, segment, elements, matrix);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.

This function returns the segment transformation matrix for a specified segment identifier.

return=**GpiQuerySetIds** (hps, maximum, types, names, lcids);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) maximum	Maximum number of objects to be queried.
(PLONG) types	Pointer to array of object types returned.
(PSTR8) names	Pointer to array of font names returned.
(PLONG) lcids	Pointer to array of local identifiers returned.

This function returns information about logical fonts and tagged bitmaps.

return=**GpiQueryStopDraw** (hps);

(LONG) return	Stop or error indicator.
(HPS) hps	Handle to presentation space.

This function returns whether a stop condition currently exists.

return=**GpiQueryTag** (hps, tag);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PLONG) tag	Pointer to tag identifier.

This function returns the current value of the tag identifier.

return=**GpiQueryTextAlignment** (hps, xAlignment, yAlignment);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PLONG) xAlignment	Pointer to horizontal alignment.
(PLONG) yAlignment	Pointer to vertical alignment.

This function returns the current values of the text alignment attribute.

return=**GpiQueryTextBox** (hps, count1, string, count2, points);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) count1	Number of characters.
(PCH) string	Pointer to character string.
(LONG) count2	Number of points in points array.
(PPOINTL) points	Pointer to points array.

This function returns the relative coordinates of the corners of a text box for a specified string.

return=**GpiQueryViewingLimits** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to viewing limits.

This function returns the current value of the viewing limits.

return=**GpiQueryViewingTransformMatrix** (hps, elements, matrix);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.

This function returns the viewing transformation matrix.

return=**GpiQueryWidthsTable** (hps, firstChar, elements, array);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

(LONG) firstChar	Code point of first character width to return.
(LONG) elements	Number of elements in array.
(PLONG) array	Pointer to array of width values.

This function returns character width information for the current font.

return=**GpiRectInRegion** (hps, region, rectangle);

(LONG) return	Inside or error indicator.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region .
(PRECTL) rectangle	Pointer to rectangle.

This function indicates whether any part of a rectangle is within a specified region.

return=**GpiRectVisible** (hps, region, rectangle);

(LONG) return	Visibility or error indicator.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to rectangle.

This function indicates whether any part of a rectangle is within the clipping region of the associated device.

return=**GpiRemoveDynamics** (hps, first, last);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) first	First segment identifier in a range.
(LONG) last	Last segment identifier in a range.

This function removes those parts of the graphic image that are drawn from dynamic segments in a specifed section of the segment chain.

return=**GpiResetBoundaryData** (hps);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

This function resets boundary data to null.

return=**GpiResetPS** (hps, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) options	Reset options.

This function resets the presentation space.

return=**GpiRestorePS** (hps, identifier);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) identifier	Identifier of saved presentation space.

This function restores the state of a previously saved presentation space.

return=**GpiRotate** (hps, matrix, options, angle, center);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PMATRIXLF) matrix	Pointer to transform matrix.
(LONG) options	Transform options.
(FIXED) angle	Rotation angle.
(PPOINTL) center	Pointer to center of rotation.

This function applies a rotation to a transform matrix.

return=**GpiSaveMetaFile** (metaFile, fileName);

(BOOL) return	Indicator for success or failure.
(HMF) metaFile	Handle to MetaFile.
(PSZ) fileName	Pointer to filename.

This function saves a MetaFile to a disk file.

return=**GpiSavePS** (hps);

(LONG) return	Identifier or error indicator.
(HPS) hps	Handle to presentation space.

This function save the state of the presentation space so it can be restored at a later time.

return=**GpiScale** (hps, matrix, options, scale, center);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PMATRIXLF) matrix	Pointer to transform matrix.
(LONG) options	Transform options.
(FIXED) scale	Scale factor.
(PPOINTL) center	Pointer to center of scale.

This function applies scaling to a transform matrix.

return=**GpiSelectPalette** (hps, palette);

(HPAL) return	Handle to old palette.
(HPS) hps	Handle to presentation space.
(HPAL) palette	Handle to palette.

This function selects a palette to a presentation space.

return=**GpiSetArcParams** (hps, parameters);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PARCPARAMS) parameters	Pointer to arc parameters.

This function sets the current arc parameters.

return=**GpiSetAttrMode** (hps, mode);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) mode	Attribute mode.

This function sets the current attribute mode.

return=**GpiSetAttrs** (hps, type, mask, defaultsMask, attributes);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Primitive type to set.
(ULONG) mask	Attributes mask.
(ULONG) defaultsMask	Defaults mask which indicates that the attributes should be set to default values if the corresponding mask flag is also set.
(PBUNDLE) attributes	Pointer to attribute values to be set.

This function sets attributes for a specified primitive type.

return=**GpiSetBackColor** (hps, color);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) color	Color.

This function sets the current background color index for each primitive type to a specified value.

return=**GpiSetBackMix** (hps, mix);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) mix	Background mix mode.

This function sets the current background mix mode for each primitive type.

return=**GpiSetBitmap** (hps, bitmap);

(HBITMAP) return	Handle to old bitmap or error indicator.

(HPS) hps — Handle to presentation space.
(HBITMAP) bitmap — Handle of bitmap.
This function sets a bitmap as the currently selected bitmap in a memory device context.

return=**GpiSetBitmapBits** (hps, startLine, scans, buffer, infoTable);

(LONG) return	Scan lines set or error indicator.
(LONG) startLine	Scan line number at which data transfer is to start.
(LONG) scans	Number of scan lines to transmit.
(PBYTE) buffer	Pointer to application storage buffer.
(PBITMAPINFO2) infoTable	Pointer to bitmap information table.

This function sets bitmap data from application storage to a bitmap.

return=**GpiSetBitmapDimension** (bitmap, size);

(BOOL) return	Indicator for success or failure.
(HBITMAP) bitmap	Handle to bitmap.
(PSIZEL) size	Pointer to size.

This function sets the height and width of a specifed bitmap.

return=**GpiSetBitmapId** (hps, bitmap, lcid);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HBITMAP) bitmap	Handle to bitmap.
(LONG) lcid	Local identifier.

This function tags a bitmap with a local identifier so it can be used as a pattern set.

return=**GpiSetCharAngle** (hps, angle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PGRADIENTL) angle	Pointer to baseline angle.

This function specifies the angle of the baseline for text strings.

return=**GpiSetCharBox** (hps, size);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PSIZEF) size	Pointer to box size.

This function sets the character box attribute to a specified value.

return=**GpiSetCharBreakExtra** (hps, breakExtra);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(FIXED) breakExtra	Character break extra value.

