

[Front Matter](#)
[Table of Contents](#)
[About the Author](#)

[Windows Graphics Programming Win32 GDI and DirectDraw®](#)

Feng Yuan

Publisher: Prentice Hall PTR

First Edition December 01, 2000

ISBN: 0-13-086985-6, 1234 pages

To deliver high-performance Windows applications, you need an in-depth understanding of the Win32 GDI and DirectDraw—but until now, it's been virtually impossible to discover what's going on "behind" Microsoft's API calls. This book rips away the veil, giving experienced Windows programmers all the information and techniques they need to maximize performance, efficiency, and reliability! You'll discover how to make the most of Microsoft's Windows graphics APIs—including the important new graphics capabilities built into Windows 2000.

Windows Graphics Programming Win32 GDI and DirectDraw®

Preface

- [What is this Book About?](#)
- [How is this Book Organized?](#)
- [How Should You Read this Book?](#)
- [What is on the CD?](#)
- [What is the Future of this Book?](#)
- [Acknowledgments](#)

1. Basic Techniques and Knowledge

- [1.1 Basic Windows Programming in C/C++](#)
- [1.2 Assembly Language](#)
- [1.3 Program Development Environment](#)
- [1.4 Win32 Executable File Format](#)
- [1.5 Architecture of Microsoft Windows OS](#)
- [1.6 Summary](#)

2. Windows Graphics System Architecture

- [2.1 Windows Graphics System Components](#)
- [2.2 GDI Architecture](#)
- [2.3 DirectX Architecture](#)
- [2.4 Printing Architecture](#)
- [2.5 Graphics Engine](#)
- [2.6 Display Drivers](#)
- [2.7 Printer Drivers](#)
- [2.8 Summary](#)

3. GDI /DirectDraw Internal Data Structures

- [3.1 Handles and Object-Oriented Programming](#)
- [3.2 Decoding GDI Object Handles](#)
- [3.3 Locating the GDI Object Handle Table](#)
- [3.4 Decoding the GDI Object Handle Table](#)
- [3.5 User Mode Data Structure of GDI Objects](#)
- [3.6 Accessing Kernel Mode Address Space](#)
- [3.7 WinDBG and the GDI Debugger Extension](#)
- [3.8 GDI Kernel Mode Data Structure](#)
- [3.9 DirectDraw Data Structure](#)
- [3.10 Summary](#)

4. Spying in the Windows Graphics System

- [4.1 Spying on Win32 API Calls](#)
- [4.2 Spying on Win32 GDI](#)
- [4.3 Spying on DirectDraw COM Interfaces](#)

[4.4 Spying on GDI System Calls](#)

[4.5 Spying on the DDI Interface](#)

[4.6 Summary](#)

[5. Graphics Device Abstraction](#)

[5.1 Modern Video Display Card](#)

[5.2 Device Context](#)

[5.3 Formalizing Device Context](#)

[5.4 Sample Program: Generic Frame Window](#)

[5.5 Sample Program: Painting and Device Context](#)

[5.6 Summary](#)

[6. Coordinate Spaces and Transformation](#)

[6.1 The Physical Device Coordinate Space](#)

[6.2 The Device Coordinate Space](#)

[6.3 The Page Coordinate Space and Mapping Modes](#)

[6.4 The World Coordinate Space](#)

[6.5 Using Coordinate Spaces](#)

[6.6 Sample Program: Scrolling and Zoom](#)

[6.7 Summary](#)

[7. Pixels](#)

[7.1 GDI Objects, Handles, and Handle Table](#)

[7.2 Clipping](#)

[7.3 Color](#)

[7.4 Drawing Pixels](#)

[7.5 Sample Program: Mandelbrot Set](#)

[7.6 Summary](#)

[8. Lines and Curves](#)

[8.1 Binary Raster Operations](#)

[8.2 Background Mode and Background Color](#)

[8.3 Pens](#)

[8.4 Lines](#)

[8.5 Bezier Curves](#)

[8.6 Arcs](#)

[8.7 Paths](#)

[8.8 Sample: Drawing Your Own Styled-Lines](#)

[8.9 Summary](#)

[9. Areas](#)

[9.1 Brushes](#)

[9.2 Rectangles](#)

[9.3 Ellipses, Chords, Pies, and Rounded Rectangles](#)

[9.4 Polygons](#)

[9.5 Closed Paths](#)

[9.6 Regions](#)
[9.7 Gradient Fills](#)
[9.8 Area Fills in Reality](#)
[9.9 Summary](#)

[10. Bitmap Basics](#)

[10.1 Device-Independent Bitmap Formats](#)
[10.2 A DIB Class](#)
[10.3 Displaying a DIB](#)
[10.4 Memory Device Contexts](#)
[10.5 Device-Dependent Bitmaps](#)
[10.6 Using DDBs](#)
[10.7 DIB Section](#)
[10.8 Summary](#)

[11. Advanced Bitmap Graphics](#)

[11.1 Ternary Raster Operations](#)
[11.2 Transparent Bitmaps](#)
[11.3 Transparency Without a Mask Bitmap](#)
[11.4 Alpha Blending](#)
[11.5 Summary](#)

[12. Image Processing Using Windows Bitmaps](#)

[12.1 Generic Pixel Access](#)
[12.2 Bitmap Affine Transformation](#)
[12.3 Fast Specialized Bitmap Transformer](#)
[12.4 Bitmap Color Transformation](#)
[12.5 Bitmap Pixel Transformation](#)
[12.6 Bitmap Spatial Filters](#)
[12.7 Summary](#)

[13. Palettes](#)

[13.1 System Palette](#)
[13.2 The Logical Palette](#)
[13.3 Palette Messages](#)
[13.4 Palette and Bitmaps](#)
[13.5 Color Quantization](#)
[13.6 Bitmap Color-Depth Reduction](#)
[13.7 Summary](#)

[14. Fonts](#)

[14.1 What's a Font?](#)
[14.2 Bitmap Fonts](#)
[14.3 Vector Fonts](#)
[14.4 Truetype Fonts](#)
[14.5 Font Installation and Embedding](#)

[14.6 Summary](#)

[15. Text](#)