This function specifies an extra increment to be used for space break characters in a text string.

return=**GpiSetCharDirection** (hps, direction);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) direction	Character direction.

This function specifies the direction in which characters are drawn.

return=**GpiSetCharExtra** (hps, extra);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(FIXED) extra	Character extra value.

This function specifies an extra increment to be used for spacing characters in a text string.

return=**GpiSetCharMode** (hps, mode);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) mode	Character mode.

This function specifies the character mode to be used when drawing a string.

return=**GpiSetCharSet** (hps, lcid);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) lcid	Character set local identifier.

This function sets the current value of the character set attribute.

return=**GpiSetCharShear** (hps, angle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) angle	Pointer to a point used to define a vector that defines the shear angle.

This function sets the character shear attribute.

return=**GpiSetClipPath** (hps, path, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) path	Path control flag.
(LONG) options	Options.

This function sets a path as the current clip path.

return=**GpiSetClipRegion** (hps, region, oldRegion);

(LONG) return	Compexity of clipping or error indicator.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region.
(PHRGN) oldRegion	Pointer to handle of old region.

This function sets the region to be used for clipping.

return=**GpiSetColor** (hps, color);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) color	Color.

This function sets the current value of color attribute for each of the individual primitive types.

return=**GpiSetCp** (hps, codePage);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(ULONG) codePage	Code page identifier.

This function sets the default graphic code page.

return=**GpiSetCurrentPosition** (hps, point);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to current position point.

This function sets the current position to a specfied point.

return=**GpiSetDefAttrs** (hps, type, mask, defaults);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) type	Primitive type.
(ULONG) mask	Attribute mask.
(PBUNDLE) attributes	Pointer to attribute bundle.

This function sets the default values to attributes for the specfied primitive type.

return=**GpiSetDefaultViewMatrix** (hps, elements, matrix, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.
(LONG) options	Options.

This function sets the default viewing transformation matrix.

return=**GpiSetDefTag** (hps, tag);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) tag	Default tag identifier.

This function sets the default value of the primitive tag.

return=**GpiSetDefViewingLimits** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to default viewing limit rectangle.

This function sets the default value of the viewing limits.

return=**GpiSetDrawControl** (hps, control, value);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) control	Draw control.
(LONG) value	Value of the drawing control.

This function sets options for subsequent drawing operations.

return=**GpiSetDrawingMode** (hps, mode);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) mode	Draw mode.

This function sets the drawing mode to control the processing of subsequent drawing operations.

return=**GpiSetEditMode** (hps, mode);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) mode	Edit mode.

This function sets the current editing mode.

return=**GpiSetElementPointer** (hps, element);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) element	Element number.

This function sets the element pointer within the current segment to the specified element number.

return=**GpiSetElementPointerAtLabel** (hps, label);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) label	Label.

This function sets the element pointer within the current segment to the element containing the specified label.

return=**GpiSetGraphicsField** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to graphics field rectangle.

This function sets the size and position of the graphics field.

return=**GpiSetInitialSegmentAttrs** (hps, attribute, value);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) attribute	Segment attribute.
(LONG) value	Value.

This function specifies a segment attribute to use when a new segment is created.

return=**GpiSetLineEnd** (hps, lineEnd);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) lineEnd	Style of line-end.

This function sets the current line-end attribute.

return=**GpiSetLineJoin** (hps, lineJoin);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) lineJoin	Style of line join.

This function sets the current line-join attribute.

return=**GpiSetLineType** (hps, lineType);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) lineType	Style of line type.

This function sets the current cosmetic line-type attribute.

return=**GpiSetLineWidth** (hps, width);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(FIXED) width	Line-width multiplier.

This function sets the current cosmetic line-width attribute.

return=**GpiSetLineWidthGeom** (hps, width);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) width	Line width.

This function sets the current geometric line-width attribute.

return=**GpiSetMarker** (hps, marker);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) marker	Marker symbol.

This function sets the value of the marker symbol attribute.

return=**GpiSetMarkerBox** (hps, size);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PSIZEF) size	Pointer to the size of the marker box.

This function sets the current marker-box attribute.

return=**GpiSetMarkerSet** (hps, lcid);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) lcid	Marker-set local identifier.

This function sets the current marker-set attribute.

return=**GpiSetMetaFileBits** (metaFile, offset, length, buffer);

(BOOL) return	Indicator for success or failure.
(HMF) metaFile	Handle to memory MetaFile.
(LONG) offset	Offset from where transfer will start.
(LONG) length	Length of MetaFile data to transfer.
(PBYTE) buffer	Pointer to the application memory buffer.

This function transfers MetaFile data from application memory to a memory MetaFile.

return=**GpiSetMix** (hps, mix);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) mix	Mix mode.

This function sets the current foreground mix attribute for each individual primitive type.

return=**GpiSetModelTransformMatrix** (hps, elements, matrix, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.
(LONG) options	Options.

This function sets the model transformation matrix.

return=**GpiSetPageViewport** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to page viewport.

This function sets the page viewport within the device space.

return=**GpiSetPaletteEntries** (palette, format, start, elements, table);

(BOOL) return	Indicator for success or failure.
(HPAL) palette	Handle to palette.
(ULONG) format	Format of entries in table.
(ULONG) start	Starting index.
(ULONG) elements	Number of elements in table.
(PULONG) table	Pointer to application table.

This function sets the entries in a palette.

return=**GpiSetPattern** (hps, pattern);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) pattern	Pattern symbol.

This function sets the current value of the pattern symbol attribute.

return=**GpiSetPatternRefPoint** (hps, refPoint);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) refPoint	Pointer to pattern reference point.

This function sets the current pattern reference point.

return=**GpiSetPatternSet** (hps, lcid);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) lcid	Pattern set local identifier.

This function sets the current value of the pattern set attribute.

return=**GpiSetPel** (hps, point);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to point to set pel.

This function sets a pel at a specified point.

return=**GpiSetPickAperturePosition** (hps, point);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PPOINTL) point	Pointer to center point.

This function sets the center of the pick aperture in the presentation page for correlation operations.

return=**GpiSetPickApertureSize** (hps, options, size);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) options	Options.
(PSIZEL) size	Pointer to pick aperture size.

This function sets the pick aperture size.

return=**GpiSetPS** (hps, size, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.

(PSIZEL) size	Pointer to the size of the presentation page.
(LONG) options	Options.

This function sets the presentation page size, units, and format.

return=**GpiSetRegion** (hps, region, count, rectangles);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(HRGN) region	Handle to region.
(LONG) count	Number of rectangles.
(PRECTL) rectangles	Pointer to array of rectangles.