- [15.1 Logical Fonts](#)
- [15.2 Querying Logical Font](#)
- [15.3 Simple Text Drawing](#)
- [15.4 Advanced Text Drawing](#)
- [15.5 Text Formatting](#)
- [15.6 Text Effects](#)
- [15.7 Summary](#)

[16. Metafile](#)

- [16.1 Metafile Basics](#)
- [16.2 Inside an Enhanced Metafile](#)
- [16.3 Enumerating an EMF](#)
- [16.4 An EMF as a Programming Tool](#)
- [16.5 Summary](#)

[17. Printing](#)

- [17.1 Understanding the Spooler](#)
- [17.2 Basic Printing Using GDI](#)
- [17.3 Design for Printing](#)
- [17.4 Drawing on Printer Device Context](#)
- [17.5 Summary](#)

[18. DirectDraw and Direct3D Immediate Mode](#)

- [18.1 Component Object Model \(COM \)](#)
- [18.2 DirectDraw Basics](#)
- [18.3 Building a DirectDraw Graphics Library](#)
- [18.4 Direct3D Immediate Mode](#)
- [18.5 Summary](#)



WINDOWS[®] GRAPHICS PROGRAMMING

Win32[®] GDI and DirectDraw[®]



Win32 GDI and DirectDraw: •
Accurate, under the hood, and in depth

Beyond the API: Internals, restrictions, •
performance, and real-life problems

Complete: Pixels, lines, curves, filled areas, •
bitmaps, image processing, fonts, text,
metafiles, printing, and more

Up to date: Windows 2000 and •
Windows 98 graphics enhancements

CD-ROM: Exclusive and professional-quality •
generic C++ classes, reusable functions,
demonstration programs, kernel mode
drivers, GDI exploration tools, and more!

Feng Yuan

Hewlett-Packard[®] Professional Books

Feng Yuan

FENG YUAN is a Software Design Engineer for Hewlett Packard in Vancouver, WA. He holds a Ph.D. in Software Engineering from Nanjing University. For the past four years, he has specialized in creating drivers for HP DeskJet printers—giving him a burning motivation and a unique opportunity to understand Windows graphics programming inside and out.

[NEXT >](#)

Preface

To be truly valuable, a new book on Windows programming should be in-depth, complete, up to date, accurate, practical, and useful.

An *in-depth* book should go beneath the API layer to talk about API design rationale, internal data structure, and implementation outline. It should provide spying and exploration tools for you.

A *complete and up-to-date* book should focus on the best implementation of Win32 API so far, Windows 2000, which will be basis for future Microsoft operating systems, and cover its new features.

An *accurate* book should be based on actual exploration of Win32 API features and verify all the details. Relying solely on Microsoft documentation is definitely not good enough, because it merely describes the abstract Win32 API and unfortunately has some incomplete, out-of-date, and vague information.

A *practical and useful* book for programmers and software engineers should go beyond mere description of API and simple illustrative examples, to solve real-world problems, provide reusable building blocks, supply useful tools, and encourage readers to write professional programs.

In particular, a book of this sort is needed to elucidate Win32 GDI, or Windows graphics programming in general, which is a fundamental building block of all Windows programs. The most in-depth coverage to date may be provided by Matt Pietrek's books, which discuss the internal working of Windows 3.1 GDI. The most complete and up-to-date description of GDI is offered by Microsoft's MSDN library. Half of Charles Petzold's famous book *Programming Windows, Fifth Edition*, is devoted to Windows 98 GDI programming.

But for the whole Windows programming community dealing with Windows GDI frequently, something more in-depth, more complete, more up-to-date, more accurate, and more useful is needed. These are the goals that have guided the preparation of this book.

[NEXT >](#)

WHAT IS THIS BOOK ABOUT?

This book is about Windows graphics programming using Win32 GDI API, with an introduction to DirectDraw, and an even more brief introduction to Direct3D Immediate Mode. It covers common features implemented on all Win32 platforms, pure 32-bit features implemented on Windows NT/2000 only, and the most recent additions to GDI for Windows 2000 and Windows 98. For example, alpha blending, transparent bit blting, gradient fill, right-to-left reading order, layered window, and sending JPEG/PNG images to printer are fully covered.

This book is about programming Windows with a deep understanding of how things are actually implemented, so as to use Win32 API more effectively, efficiently, and confidently.

This book is about reading any Win32 documentation perceptively and analytically, trying to understand the minds of people designing and implementing Win32 API, and using logical reasoning and experiments to fully understand Win32 API, even to find missing information and identify mistakes in its documentation.

This book is about using programming tools effectively to help you understand Win32 API. More importantly, it demonstrates how you can build your own tools, often using hard-core system programming techniques, and design interesting experiments to explore the undocumented world underneath Win32 API. The first few chapters can be read as a system-level Windows programming book, which can be applied to other areas of Windows programming.

This book is about creating reusable building blocks for practical employment. Besides simple testing and demonstration programs, the book contains lots of reusable functions, C++ classes, drivers, tools, and nontrivial programs, which can be used in actual production-quality Windows programs. It develops a whole C++ class library of object-oriented Windows programming, which supports simple window, SDI window, MDI window, dialog box, toolbar, status window, property sheet, sub-classing, common dialog box, etc. It provides classes for DIB/DDB/DIB section handling, EMF rendering, image processing, color quantization, error-diffusion halftoning, JPEG image decoding/encoding, font file decoding, font embed ding, PANOSE font matching, glyph drawing, 3D text, device-independent multiple-page layout, DirectDraw wrapper, Direct3D IM wrapper, and so on.

The code shown in this book does not rely on Microsoft Foundation Class, or any third-party class library, so it can be used in any C++ program. All the class names start with the letter "K," so you can easily mix them with MFC, ATL, or OWL, or your own class library.

HOW IS THIS BOOK ORGANIZED?

The book covers Windows graphics programming at three layers: the implementation layer, the API layer, and the application layer.

The implementation layer covers what is behind the Win32 GDI API and DirectX COM interfaces, which is the undocumented world of Windows graphics engine and the Win32 subsystem client DLLs. [Chapters 2, 3](#), and [4](#) cover the behind-the-scenes implementation layer to build a solid foundation for understanding the API layer.