This function sets the region to the union of a set of specified rectangles.

return=**GpiSetSegmentAttrs** (hps, segment, attribute, values);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier.
(LONG) attribute	Segment attribute.
(LONG) value	Attribute value.

This function sets a segment attribute to a specified value.

return=**GpiSetSegmentPriority** (hps, segment, refSegment, order);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier.
(LONG) refSegment	Reference segment identifier to place segment before or after.
(LONG) order	Position higher or lower.

This function sets the position of a segment within the segment chain.

return=**GpiSetSegmentTransformMatrix** (hps, segment, elements, matrix, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) segment	Segment identifier.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.
(LONG) options	Options.

This function sets the segment transformation matrix.

return=**GpiSetStopDraw** (hps, value);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) value	Stop draw condition.

This function sets or clears the stop draw condition.

return=**GpiSetTag** (hps, tag);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) tag	Tag identifier.

This function specifies a tag by which the following primitives are to be associated.

return=**GpiSetTextAlignment** (hps, horizontal, vertical);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) horizontal	Horizontal alignment.
(LONG) vertical	Vertical alignment.

This function sets the alignment used to position characters in a string.

return=**GpiSetViewingLimits** (hps, rectangle);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PRECTL) rectangle	Pointer to rectangle.

This function sets a clipping rectangle in model space.

return=**GpiSetViewingTransformMatrix** (hps, elements, matrix, options);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) elements	Number of elements in transformation matrix.
(PMATRIXLF) matrix	Pointer to transformation matrix.
(LONG) options	Options.

This function sets the viewing transformation matrix.

return=**GpiStrokePath** (hps, path, options);

(LONG) return	Number of correlation hits or indicator for success or failure.

(HPS) hps	Handle to presentation space.
(LONG) path	Path identifier to be stroked.
(LONG)options	Options.

This function strokes a path and then draws it.

return=**GpiTranslate** (hps, path, options);

(LONG) return	Number of correlation hits or indicator for success or failure.
(HPS) hps	Handle to presentation space.
(LONG) path	Path identifier to be stroked.
(LONG) options	Options.

This function strokes a path and then draws it.

return=**GpiTranslate** (hps, matrix, options, translation);

(BOOL) return	Indicator for success or failure.
(HPS) hps	Handle to presentation space.
(PMATRIXLF) matrix	Pointer to transform matrix.
(LONG) options	Transform options.
(PPOINTL) translation	Pointer to translation point.

This function applies a translation to a transform matrix.

return=**GpiUnloadFonts** (hab, fileName);

(BOOL) return	Indicator for success or failure.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Pointer to filename that contains font.

This function unloads any fonts previously loaded from a specified resource file.

return=**GpiUnloadPublicFonts** (hab, fileName);

(BOOL) return	Indicator for success or failure.
(HAB) hab	Handle to anchor-block.
(PSZ) fileName	Pointer to filename that contains font.

This function unloads any public fonts previously loaded from a specified resource file.

return=**GpiWCBitBlt** (target, source, pointCount, pointArray, rop, options);

(LONG) return	Number of correlation hits or indicatorfor success or failure.
(HPS) target	Target presentation space handle.
(HPS) source	Source presentation space handle.

(LONG) pointCount	Number of points in point array.
(PPOINTL) pointArray	Pointer to point array.
(LONG) rop	Raster operation mix function.
(ULONG) options	Options.

This function copies bitmap image data from one rectangular area to another.

APPENDIX B
Working with the Diskette

The software that comes on the diskette is compressed and, when installed on your system, will take up slightly less than 2MB of disk storage. An install program is provided to make installation simple. Before installing the software on your machine, you should make a backup copy of the original diskette.

To run the installation program, simply insert the install diskette into the A: drive and enter the command A:INSTALL at an OS/2 or DOS command prompt. If your 3.5 inch diskette is not the A: drive then substitute A: with the appropriate drive letter.

IMPORTANT!!! Please read the following paragraphs BEFORE actually installing the software. Especially if you:

- Plan to alter the drive and/or directory that the software is installed into.
- Do not use the IBM compiler, toolkit, and WorkFrame tools.
- Did not install your WorkFrame tool into the C:\IBMWF directory.

The install program makes several assumptions about the configuration of your machine. If these assumptions are valid, then you can accept the default install parameters when prompted. If these assumptions are not valid, you may want to alter the install parameters.

When you run the installation program, it will bring up a menu that allows you to configure the installation. At this point, you can go ahead and run the installation program, but do NOT choose the option *Start Installation* until you understand the following assumptions made by the install program:

1. The install program assumes that you are using the IBM C/C++ compiler, IBM WorkFrame, and IBM Toolkit. The supplied make files and project files are designed for this environment. If you do not use these tools, you can specify any destination path you desire and perform the installation. Then refer to the section "Building the Sample Programs from Scratch" (found later in this appendix) for more details on creating the programs.
2. The install program assumes that the WorkFrame software is installed in the directory C:\IBMWF. The install program copies the WorkFrame project files into this directory so that the WorkFrame software can find them. If you have the WorkFrame software installed in a different directory, select *Edit the destination paths* from the main install menu and set the *Project Files* path to point to the directory in which WorkFrame is located.
3. The install program assumes that you want to install the program into the directory C:\WLYDRAW. The project and make files supplied with this diskette depend on the software being installed at this location. If you choose to install it somewhere else, then you will need to create new project and make files. If you decide to do this, we assume you already know how to create project and make files. See the section "Building the Sample Programs from Scratch" for details on the source files required for each program.

At this point you can go ahead and install the software. If possible, the easiest way to install, run, and build the software is to use the defaults.

RUNNING THE SAMPLE PROGRAMS

The installation program copies prebuilt versions of the programs into the specified source directory (the default library is C:\WLYDRAW). The following programs should be installed into that directory:

`DRAW.EXE` – The graphical drawing application.

`PRINTERS.EXE` – This program shows information about printers on your system.

`VIEWMET.EXE` – This program displays MetaFiles and allows you to examine the orders found inside the MetaFile.

`BROWSE.DLL` – This is a text browsing utility that is called by other programs. It cannot be run directly from the command line.

To run any of these programs simply open an OS/2 command window, change to the directory containing the programs, and enter the name of the program you want to run (i.e., DRAW). Try this out and make sure you can run each of the programs as they have been installed.

REBUILDING THE PROGRAMS USING THE MAKE FILES

There are MAKE files included on the diskette for each of the sample programs. If you have installed the software into the default directory (C:\WLYDRAW), you can use the supplied make files to rebuild the programs. If you have installed the software to a different directory, you will have to rebuild the make files yourself using WorkFrame (see the "Building the Sample Programs from Scratch" section for details).

To rebuild one of the programs open an OS/2 command window and switch to the C:\WLYDRAW directory. Then, invoke NMAKE on the makefile of the desired program (for example: NMAKE DRAWMAKE). This will build the specified program. The make files for the various programs are:

DRAWMAKE – Builds the draw program

PRNTMAKE – Builds the printers program

VIEWMAKE – Builds the metaFile viewer program

BROWMAKE – Builds the Browser DLL

REBUILDING THE PROGRAMS USING THE PROJECT FILES

The install program copies project files for WorkFrame/2 V1.1 into the workframe directory on your system. Make sure you installed the software according to assumptions #2 and #3 as previously described.

Start the WorkFrame/2 V1.1 program (make sure you run V1.1 and not V2.1, as both come with the ToolKit). If the install worked correctly, the following projects should now show up in the WorkFrame Project Control list:

Wiley Draw

Wiley Browse

Wiley View MetaFile

Wiley Printers

To build one of the projects, first open it by selecting it from the Project Control list and pressing the *Open* button (make sure you only have one project selected when you open it because otherwise workframe will only open the first one selected). Once the project is open, choose the *Make* item from the *Actions* menu in the IBM WorkFrame/2 window. This starts the build activity. A dialog should appear that shows the MAKE activity as it progresses. If everything builds okay, you should be able to choose *Run* from the *Actions* menu to execute the application. Note, you won't be able to run the Browse project since it simply builds a .DLL file that is used by the other programs.

To avoid confusion over which project is being built, you might want to close any currently open projects before opening a new one. Finally, remember that MAKE only recompiles and relinks those files that are outdated. If you want to force a complete recompile and link, then choose *Build* rather than *Make* from the *Action* menu.

BUILDING PROGRAMS FROM SCRATCH

This section provides information on how to build the example programs without using the supplied project and make files. This is necessary if you are not using the IBM tool set or if you installed the software into a directory other than C:\WLYDRAW.

The following compile and link options should be specified when building the various programs.

Compile Options:

- Allow use of // style comments (found on the source page of IBM compiler settings)

- Choose Multithread libraries (found on the object page of IBM compiler settings)

Link Options:

- Application type is PM (found on the Generation page of IBM linker settings)
- Definition file should be set for program being built, such as DRAW.DEF (found on the file names page of IBM linker settings)

You may also want to compile and link the programs with debug turned on so that you can step through execution of the programs. The following paragraphs describe what activities are required to build the various program files and what source files are involved.

To build the Browse Utility DLL (BROWSE.DLL), the following build activities are required:

1. Compile C source code files.
2. Link object files.
3. Compile dialogs and other resource files using resource compiler (RC).
4. Bind resources to DLL using resource compiler (RC).
5. Compile help text using IPF compile.

Table B.1 shows which files are required for building the Browse Utility DLL.

TABLE B.1 Building the Browse Utility DLL.

File Name	Description
BROWSE.C, BROWSE.H	Main C source file for BROWSE.DLL.
BROWSE.DLG	BROWSE.DLL dialog boxes definitions.
BRWSDIAG.H	Header file for BROWSE.DLL dialog boxes definitions.
BROWSE.DEF	Definition file for BROWSE.DLL.
BROWSE.RC	Resource source file for BROWSE.DLL resource.
BROWSE.IPF	Help text source for the Browse Utility.
BROWSE.ICO	Browse Utility icon file.

TABLE B.2 Building the Draw program

File Name	Description
DRAW.C, DRAW.H	Main C source file for DRAW.EXE.
BROWSE.H	Interface definitions for Browse Utility.
DRAW.DLG	DRAW.EXE dialog boxes definitions.
DIALOG.H	Header file for DRAW.EXE dialog boxes definitions.
DRAW.DEF	Definition file for DRAW.EXE.
DRAW.RC	Resource source file for DRAW.EXE resource.
DRAW.IPF	Help text source file for the graphics editor.
ATTR.C, ATTR.H	Attribute processing source code for DRAW.EXE.
DRAWPRNT.C, DRAWPRNT.H	Print graphic source code.
EDIT.C, EDIT.H	Edit source code for DRAW.EXE. This includes functions for cut, copy, paste, and update the client area.
FUNCS.C, FUNCS.H	Dialog procedures and miscellaneous functions for DRAW.EXE.
GENERAL.H	Miscellaneous definitions.
GOL.C, GOL.H	List processing functions for DRAW.EXE.
OBJECT.C, OBJECT.H	Graphic object management source code for DRAW.EXE. Includes transformation management for objects.
TOOL.C, TOOL.H	Tool source code for DRAW.EXE. (Processes event data for each tool type in the graphic editor. This code makes each tool polymorphic.)
PARSESEG.C, PARSESEG.H	Parse graphic segment source code.
PORDERS.C, PORDER.H	Parse graphics orders source code.
WRITETIF.C, WRITETIF.H	Write TIFF file source code.
ARC.ICO, BOX.ICO, DRAW.ICO,	Icons resources for DRAW.EXE.

TABLE B.2 (Continued)

File Name	Description
ELLIPSE.ICO, LINE.ICO, PENCIL.ICO, RBOX.ICO, SELECT.ICO, TEXT.ICO, FILLET.ICO, FILLETF.ICO, POLYLINE.ICO, POLYFILL.ICO	
BOX.PTR, CROSS.PTR, ELLIPSE.PTR, PENCIL.PTR, RBOX.PTR, SELECT.PTR, TEXT.PTR	Pointer resources for DRAW.EXE.

To build the Draw program (DRAW.EXE), the following build activities are required:

1. Compile C source code files.
2. Link object files.
3. Compile dialogs and other resource files using resource compiler (RC).
4. Bind resources to DLL using resource compiler (RC).
5. Compile help text using IPF compile.

Table B.2 shows which files are required for building the Draw program. To execute the Draw program, the following run-time files are also required:

DLL Files: BROWSE.DLL
Help Files: BROWSE.HLP

To build the MetaFile Viewer program (VIEWMET.EXE), the following build activities are required;

1. Compile C source code files.
2. Link object files.
3. Compile dialogs and other resource files using resource compiler (RC).
4. Bind resources to DLL using resource compiler (RC).
5. Compile help text using IPF compile.

Table B.3 shows which files are required for building the MetaFile Viewer program.