The API layer provides precise, accurate, step-by-step description of Win32 GDI API, DirectDraw, and a little bit of Direct3D IM. The application layer builds on top of the API layer to solve real-world problems, implement reusable functions, C++ classes, and nontrivial programs. The API layer and application layer are mixed together according to individual topics. Normally, the first part of a chapter covers the API layer and then moves to real application-related programs. For more complicated topics like bitmap, one chapter covers the basics and two chapters cover more advanced usage.

[Chapter 1](#), “Basic Techniques and Knowledge,” reviews basic Windows programming techniques, which will be used in the rest of the book. It covers basic Windows programming, Intel assembly language, the program development environment, the Win32 executable file format, and the architecture of the Windows operating system. My favorite part is simple API hooking by overwriting Win32 module's import/export directories.

[Chapter 2](#), “Windows Graphics System Architecture,” gives a grand tour of the Windows graphics system from the Win32 subsystem DLLs down to the graphics device drivers. It touches on Windows graphics system components, GDI architecture, DirectX architecture, printing subsystem architecture, the graphics engine, display drivers, and finally printer drivers. My favorite parts are the description of system service calls, which bridges user mode GDI implementation with the kernel mode graphics engine, a tool to list undocumented system service calls (from GDI32.DLL, USER32.DLL, NTDLL.DLL, and WIN32K.SYS), and a simple printer driver which generates HTML pages with embedded bitmaps.

[Chapter 3](#), “GDI/DirectDraw Internal Data Structures,” can be read as a detective story or treasure-hunting adventure. It starts by explaining the Win32 handle-based object-oriented programming paradigm, then tries to understand what a GDI object handle is, proceeds to actually locate the GDI handle table, decode it, and finally expose the complicated web of data structures the Windows graphics system keeps internally. Virtual memory querying, Microsoft's debug symbol files, homegrown tools and Microsoft Visual C++ debugger are used to locate the GDI handle table. A kernel mode driver is developed to read from the kernel mode address space. The *Fosterer* program developed in [Chapter 3](#) uses Microsoft's GDI debugger extension to decode the GDI handle table, and graphics engine/DirectX internal data structures, all with the convenience of a single machine.

You just can't afford to miss trying the Fosterer program, on either a Windows NT or Windows 2000 machine. But first, you need to install debug symbol files and get Microsoft's WinDbg.

Description of internal data structure should be treated as reference material to gain deep understanding or help in DDI level debugging, as the details may change from release to release, even from service pack to service pack. Feel free to skip any sections not interesting to you and come back when you need some insights, for example to understand GDI object resource usage or performance issues.

[Chapter 4](#), "Spying in the Windows Graphics System," presents various techniques and tools for spying on Windows graphics system and the whole Windows system in general. You will learn about injecting DLL into foreign processes, hooking into the API calling chain, information gathering, decoding, and reporting, API specification for the spying program, spying on Win32 API calls, API hooking by binary-relocation, system service call hooking, COM interface hooking, and finally kernel mode DDI interface hooking. My favorite parts are using assembly to write proxy routines, and hooking on intramodular calls, system service calls, and DDI calls to watch how the system really works. [Chapter 4](#) is for hard-core programmers; skip it if you do not need it now.

[Chapter 5](#), "Graphics Device Abstraction," begins a description of the whole Windows graphics programming APIs and practice usage. [Chapter 5](#) deals with video display card, frame buffer, GDI device context object, generic frame window class, and painting in a window. My favorite part is the *WinPaint* program that visualizes a window's painting messages.

[Chapter 6](#), "Coordinate Spaces and Transformation," discusses the four coordinate spaces supported by GDI, window-to-viewport mapping, world coordinate transformation (affine transformation), and finally their usage in scrolling and zooming. I missed playing WeiQi, an oriental board game, so much during the writing of this book that I wrote a simple WeiQi board displaying program as [Chapter 6](#)'s sample program.

[Chapter 7](#), "Pixels," gives a generic description of GDI objects, handles, and handle table on the GDI API level, which is followed by a program to monitor system-wide GDI handle usage. It then discusses simple region and provides a complete picture of GDI clipping, color spaces, and pixel drawing, ending with a Mandelbrot set drawing program. The best part may be the description of the system region, meta region, clipping region, and Rao region, which are used by GDI to control clipping, and the *ClipRegion* program to visualize them.

[Chapter 8](#), "Lines and Curves," covers binary raster operations, background modes, background colors, logical pen object, lines, Bezier curves, arcs, paths, and drawing your own style lines not supported directly by GDI. The part I like most may be the mathematics involved in converting elliptical curves to Bezier curves.

[Chapter 9](#), "Areas," covers brushes, rectangles, ellipses, chords, pies, rounded rectangles, polygons, closed paths, regions, gradient fills, and a summary of various area-fill techniques used by graphics applications. My favorites are the use of gradient fill to draw 3D buttons and the description of region-related data structure.

[Chapter 10](#), “Bitmap Basics,” focuses on the three bitmap formats supported by GDI, which are device-independent bitmaps, device-dependent bitmaps, and DIB section. Also covered are wrapper classes for DIB, DDB, and DIB section, memory device contexts, and common usage of these bitmaps. My favorite parts are the wrapper classes, especially using memory-mapped DIB section to implement device-independent high-resolution rendering of enhanced metafiles.

[Chapter 11](#), “Advanced Bitmap Graphics,” covers ternary raster operations, transparent bitmap, bitmap transparency without masks, alpha blending, and the new Windows 2000 feature, layered window. My favorite part is the complete coverage of raster operations, especially the raster operating chart and simulation of quaternary raster operation using multiple ternary raster operations.

[Chapter 12](#), “Image Processing Using Windows Bitmaps,” discusses direct pixel access in bitmaps, bitmap affine transformation, bitmap color transformation, bitmap pixel transformation, and bitmap spatial filters. My favorite part is the object-oriented template-based design of the generic image-processing framework presented in this chapter, which can be easily extended.

[Chapter 13](#), “Palettes,” covers the system palette, logical palettes, palette messages, palettes for bitmaps, color quantization, and bitmap color-depth reduction using error diffusion. The octree color quantization algorithm implementation has been seen to generate better palette than commercial software.