TABLE B.3 Building the MetaFile Viewer program

File Name	Description
VIEWMET.C, VIEWMET.H	Main C source file for VIEWMET.EXE.
BROWSE.H	Interface definitions for Browse Utility.
VIEWMET.DLG	VIEWMET.EXE dialog boxes definitions.
VMDLG.H	Header file for VIEWMET.EXE dialog boxes definitions.
VIEWMET.DEF	Definition file for VIEWMET.EXE.
VIEWMET.ICO	Application icon for MetaFile Viewer.
VIEWMET.RC	Resource source file for VIEWMET.EXE resource.
VIEWMET.IPF	Help text for MetaFile Viewer.
PARSEMET.C, PARSEMET.H	Parse OS/2 MetaFile source code.
PORDERS.C, PORDER.H	Parse graphics ord0ers source code.
ORDERTAB.C	Source code for graphic orders branch table.

To execute the MetaFileViewer program, the following run-time files are also required:

DLL Files: BROWSE.DLL
Help Files: BROWSE.HLP

To build the Printer Information viewer program (PRINTERS.EXE), the following build activities are required:

1. Compile C source code files.
2. Link object files.

Table B.4 shows which files are required for building the Printers Information program.

TABLE B.4 Building the Printers Information program

File Name	Description
PRINTERS.C	Main C source file for PRINTERS.EXE.
BROWSE.H	Interface definitions for Browse Utility.
PRINTERS.DEF	Definition file for PRINTERS.EXE.

To execute the Printer Information program, the following run-time files are also required:

DLL Files: BROWSE.DLL
Help Files: BROWSE.HLP

We have built the programs using the IBM tools (C/C++ V2.01 ToolKit V2.1, and WorkFrame V1.1 or V2.1). We have also been able to build the programs using the Borland C++ toolset V1.0. The Borland compiler is a C++ compiler and issues a significant number of warnings due to the tightness of the C++ compiler and the large amount of type casting done in the example programs. However, the programs should compile and execute properly using the Borland compiler. Other compilers that support the above options should work as well (as long as they provide compatible OS/2 system header files and run-time libraries).

Index

1Match, 141

Antialiasing, 140
Application icon, 26, 27
Application installation complete, 26
Arc, 59
Arc parameters, 70, 73, 85
Arc tool, 6, 14
Architecture, 33
Arcs, 67
  full, 71
  multi, 67
  partial, 71
  simple, 67
Area background color, 79
Area background mix, 79
Area color, 77
Area fill, 246
Area mix, 79
ASCII, 123
Ascender, 133
Ascenders, 132
Aspect ratio, 42, 133
Association, 27
ATTR_CHAINED, 177, 178
ATTR_DETECTABLE, 177
ATTR_DYNAMIC, 178, 181
ATTR_FASTCHAIN, 177
ATTR_PROP_DETECTABLE, 177
ATTR_PROP_VISIBLE, 178
ATTR_VISIBLE, 178
Atom, 138
Attributes, 15, 21, 22, 35, 52

Back, 17, 179
Background color, 58
Background mix, 58
Backward 1, 17, 179
Baseline, 124, 132
Begin Document, 328, 330
Bezier splines, 73
BITMAPINFO2, 108, 110, 116
BITMAPINFOHEADER2, 108, 110
Bitmap, 36, 102, 104, 105, 108, 111, 114, 115
Boundary, 183
Boundary accumulation, 183, 184, 269, 270
Box radius, 16
Box tool, 5, 14
Break character, 96, 135
Bundle, 52, 53
  area, 53, 253
  image, 53
  line and arc, 53
  marker, 53
  text, 53
Bundles
  area, 67, 70, 78, 82, 85, 86, 88
  line, 67, 69, 88
  marker, 100
  text, 88, 156

C programming language, 19

Capture TIFF, 9
Chain, 275, 278, 281
Chained segment, 181, 183
Char break extra, 96
Char extra, 96
Character angle, 91, 92
Character background color, 90
Character background mix, 90
Character box, 90, 91, 132, 141, 152
Character color, 90
Character direction, 94
Character mix, 90
Character mode, 90
Character set, 90
Character shear, 93
Character text alignment, 94
Clip area
  clip path, 40
  clip region, 43
  graphics field, 42
  viewing limit, 41
Clip path, 259, 262
Clip region, 265–267
Clipping, 40, 120, 243, 248, 257–259, 261, 264, 265, 267, 268
CONFIG.SYS, 27
Code page, 122, 149
Code point, 123, 135, 155
Color, 15, 55
Color plane, 105, 108
Coordinate space, 38–40, 207
  default page, 213, 232–234
  device, 207, 213, 238, 239, 248, 254, 265, 281
  device space, 43, 44
  media space, 44
  model, 168, 211, 224, 232, 233, 264, 270
  model space, 41, 43
  notional font definition, 134, 142
  notional font definition space, 44
  page, 234, 238–240, 265, 270
  page space, 41, 44
  presentation page, 157
  world, 142, 157, 168, 207, 211, 224, 244, 259, 267, 268, 270
  world space, 40, 43
Coordinates, 38
Copy, 4, 11
Correlation, 177, 183, 184, 191, 206, 271–273, 276–278, 281, 282
Cosmetic line, 246
CRGN_AND, 250
CRGN_COPY, 250
CRGN_DIFF, 250
CRGN_OR, 250
CRGN_XOR, 250
Current font, 128, 149, 155
Current model transformation matrix, 225
Current position, 154
Cursor, 158, 167
Cut, 4, 11, 22, 25

DCTL_BOUNDARY, 184, 269
DCTL_CORRELATE, 184
DCTL_DISPLAY, 184
DCTL_DYNAMIC, 184
DCTL_ERASE, 184
DCTL_OFF, 184, 269
DCTL_ON, 184, 269
DEVESC_ENDDOC, 308
DEVESC_STARTDOC, 308
Decipoints, 135
Default Viewing Transform, 212
Default view transform, 234, 237, 240
Default viewing matrix, 325
Default viewing transform, 234
Delete, 4, 11, 22
Descender, 133
Desktop, 26
DevEscape, 308
Device context, 36, 37, 176, 184, 244, 245, 286, 303, 305, 306, 309, 313, 314, 316
Device driver, 37
Device name, 291
Device Transform, 213
Device transform, 238
Device units, 37, 209
Device viewport, 239
Device-specific characteristics, 37
DevOpenDC, 304–306, 313, 316
DevPostDeviceModes, 304
DevQueryCaps, 142, 151
DevQueryHardcopyCaps, 308
DISPLAY, 313
Diskette, 26, 30
Display list, 20, 205, 230
DM_DRAW, 176
DM_DRAWANDRETAIN, 176
DM_RETAIN, 176
DOS, 32, 33
Double-byte code page, 135, 140
DPDM_POSTJOBPROP, 304
DPDM_QUERYJOBPROP, 304
DRAW, 314

DRAWANDRETAIN, 314
Dragging, 22
Draw list, 21
Drawing mode, 176
Dynamic Link Library (DLL), 21, 27
Dynamic segment, 181, 184

EBCDIC, 123
Edit handles, 102
Edit options, 11
Edit tools, 4, 22
  Arc tool, 6
  Box tool, 5
  Ellipse tool, 6
  Fillet Fill tool, 4
  Fillet tool, 6
  Line tool, 5
  Pencil tool, 5
  Polyline Fill tool, 7
  Polyline tool, 7
  Rounded Box tool, 5
  Select tool, 4
  Text tool, 6
Element, 169, 171–174, 185, 190, 191, 195, 200, 276
Element pointer, 172–175, 195
Ellipse tool, 6, 14, 23
Em Square, 132–134, 141
EmHeight, 132
Emphasis, 15
End Document, 330, 336
End document, 286
Exit, 11
Export MetaFile, 10
Extension, 27

Facename, 123, 128, 131, 149, 150
Family name, 130
Fast Correlate/Draw, 12
File actions
  import WLY, 10
  new, 9
  open, 9
  save, 9
  save as, 9
Fill color, 15
Fill pattern, 15, 16, 247
Fill patterns, 119
Filled area, 243
Fillet, 74–76
Fillet Fill tool, 4, 13
Fillet tool, 6, 14
Flip, 221
Flip horizontal, 17
Flip vertical, 17
FONTMETRICS, 129
Font, 15, 24
  outline, 247
Font attributes structure, 139, 141
Font family, 138
Font family names, 123
Font metrics, 128, 149, 151, 157
Font name, 15
Fonts, 122
  bold, 136, 140
  device, 126, 136
  hollow, 136, 140, 143
  image, 126
  italic, 136, 140
  monospaced, 157, 158
  outline, 90, 91, 93, 96, 102, 126, 135, 140, 143, 149, 152, 156
  overstrike, 136, 140
  private, 126
  proportional, 157, 159
  public, 126, 149
  raster, 90, 92, 93, 96, 102, 125, 126, 135
  strike out, 143
  strikeout, 138
  underline, 143
  underscore, 136, 137, 140
  vector, 126
Foreground color, 58
Foreground mix, 58
Forward 1, 17, 179
Front, 17, 179
Function parameters, 30

GEOM_LINE_WIDTH, 244
Geometric line, 243–246
Geometric line width, 60
Global variables, 21
Glyph, 123
GpiBeginArea, 78, 82, 87
GpiBeginElement, 171
GpiBeginPath, 78, 85, 244, 245
GpiBitBlt, 103, 114, 116, 324
GpiBox, 59, 66, 78, 83, 263
GpiCallSegmentMatrix, 171, 225, 226
GpiCharString, 89, 96, 154
GpiCharStringAt, 39, 89, 97, 154
GpiCharStringPos, 89, 98, 99, 154, 155
GpiCharStringPosAt, 89, 99, 154
GpiCloseFigure, 78, 86
GpiCloseSegment, 176

GpiCombineRegion, 249, 250, 256
GpiConvert, 38, 39, 151, 157, 225, 240
GpiConvertWithMatrix, 225, 240
GpiCopyMetaFile, 317
GpiCorrelateChain, 273, 274, 276, 281
GpiCorrelateFrom, 275, 276
GpiCorrelateSegment, 276, 281
GpiCreateBitmap, 103, 108, 111, 119
GpiCreateLogFont, 139, 141, 151, 152
GpiCreateLogicalFont, 90
GpiCreateRegion, 249, 250
GpiDeleteBitmap, 103, 111
GpiDeleteElement, 175
GpiDeleteElementRange, 175
GpiDeleteElementsBetweenLabels, 175
GpiDeleteMetaFile, 316, 324
GpiDestroyRegion, 249, 250
GpiDrawBits, 103, 114, 116
GpiDrawChain, 179, 181, 182, 184, 205, 281, 309, 314, 316
GpiDrawDynamics, 181, 182, 184
GpiDrawFrom, 179, 181, 182, 184, 314
GpiDrawSegment, 179, 181, 182, 184, 314
GpiElement, 171, 184
GpiEndArea, 78, 82
GpiEndElement, 171
GpiEndPath, 78, 85, 244, 245
GpiEqualRegion, 249, 255
GpiExcludeClipRectangle, 259, 266
GpiFillPath, 78, 85, 245, 246, 248
GpiFrameRegion, 253
GpiFullArc, 59, 68, 78
GpiGetBitmapDimensions, 111
GpiGetData, 170
GpiImage, 103, 113, 116
GpiIntersectClipRectangle, 259, 266
GpiLabel, 175
GpiLine, 37, 59, 62
GpiLoadBitmap, 111, 119
GpiLoadMetaFile, 317, 324, 330
GpiMarker, 100, 102
GpiModifyPath, 78, 86, 244–246
GpiMove, 37, 97
GpiOffsetClipRegion, 259, 266
GpiOffsetElementPointer, 173, 175
GpiOffsetRegion, 249, 252
GpiOpenSegment, 172, 173, 176, 195
GpiOutlinePath, 244, 247, 248, 260
GpiPaintRegion, 249, 253, 256
GpiPartialArc, 59, 70
GpiPathToRegion, 244, 252
GpiPlayMetaFile, 182, 317, 318, 320, 324
GpiPointArc, 59, 72
GpiPolyFillet, 59, 74–76
GpiPolyFilletSharp, 59, 75, 76
GpiPolygons, 78, 87
GpiPolyLine, 59, 64
GpiPolyLineDisjoint, 59, 65
GpiPolyMarker, 100, 102
GpiPolySpline, 73, 75
GpiPtInRegion, 249, 254, 271
GpiPtVisible, 259, 268
GpiPutData, 170, 172, 173, 182, 184
GpiQueryAttr, 53
GpiQueryBoundaryData, 270, 324
GpiQueryClipBox, 259, 267
GpiQueryClipRegion, 259, 266
GpiQueryDeviceBitmapFormats, 108, 111
GpiQueryDrawControl, 184
GpiQueryEditMode, 172
GpiQueryElement, 175, 195
GpiQueryElementPointer, 195
GpiQueryElementType, 172, 174, 195
GpiQueryFontMetrics, 155
GpiQueryFonts, 128, 129, 149
GpiQueryInitialSegmentAttrs, 178
GpiQueryKerningPairs, 155
GpiQueryLineEnd, 55
GpiQueryMetaFileBits, 316, 317, 330
GpiQueryMetaFileLength, 317, 330
GpiQueryModelTransformMatrix, 225, 226
GpiQueryPel, 120
GpiQueryRegionBox, 249, 255
GpiQueryRegionRects, 249, 255, 266
GpiQuerySegmentAttr, 178
GpiQuerySegmentNames, 179, 194
GpiQuerySegmentPriority, 179, 201
GpiQueryWidthsTable, 155
GpiRectInRegion, 249, 255, 256, 281
GpiRectVisible, 259, 268
GpiRemoveDynamics, 181, 183, 184
GpiResetBoundaryData, 270, 324
GpiRotate, 216, 220
GpiSaveMetaFile, 314, 316
GpiScale, 216, 217, 220, 221
GpiSetArcParams, 68
GpiSetAttr, 53, 55
GpiSetBackColor, 55
GpiSetBackMix, 55
GpiSetBitmap, 103, 111
GpiSetBitmapBits, 103, 111
GpiSetBitmapDimension, 111
GpiSetBitmapDimensions, 103
GpiSetBitmapId, 119