[Chapter 14](#), “Fonts,” covers character sets, code pages, glyph, typeface, font family, bitmap fonts, vector fonts, TrueType fonts, font installation, and font embedding. I especially like decoding the TrueType font file.

[Chapter 15](#), “Text,” discusses logical fonts, font mapping, PANOSE typeface matching, text metrics, simple text drawing, advanced text drawing, text formatting, and text effects. The last section on text effects is the best; it includes coloring text, shadowing, embossing, engraving, soft-shadowing, rotation, vertical text, fitting text on curves, treating text as images, treating text as outline, and even simple 3D text.

[Chapter 16](#), “Metafile,” covers basic creating and displaying of metafiles, detailed internal metafile organization, encoding of GDI feature in metafiles, decoding metafiles, enumerating metafiles, EMF decompiler, and capturing EMF from spooler. The best parts are the EMF decompiler and the *EmfScope* program, which captures Windows 95/98 EMF spool files.

[Chapter 17](#), “Printing,” discusses print spooler, basic printing using GDI, designing your application for printing, JPEG image printing (including sending JPEG directly to a printer driver), and C++ syntax highlighted source code printing. The best part is a generic framework for multiple-page, multiple-column page layout that is independent of device resolution and display scale. Both the JPEG and source code printing programs use this generic framework.

[Chapter 18](#), “DirectDraw and Direct3D Immediate Mode,” is an introduction to the 2D/3D drawing

portions of DirectX for experienced GDI programmers. It covers a COM primer, wrapper classes for DirectDraw and DirectDraw surface, three ways of drawing in DirectDraw, DirectDraw clipper, off-screen surface, font and text in DirectDraw, wrapper class for basic DirectDraw Immediate Mode, windowed-mode DirectX, double-buffering, and texture. The part I enjoy most is using GDI to create a DirectDraw font surface, which can be used to display text efficiently on DirectX surfaces.

[< BACK](#) [NEXT >](#)

HOW SHOULD YOU READ THIS BOOK?

This book is mainly written for intermediate, experienced, or advanced programmers using the Win32 API or class libraries based upon it.

Novice Windows programmers should start either with another book or with on-line help to get familiar with the skeleton and basic concepts of a Windows program, and step through a program to understand how it works.

If you're interested only in Windows graphics programming stuff, or you're not interested in behind-the-scenes system-level implementation details, read [Chapters 1](#) and [2](#), skip [Chapters 3](#) and [4](#), and read [Chapter 5](#) onward. You can even skip some sections in [Chapters 1](#) and [2](#). Starting from [Chapter 5](#), the contents are developed quite gradually in a systematic fashion.

If you're an experienced or advanced programmer, you know perfectly well what interests you most. Maybe you will check a few things first and then jump to [Chapter 3](#).

If you're interested only in system-level programming like API spying, read parts of [Chapters 1](#) and [2](#), and then [Chapters 3](#) and [4](#).

If you're not a programmer—for example, if you're a test engineer—you may want to read [Chapter 2](#) to get an overall view of the Windows graphics system, and maybe the first part of [Chapter 3](#) in order to know more than enough to talk about GDI resource leak problems, and get a tool or tools to detect resource leak.

WHAT IS ON THE CD?

This book comes with lots of sample programs, together with reusable functions and classes. To be more specific, it has over 1300 KB of C++ source files, 400 KB of C++ header files, plus a slightly modified version of JPEG library source files, which is based on the Independent JPEG Group's free source code (www.ijg.org). The complete source code is compiled into 49 executable files, three kernel mode drivers, and one user mode dynamic library.

Naturally, not all the source code can be printed in the book. The CD contains all the source code, Microsoft Visual C++ 6.0 workspace files, precompiled debug mode and release mode binary files, and some JPEG images to be used with chapters related to bitmap/image handling. The CD has an automatic installation program, which will install the contents of the CD, program groups, links and important web addresses for downloading Microsoft tools and for support.

The programs were developed and tested on the Windows 2000 final release (build 2195), with a display card supporting DirectX 7.0 2D/3D-hardware acceleration, although most programs should work on Windows 95/98/NT 4.0 systems and do not need DirectX support.

To compile all the programs, make sure the following are installed on your system:

- **Visual C++ 6.0**, for compiler.
- **Visual Studio 6.0 Service pack 3**, for compiler update,
msdn.microsoft.com/vstudio/sp/vs6sp3.
- **MSDN library**, for on-line documentation.
- **Platform SDK**, for latest header files, library files, and tools,
www.microsoft.com/downloads/sdks/platform/platform.asp. Make sure your VC 6.0 include and library directories are updated to use Platform SDK.
- **Windows 2000 debug symbol files**, used by several tools, and good for debugging,
www.microsoft.com/windows2000/downloads/otherdownloads/symbols.
- **Windows 2000 DDK**, needed by several kernel mode drivers, www.microsoft.com/ddk. Add DDK's inc directory to your VC include directories. Add DDK's libfre\i386 directory to VC library file directories.
- **WinDebug**, used by tools in [Chapter 3](#), www.microsoft.com/ddk/debugging.

Although all the sample code in this book is written in C++ without MFC, MFC/ATL/OWL programmers can easily make use of the code developed here. Even Visual Basic or Delphi programmers can benefit from the text and sample code from the book, because these development environments allow direct calling of Win32 API functions.

[< BACK](#) [NEXT >](#)

WHAT IS THE FUTURE OF THIS BOOK?

Writing is a great opportunity for the author, forcing him to organize ideas, conduct research, and present material in an orderly fashion. When a book is finished, the author benefits most. Hopefully, fellow programmers can also learn from the book, which is detailed recording of my own learning.

More importantly, the classroom for learning is now suddenly exploded from my office cubicle and my home office to the whole wide world; now you all become my teachers and classmates. If you find any mistake or bug, or if you have a comment, suggestion, or complaint, please contact me through my personal web site: <http://www.fengyuan.com>.

The web site will also provide frequently asked questions, updates, instructions for using the more complicated sample programs, and so on.

[< BACK](#) [NEXT >](#)

ACKNOWLEDGMENTS

This book would not have come into being without the help, encouragement, and support of many people, to whom I'm truly grateful.