GpiSetCharBox, 142, 151
GpiSetCharSet, 141, 151, 152
GpiSetClipPath, 244, 259
GpiSetClipRegion, 250, 259, 265
GpiSetColor, 55
GpiSetDefaultView, 236
GpiSetDefaultViewMatrix, 234
GpiSetDrawControl, 184, 269
GpiSetDrawMode, 314
GpiSetEditMode, 172
GpiSetElementPointer, 173, 195
GpiSetElementPointerAtLabel, 175
GpiSetGraphicsField, 265
GpiSetInitialSegmentAttrs, 178
GpiSetLineEnd, 55
GpiSetLineJoin, 55
GpiSetLineType, 55
GpiSetLineWidth, 55
GpiSetLineWidthGeom, 55
GpiSetMetaFileBits, 316, 317
GpiSetMix, 55
GpiSetModelTransform, 227
GpiSetModelTransformMatrix, 216, 225, 234
GpiSetPageViewport, 239
GpiSetPel, 120
GpiSetPickAperturePosition, 276
GpiSetPickApertureSize, 276
GpiSetRegion, 249, 250
GpiSetSegmentAttrs, 178
GpiSetSegmentPriority, 179, 180
GpiSetSegmentTransformMatrix, 225–227
GpiSetStopDraw, 182
GpiSetTag, 272
GpiSetViewingLimits, 264
GpiSetViewingMatrix, 234
GpiSetViewingTransformMatrix, 233
GpiSpline, 59
GpiStrokePath, 78, 244, 246, 248
GpiTranslate, 216, 217
GpiWCBitBlt, 103, 114, 116
Graphic Data, 330, 336
Graphic order, 23
Graphic primitives, 52
Graphics data, 328
Graphics field, 265
Grid, 12, 21
Grid size, 12
Group, 10, 11, 22
Group object, 205

Help, 27
HIGHER_PRI, 179
Hit, 273, 274, 278, 281

Image, 23, 113
Image background color, 104
Image background mix, 104
Image color, 104
Image mix, 104
Images, 102, 103
Import WLY, 10
Initial program load, 141
Instance transform, 226
Internal leading, 158

KERNINGPAIRS, 155
Kernable, 125
Kerning, 99, 125, 140, 155
Kerning pair, 138
Kerning pairs, 99, 155

Label, 175
LINE_END, 244
LINE_JOIN, 244
LINE_TYPE, 244
Line, 59
Line color, 15, 16, 59
Line mix, 59
Line thickness, 15, 16
Line tool, 5, 13
Line type, 15, 16, 60
Line width, 60
Lines
  cosmetic, 60, 86
  geometric, 60, 85
Link, 201
List, 25
Lists, 21, 22
LOWER_PRI, 179
Local identifier, 90, 139, 150, 152
Logical font, 157, 158
Logical font description, 139

Marker, 24, 99
Marker background color, 100
Marker background mix, 100
Marker box, 102
Marker color, 100
Marker mix, 100
Marker set, 101
Marker symbol, 101
Markers, 4, 18
Marque select, 4

Matrix, 213, 220, 221, 225, 233, 238
Matrix multiplication, 222
Message queue, 46
Messages, 19
MetaFile, 10, 36, 176, 183, 233, 311–314, 316–320, 323, 325–328, 330, 336
Methods, 22
Metrics options, 12
Mirror, 209, 221, 227
Mix, 55, 58
Mix option, 114
Model parameter, 87
Model Transform, 211
Model transform, 17, 190, 191, 225, 233
Model transform matrix, 225, 227, 229
Monospaced, 124
Mouse actions, 18
Move, 4
Moving objects, 18
Multitasking, 32, 45

New, 9
Nonproportional, 124
Nonretained graphic segment, 176, 179
Normal presentation space, 176
Notebook dialog, 26
Notional units, 133, 134

Object oriented design, 50
Object oriented programming, 19, 50
OD_DIRECT, 305
OD_INFO, 305
OD_MEMORY, 305
OD_METAFILE, 305
OD_METAFILE_NOQUERY, 305
OD_MetaFile, 313
OD_MetaFile_NOQUERY, 313
OD_QUEUED, 305, 306
Open, 9
Operating system, 32
Order, 169, 173, 185, 190, 191, 195, 200, 312, 326, 328
OS/2 kernel, 37
OS/2 Toolkit, 31
Outline font, 44

Pan, 44, 210, 234
Panose descriptor, 138
Paste, 11, 22
Paste list, 11, 25, 205
Path, 244, 245, 247, 252, 259, 260, 263, 268
Paths, 85
Pattern reference point, 81
Pattern set, 80, 114
Pattern symbol, 80, 114
Pels, 104
Pencil tool, 5, 13
Pick aperture, 276
Picture elements, 104
PM_Q_RAW, 306
PM_Q_STD, 306
PM_SPOOLER_PRINTER, 290, 303
PM_SPOOLER_QP, 292
PM_SPOOLER_QUEUE, 291
PM_SPOOLER_QUEUE_DD, 291
PMF_COLORREALIZEABLE, 319
PMF_COLORTABLES, 319
PMF_DEFAULTS, 319
PMF_LCIDS, 318
PMF_LOADTYPE, 318
PMF_RESET, 318
PMF_SUPRESS, 319
Point, 123
Point size, 15, 21, 24, 126, 143, 150–152
Polyfill tool, 15
Polyline Fill tool, 7
Polyline tool, 7, 15
Port name, 284, 285, 287, 291
Presentation driver, 284, 286, 291, 304, 305, 307, 308
Presentation drivers, 36, 37
Presentation page, 34, 35, 38, 41, 154, 157, 231, 234, 239, 314
    units, 36
Presentation page size, 35
Presentation page units, 209
Presentation sapce, 36
Presentation space, 34, 37, 38, 41, 128, 139
    cached-micro presentation space, 34
    micro presentation space, 35
    normal presentation space, 35
PrfQueryProfileString, 290, 292, 303, 305
Print, 10
Print queue, 284–286
Printer device driver, 285
Printer name, 284, 286, 290, 303
Printer pooling, 285
Printer presentation driver, 286
Printer queue, 286
Printer sharing, 285
Product information, 11

Program objects, 26
Program template, 26
Proportional, 124

Queue, 291
Queue processor, 284, 286, 292, 306

Raw print mode, 285
Region, 248, 250, 252, 254, 265, 268, 281, 282
RES_NORESET, 324
RES_RESET, 324
RETAIN, 314
Resizing objects, 18
Retained graphics, 35, 175, 309
Retained segment, 22, 113, 116, 204, 206, 226
RGB, 110
Root segment, 178, 205, 273
Rotate, 4, 17, 22, 209, 213, 214, 227, 228, 231, 238
Rotation, 191
Rounded Box tool, 5, 14
Rubberband, 24, 59

Save, 9
Save as, 9
SCP_ALTERNATE, 260, 263
SCP_AND, 260
SCP_RESET, 260
SCP_WINDING, 260
Scale, 17, 22, 191, 209, 213, 214, 216, 217, 219, 227, 231, 234, 238, 240, 312, 325
Scaling, 324, 326
Scan line, 106
Scanline, 114
Scroll, 159, 234
SEGEM_INSERT, 172–174
SEGEM_REPLACE, 172, 174
Segment, 22, 23, 169–179, 183, 185, 190, 191, 195, 204, 206, 227, 233, 272–274, 277–279, 281, 282, 312, 326
  chained, 180
  unchained, 180
Segment chain, 177, 178, 205
Segment priority, 179
Select All, 12
Select list, 20, 21, 25, 192, 205
Select tool, 4, 13, 24
Semaphores, 46, 157
Set attribute, 22
Set Grid Size, 12
Set model transform, 17
SetWidthsTable, *99*
Shear, 17, 93, 209, 213, 224, 227, 231, 238
Shear angle, 191
Single/double-byte code page, 136, 140
Size, 4
Slope, 133
Snap to Grid, 12
Snap to grid, 65
Snap-to-grid, 238
Source code, 26, 30
Source files, 28
Space character, 135
Spline, 73
Spool Queue Manager, 285
Spooler, 285, 286, 307
Start document, 286
Status line, 167
Stroking the path, 245
Structured field, 326–328
Style options, 15
SUP_NOSUPPRESS, 324
SUP_SUPPRESS, 324
Sub–class, 22
Subclassing, 19
Subpicture, 208
Sweep angle, 70, 85
System Atom Table, 136, 138

Tab stops, 158
Tabs, 158, 166
Tag, 272, 277
Tag Image File Format, 312
Technical reference, 31
Template folder, 26
Terminate and Stay Resident (TSR), 32
Text alignment, 157
Text tool, 6, 14
Thread, 45, 49, 157, 166, 182, 193
TIFF, 9, 23, 312
Tool dialog box, 4, 13
Tool Meter, 12
Tool Palette, 8, 13
Tool palette, 23
Tools
  arc, 72
  ellipse, 68
  fillet, 75
  line, 63
  PolyLine, 64
  pencil, 65

rectangle, 66
Tools options, 13
TRANSFORM_ADD, 216, 225, 234
TRANSFORM_PREEMPT, 225, 234
TRANSFORM_REPLACE, 216, 225, 234
Transform coefficients, 214
Transform matrix, 226, 228, 231
Transformation, 120, 191, 207, 208, 215
Transformation coefficients, 213
Transformation matrix, 214, 227
Transformations, 39–41, 156, 168
Transforms, 126
Transforms options, 16
Translate, 308
Translation, 191, 209, 213, 214, 216, 234, 238, 240, 325
Translation coefficients, 227
Triplets, 327

Unchained graphic segment, 177
Unchained segment, 205
Ungroup, 10, 11, 22, 229
Update region, 256
updateRegion, 251

View selected objects, 10, 23
View transform matrix, 176
Viewing limit, 264
Viewing pipeline, 40, 168, 207, 211, 241, 257
Viewing Transform, 211
Viewing transform, 231, 232

Widths table, 21
Window context, 35
Window procedure, 19
WinFileDlg, 316
WinFontDlg, 151
WinOpenWindowDC, 36
WinQueryWindowRect, 324
World coordinates, 38, 90

XOR, 24

Z order, 10, 16
Zoom, 12, 44, 210, 234, 236, 308

Operating Systems/OS/2 $39.95

Learn by example how to get the most out of the full range of advanced **OS/2™ WARP GPI** *functions*

Programmers who work in the OS/2 Presentation Manager environment will appreciate this in-depth guide to fully exploiting the more advanced features of OS/2's GPI. The only guide devoted exclusively to the subject, it does much more than describe how GPI functions work, it actually shows you with numerous examples and a fully fleshed-out application. Written by IBM insiders Steve Knight and Jeffrey Ryan, it offers proven solutions to a wide range of user-interface problems, and detailed, step-by-step guidance to the full range of functions rarely covered in more general books on Presentation Manager.

- **The first book to focus entirely on the advanced features of OS/2's GPI updated for OS/2 WARP Version 3**
- **Covers such important topics as working with graphic primitives, working in different viewing coordinate spaces, metafiles, correlation, and more**
- **Helps programmers quickly learn how to use all the features discussed by developing a complete working application (a graphics editor) that they can easily adapt to their own work**
- **Extremely well illustrated, includes more than 60 screen shots, line drawings, and tables—all created with the enclosed sample application**

Disk includes:

- ✓ **A complete graphics editor application—both in source code and as an executable file**
- ✓ **Text browser file, query printer information, metafile viewer, and other utilities**

STEPHEN A. KNIGHT and JEFFREY M. RYAN are scientific programmers in the IBM Rochester Engineering Development Lab, Rochester, Minnesota, and patent holders. Other books authored by Stephen A. Knight include *Learning to Program OS/2 2.0 Presentation Manager by Example* and *Cooperative Processing with AS/400 PC Support.*

Cover Design: Adrienne Weiss
Cover Illustration: Pam-ela Harrelson

John Wiley & Sons, Inc.
Professional, Reference and Trade Group
605 Third Avenue, New York, N.Y. 10158-0012
New York • Chichester • Brisbane • Toronto • Singapore

ISBN 0-471-10718-2