I would like to thank HP Press acquisition editor Susan Wright and Prentice Hall PTR acquisition editor Jill Pisoni for signing an unknown software engineer to write a 650-page book which has turned out to be 1200 pages, and for forgiving all the delays.

At Prentice Hall PTR, Development Editor James Markham and Production Editor Faye Gemmellaro did a superb job in adjusting the book's outline, organizing technical reviews, suggesting ways for better presentation, improving grammar, and identifying and resolving problems.

Hewlett-Packard has a wonderful program to encourage employees to write technical books with well-defined HP management support. I would like to thank my book champion, and my manager, Ivan Crespo, for his ongoing support during the development of this entire book.

Four years ago I transferred to Hewlett-Packard's Vancouver R&D Lab, the home for the world-famous HP DeskJet printers, with some basic understanding of Win16 programming. Through source code reading, discussing, debating, and developing tools, not to mention tracing into assembly code using SoftICE/W, I learned so much that I felt confident enough to contact HP Press about eighteen months ago to start this project. I would like to thank members of the HP Vancouver R&D Lab for what I've learned from you, what we have learned together, and your encouragement.

Most importantly, my wife Ying Peng has my eternal gratitude for simply believing in me, understanding me, and supporting me, before and during the long battle with GDI on weekends, evenings, and late nights. Our son, QiaoChu, also tried to help by reading from the screen every evening before going to bed. Now we finally have time to build his underwater robot together this summer.

Feng Yuan

Chapter 1. Basic Techniques and Knowledge

We are about to begin our adventure into the Windows graphics system, from the surface of the Win32 API to the rock bottom of display/printer drivers. There are lots of important areas in the Windows graphics system. We will focus on the most important parts: the Win32 GDI (Graphics Device Interface) and the DirectDraw portion of DirectX.

Win32 GDI APIs are implemented on multiple platforms like Win32s, Win95/98, Win NT 3.5/4.0, Windows 2000, and WinCE, with significant differences among them. For example, Win32s and Win95/98 are based on the old 16-bit GDI implementation that has lots of restrictions, while the true 32-bit implementations on Windows NT 3.5/4.0 and Windows 2000 are much more powerful. The DirectDraw interfaces are implemented on Win95/98, Win 4.0, and Windows 2000 platforms. This book will be targeted mainly at the Windows NT 4.0 and Windows 2000 platforms, the most powerful implementation of those APIs. Special notes for other platforms will be added as appropriate.

Before we go deeply into the Windows graphics system, we need to review a few basic techniques that will be vital to our exploration. In this chapter, we will cover simple Windows programming in C/C++, a little bit of assembly-language programming, program development and debugging tools, the format of the Win32 execution file, and the architecture of the Windows Operating System.

Note

It is assumed that readers are intermediate to advanced Windows programmers, so the coverage of those techniques is very brief. Readers will be referred to other books for in-depth coverage of those topics.

1.1 BASIC WINDOWS PROGRAMMING IN C/C++

The working languages of our profession have developed dramatically from the dark age of machine languages to the modern-day programming languages like C, Visual Basic, Pascal, C++, Delphi, and Java. It's said that the number of lines of code a programmer writes a day is almost the same no matter what language he or she uses. So, naturally, with languages moving higher and higher in level of abstraction, programmers' productivity gets higher and higher.

The most common single language used for Windows programming may be C, as can be seen from the sample programs that come with the Microsoft Platform Software Development Kit (Platform SDK) and Device Driver Kit (DDK). Object-oriented languages like C++, Delphi, and Java are catching up quickly, to replace C and Pascal as the new-generation languages for Windows programming.

Object-oriented languages are definitely improvements over their predecessors. C++, for example, even without its pure object-oriented features such as class, inheritance, and virtual function, improves upon C with modern features such as strict function prototyping, inline function, overloading, operator, and template.

But writing object-oriented Windows programs is not an easy job, mainly because the Windows API was not designed to support object-oriented languages. For example, callback functions like the Windows message procedure and the dialog message procedure must be global functions. The C++ compiler will not let you pass a normal class member function as a callback function. Microsoft Foundation Class (MFC) is designed to wrap the Windows API in a class hierarchy, which has now almost become the de facto standard for writing object-oriented Windows programs. MFC goes to great length to bridge object-oriented C++ and C oriented Win32 API. MFC passes a single global function as the generic window message procedure, which uses a map from HWND to a CWnd object pointer to translate a Win32 window handle into a pointer to a C++ window object. Even Microsoft is worried about the size and complexity of MFC when OLE, COM, and Active X are becoming popular, so the recommended class library to write lightweight COM server and Active X controls is yet another class library from Microsoft, Active Template Library (ATL).

Following the trend of moving toward object-oriented programming, the sample code presented in this book is mostly C++, rather than C. To make the sample code readily useable to C programmers, C++ programmers, MFC programmers, ATL programmers, C++ builder programmers, or even Delphi and Visual BASIC programmers, neither fancy features of C++ nor MFC/ATL are used in this book.

Hello World Version 1: Starting Your Browser

Let's get down now to the business of writing a basic Windows program in C. Here is our first Windows program:

```
// Hello1.cpp
#define STRICT
#include <windows.h>
#include <tchar.h>
#include <assert.h>
```

```
const TCHAR szOperation[] = _T("open");
const TCHAR szAddress[] = _T("www.helloworld.com");
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR lpCmd, int nShow)
{
    HINSTANCE hRslt = ShellExecute(NULL, szOperation,
        szAddress, NULL, NULL, SW_SHOWNORMAL);

    assert( hRslt > (HINSTANCE) HINSTANCE_ERROR);

    return 0;
}
```

Note

Sample programs accompanying the book are put into individual subdirectories for each chapter, named **Chapt_01**, **Chapt_02**, etc. An extra directory on the same level, called **include**, contains all the common shared codes. Each chapter directory has one Microsoft Visual C++ workspace file, which contains all the projects within the chapter. Each project resides in its own subdirectory. For example, the Hello1 project is in the Chapt_01\Hello1 directory. To reference common files like **win.h**, relative paths like “**..\\..\\include\\win.h**” are used in the source code.

This “Hello World” program is not just another hello world program, it's the hello world program for the Internet age. If you could run this program, your browser would bring up a web page through a powerful Win32 API call, **ShellExecute**.

A few points may worth mentioning in this simple program, as it may not look the same as the skeleton program shown in other books. These points are included here to encourage good programming habits.

The program starts by defining a macro **STRICT**, which tells the Windows include file to treat different object types differently, making it easier for the compiler to give programmers warning messages if they mix **HANDLE** with **HINSTANCE**, or **HPEN** with **HBRUSH**. If you hear a reader complaining that sample programs from certain books can't even compile, it is likely that the sample programs were not tested with the macro **STRICT** defined. This happens because the newer versions of Windows include files that turn **STRICT** on by default, while the older versions do not.

Including **<tchar.h>** makes the single source code compilable both for non-UniCode and UniCode versions of binary code. Programs targeted for Windows 95/98 lines of OS are not recommended to be compiled in UniCode mode unless the programmer is extremely careful in avoiding UniCode-based API, which are not implemented on Win95/98. Be careful that the **lpCmd** parameter to **WinMain** is always non-UniCode; you have to use the **GetCommandLine()** to get the **TCHAR** version of the full command line.

Including **<assert.h>** is for defensive programming. It's highly recommended that programmers step through every line of their code in a debugger to catch possible errors. Putting asserts on incoming parameters and function return results helps check for unexpected situations efficiently over the whole development phase. Another way to contain programming errors is to add exception handling to your code.

The two “const TCHAR” array definitions ensure that those constant strings are put into the read-only data section in the final binary code generated by the compiler and linker. If you include strings like _T(“print”) directly in the call to ShellExecute, it will likely end up in the read-write data section of the binary code. Putting constants into the read-only section ensures that they are really read-only, so that attempts to write to them will generate protection faults. It also makes them shareable among multiple instances, thus reducing physical RAM usage if multiple instances of the same module are loaded on a system.

The program does not name the second parameter of WinMain, normally called hPrevInstance, because it's not used by Win32 programs. The hPrevInstance is supposed to be the instance handle of the previous instance of the current program for Win16 programs. For Win32 programs, all programs have separate address spaces. Even if there are multiple instances of the same program running, they can't normally see each other.

It's hard, if not impossible, to write the perfect source code, yet we can learn some tricks to tell the compiler to build the perfect binary code. The important thing is to select the right CPU, runtime library, optimization, structure alignment, and DLL base address. Adding debug information, a map file, or even an assembly code listing will help you debug, analyze defect reports, or fine-tune performance. Another idea is to use map file, the quick viewer feature of Windows 95/96/NT Explorers, or Dumpbin to analyze your binary code to see that the right functions are exported, no strange functions are imported, and no section within the binary code surprises you. For example, if your code imports oleauto32.dll function 420, your code would not run on earlier releases of Win95. If your program loads lots of DLLs using the same default base address, it will be slower during loading because of dynamic relocation.

If you compile the Hello1 project with the default setting, the release mode binary executable is 24 kB in size. It imports three dozen Win32 API functions, although the source code uses only one of them. It has close to 3000 bytes of initialized global data, although the program does not use any directly. If you try to step into the code, you will soon find out that WinMain is not actually the starting point of our program. Lots of things happen before WinMain is called in the actual program startup routine [WinMainCRTStartup](#)

If your program is as simple as Hello1.cpp, you can choose to use the DLL version of the C runtime library, write your own implementation of WinMainCRTStartup, and have the compiler/linker generating really small binary code. This is shown in the next code sample.

Hello World Version 2: Drawing Directly to Desktop

Since this is a book on Windows graphics programming, we should focus on using graphics API functions. So here is our second version of “Hello World”:

```
// Hello2.cpp
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <tchar.h>
#include <assert.h>

void CenterText(HDC hDC, int x, int y, LPCTSTR szFace,
    LPCTSTR szMessage, int point)
```

```
{  
    HFONT hFont = CreateFont(  
        —point * GetDeviceCaps(hDC, LOGPIXELSY) / 72,  
        0, 0, 0, FW_BOLD, TRUE, FALSE, FALSE,  
        ANSI_CHARSET, OUT_TT_PRECIS, CLIP_DEFAULT_PRECIS,  
        PROOF_QUALITY, VARIABLE_PITCH, szFace);  
    assert(hFont);  
  
    HGDIOBJ hOld = SelectObject(hDC, hFont);  
  
    SetTextAlign(hDC, TA_CENTER | TA_BASELINE);  
  
    SetBkMode(hDC, TRANSPARENT);  
    SetTextColor(hDC, RGB(0, 0, 0xFF));  
    TextOut(hDC, x, y, szMessage, _tcslen(szMessage));  
    SelectObject(hDC, hOld);  
    DeleteObject(hFont);  
}  
  
const TCHAR szMessage[] = _T("Hello, World");  
const TCHAR szFace[] = _T("Times New Roman");  
  
#pragma comment(linker, "-merge:.rdata=.text")  
#pragma comment(linker, "-align:512")  
  
extern "C" void WinMainCRTStartup()  
{  
    HDC hDC = GetDC(NULL);  
    assert(hDC);  
  
    CenterText(hDC, GetSystemMetrics(SM_CXSCREEN) / 2,  
        GetSystemMetrics(SM_CYSCREEN) / 2,  
        szFace, szMessage, 72);  
  
    ReleaseDC(NULL, hDC);  
    ExitProcess(0);  
}
```

The program shown above uses some simple GDI functions to draw the string “Hello, World” in the center of the screen without creating a window. The program gets a device context of the desktop window, or that of the primary monitor if you have multiple monitors, creates an italic font one inch in height, and draws the “Hello, World” string in transparent mode using a solid blue color.

To make the binary code small, the code implements the startup routine WinMainCRTStartup directly, instead of using its default implementation provided by the C/C++ runtime library. ExitProcess is the last statement of the program to end the process. The code also instructs the linker to merge the read-only data section (.rdata) with the read-executable code section (.text). The final executable generated in release mode is 1536 bytes in size, only a small portion of which is the actual machine code.

Hello World Version 3: Creating a Full-Screen Window

The first and second versions of “Hello, World” are not the usual window programs. They use only a few Windows API calls to illustrate how to write very simple skeleton Windows programs.

A normal window program in C/C++ starts with registering a few window classes, followed by creating a main window and possibly a few child windows, and finally goes to a message loop to dispatch all incoming messages to the corresponding window message procedures.

Most readers should be very familiar with the structure of this type of a skeleton Windows program. Instead of duplicating such a program, we will try to develop a simple object-oriented window program in C++ without the help of Microsoft Foundation Class.

What we need is a very basic KWindow class, which implements the basic task of registering the window class, creating a window, and dispatching window messages. The first two tasks are quite simple, but the third is a little tricky. We would certainly like the window message handler to be a virtual member function of the KWindow class, but the Win32 API will not accept it as a valid window procedure. A C++ class member function passes an implicit “this” pointer for every function call, and it may use a different calling convention than what's used by the window procedure. One common way to solve the problem is to provide a static window procedure, which then dispatches to the right C++ member function. To do this, the static window procedure needs a pointer to an instance of KWindow. Our solution to this problem is to pass a pointer to a KWindow instance through the CreateWindowEx call, which is then set into the data structure associated with each window.

Note

All the C++ class names in this book start with letter “K,” instead of the traditional letter “C.” The intention is to allow easy reuse of these classes in programs using MFC, ATL, or other class libraries.

Here is the header file for the KWindow class:

```
// win.h
#pragma once
class KWindow
{
    virtual void OnDraw(HDC hDC)
    {
    }
    virtual void OnKeyDown(WPARAM wParam, LPARAM lParam)
    {
    }

    virtual LRESULT WndProc(HWND hWnd, UINT uMsg,
        WPARAM wParam, LPARAM lParam);
```

```
static LRESULT CALLBACK WindowProc(HWND hWnd,
    UINT uMsg, WPARAM wParam, LPARAM lParam);

virtual void GetWndClassEx(WNDCLASSEX & wc);
public:

HWND m_hWnd;

KWindow(void)
{
    m_hWnd = NULL;
}

virtual ~KWindow(void)
{
}

virtual bool CreateEx(DWORD dwExStyle,
    LPCTSTR lpszClass, LPCTSTR lpszName, DWORD dwStyle,
    int x, int y, int nWidth, int nHeight, HWND hParent,
    HMENU hMenu, HINSTANCE hInst);

bool RegisterClass(LPCTSTR lpszClass, HINSTANCE hInst);

virtual WPARAM MessageLoop(void);

BOOL ShowWindow(int nCmdShow) const
{
    return ::ShowWindow(m_hWnd, nCmdShow);
}

BOOL UpdateWindow(void) const
{
    return ::UpdateWindow(m_hWnd);
}
};
```

The KWindow class has a single member variable, `m_hWnd`, which is the window handle. It has a constructor, a virtual destructor, a function to create the window, and functions for message loop, displaying, and updating windows. KWindow's private member function defines the `WNDCLASSEX` structure and handles window messages. `WindowProc` here is the static function as required by the Win32 API, which dispatches messages to the C++ virtual member function `WndProc`.

Quite a few member functions are defined as virtual functions to allow derived classes of KWindow to change their behaviors. For example, different classes will have different `OnDraw` implementations and will have different menus and cursors in their `GetWndClassEx` implementation.

We use a nice pragma provided by the Visual C++ compiler (`#pragma once`) to avoid including the same header file

more than once. This can be replaced by defining a unique macro for every header file and then skipping a header file if the macro is already defined.

Here is the implementation for the KWindow class:

```
// win.cpp
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <assert.h>
#include <tchar.h>
#include ".\win.h"

LRESULT KWindow::WndProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    switch( uMsg )
    {
        case WM_KEYDOWN:
            OnKeyDown(wParam, lParam);
            return 0;
        case WM_PAINT:
        {
            PAINTSTRUCT ps;

            BeginPaint(m_hWnd, &ps);
            OnDraw(ps.hdc);
            EndPaint(m_hWnd, &ps);
        }
        return 0;
        case WM_DESTROY:
            PostQuitMessage(0);
            return 0;
    }

    return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

LRESULT CALLBACK KWindow::WindowProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    KWindow * pWindow;

    if ( uMsg == WM_NCCREATE )
    {
        assert( ! IsBadReadPtr((void *) lParam,
            sizeof(CREATESTRUCT)) );
    }
}
```

```
MDICREATESTRUCT * pMDIC = (MDICREATESTRUCT *)
    ((LPCREATESTRUCT) IParam)->lpCreateParams;
pWindow = (KWindow *) (pMDIC->IParam);

assert( ! IsBadReadPtr(pWindow, sizeof(KWindow)) );
SetWindowLong(hWnd, GWL_USERDATA, (LONG) pWindow);
}

else
pWindow=(KWindow *)GetWindowLong(hWnd, GWL_USERDATA);

if ( pWindow )
    return pWindow->WndProc(hWnd, uMsg, wParam, IParam);
else
    return DefWindowProc(hWnd, uMsg, wParam, IParam);
}

bool KWindow::RegisterClass(LPCTSTR lpszClass, HINSTANCE hInst)
{
    WNDCLASSEX wc;

    if ( ! GetClassInfoEx(hInst, lpszClass, &wc) )
    {
        GetWndClassEx(wc);

        wc.hInstance    = hInst;
        wc.lpszClassName = lpszClass;
        if ( !RegisterClassEx(&wc) )
            return false;
    }

    return true;
}

bool KWindow::CreateEx(DWORD dwExStyle,
    LPCTSTR lpszClass, LPCTSTR lpszName, DWORD dwStyle,
    int x, int y, int nWidth, int nHeight, HWND hParent,
    HMENU hMenu, HINSTANCE hInst)
{
    if ( ! RegisterClass(lpszClass, hInst) )
        return false;

    // use MDICREATESTRUCT to pass this pointer, support MDI child window
    MDICREATESTRUCT mdic;
    memset(& mdic, 0, sizeof(mdic));
    mdic.IParam = (LPARAM) this;
    m_hWnd = CreateWindowEx(dwExStyle, lpszClass, lpszName,
        dwStyle, x, y, nWidth, nHeight,
        hParent, hMenu, hInst, & mdic);
```

```
return m_hWnd!=NULL;
}

void KWindow::GetWndClassEx(WNDCLASSEX & wc)
{
    memset(& wc, 0, sizeof(wc));

    wc.cbSize      = sizeof(WNDCLASSEX);
    wc.style       = 0;
    wc.lpfnWndProc = WindowProc;
    wc.cbClsExtra  = 0;
    wc.cbWndExtra   = 0;
    wc.hInstance   = NULL;
    wc.hIcon        = NULL;
    wc.hCursor      = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)GetStockObject(WHITE_BRUSH);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = NULL;
    wc.hIconSm     = NULL;
}
WPARAM KWindow::MessageLoop(void)
{
    MSG msg;

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

    return msg.wParam;
}
```

Implementing KWindow is quite straightforward, except for the static function WindowProc. WindowProc is responsible for dispatching all messages from the Windows OS to the right KWindow message handler. To do this, we need a way to find a pointer to a KWindow class instance from a WIN32 window handler. But the pointer is passed to CreateWindowEx call only once. If something we need over and over is given to us only once, we have to store it somewhere handy. Microsoft Foundation Class chooses to store the information in a global map that associates HWND with pointers to CWnd instances, so that every time a message needs to be dispatched, a search is made to find the right CWnd through a hash-based search in the map. The implementation for our simple KWindow class here chooses to store the KWindow instance pointer with the data structure that the Windows OS maintains with every window. WindowProc initially gets a pointer to KWindow during WM_NCCREATE message handling, which is actually sent before the WM_CREATE message and has the same pointer to the CREATE STRUCT structure. It stores the pointer using the Set Window Long(GWL_USERDATA) call, which can later be retrieved by the Get WindowLong (GWL_USERDATA) call. This completes the bridge from WindowProc to KWindow::WndProc for our simple window class.

The trouble with the normal C-based message handler is that if it needs to access extra data, it has to use global data. If multiple instances of windows are created sharing the same message handler, the message handler will normally not work. To allow for multiple instances of windows for a single window class, each instance needs its own

copy of the data, which can be accessed by the generic message handler. The KWindow class solves this problem by providing a C++ message handler that has access to per-instance data.

The KWindow::CreateEx method does not pass “this” pointer directly to the CreateWindowEx WIN32 function; instead it's passed as a member of the MDI CREATE STRUCT. This is needed to support an MDI (Multiple Document Interface) child window using the same KWindow class. To create an MDI child window, the user application sends a WM_MDICREATE message to an MDI client window, passing it a MDICREATESTRUCT. The client window, which is implemented by the operating system, is the one responsible for calling the final window creation function CreateWindowEx. Also note that the CreateEx function handles both window class registration and window creation in a single function. It checks if the class is already registered every time a window needs to be created, registering the class only when needed.

With the KWindow done, we don't have to repeat the tasks of registering a class, creating a window, and message looping over and over again in our book. We can just derive a class from KWindow and define what's special about that derived class.

Here is our third version of “Hello, World,” a normal window program version using C++:

```
// Hello3.cpp
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <assert.h>
#include <tchar.h>

#include "..\..\include\win.h"

const TCHAR szMessage[] = _T("Hello, World !");
const TCHAR szFace[]   = _T("Times New Roman");
const TCHAR szHint[]   = _T("Press ESC to quit.");
const TCHAR szProgram[] = _T("HelloWorld3");

// copy CenterText from Hello2.cpp

class KHelloWindow : public KWindow
{
    void OnKeyDown(WPARAM wParam, LPARAM lParam)
    {
        if (wParam==VK_ESCAPE )
            PostMessage(m_hWnd, WM_CLOSE, 0, 0);
    }
    void OnDraw(HDC hDC)
    {
        TextOut(hDC, 0, 0, szHint, lstrlen(szHint));
        CenterText(hDC, GetDeviceCaps(hDC, HORZRES)/2,
                   GetDeviceCaps(hDC, VERTRES)/2,
                   szFace, szMessage, 72);
    }
}
```

```
}

public:

};

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE,
                   LPSTR lpCmd, int nShow)
{
    KHelloWindow win;

    win.CreateEx(0, szProgram, szProgram, WS_POPUP, 0, 0,
                GetSystemMetrics(SM_CXSCREEN),
                GetSystemMetrics(SM_CYSCREEN),
                NULL, NULL, hInst);

    win.ShowWindow(nShow);
    win.UpdateWindow();

    return win.MessageLoop();
}
```

The code shown derives a KHelloWindow class from a KWindow class, overriding the virtual function OnKeyDown to handle the ESC key and virtual function OnDraw to handle the WM_PAINT message. The main program allocates an instance of a KHelloWindow class on the stack, creates a full-screen window, displays it, and goes into the normal message loop. So where is the “Hello, World”? The OnDraw function will display it during WM_PAINT message handling. We’ve just finished a C++ Windows program with our own class and not a single global variable.

Hello World Version 4: Drawing with DirectDraw

The second and third versions of “Hello, World” may look like the old-fashioned DOS programs that normally take over the whole screen, writing directly to the screen. Micro soft’s DirectDraw API, initially designed for high-performance game programming, allows programs to get even closer with a screen buffer and features offered by advanced display cards.

Here is a simple program using DirectDraw:

```
// Hello4.cpp
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <assert.h>
#include <tchar.h>
#include <ddraw.h>

#include "..\..\include\win.h"
```

```
const TCHAR szMessage[] = _T("Hello, World !");
const TCHAR szFace[]   = _T("Times New Roman");
const TCHAR szHint[]   = _T("Press ESC to quit.");
const TCHAR szProgram[] = _T("HelloWorld4");

// Copy CenterText from Hello2.cpp

class KDDrawWindow : public KWindow
{
    LPDIRECTDRAW     lpdd;
    LPDIRECTDRAWSURFACE lpddsprimary;

    void OnKeyDown(WPARAM wParam, LPARAM lParam)
    {
        if (wParam==VK_ESCAPE )
            PostMessage(m_hWnd, WM_CLOSE, 0, 0);
    }

    void Blend(int left, int right, int top, int bottom);

    void OnDraw(HDC hDC)
    {
        TextOut(hDC, 0, 0, szHint, lstrlen(szHint));
        CenterText(hDC, GetSystemMetrics(SM_CXSCREEN)/2,
                   GetSystemMetrics(SM_CYSCREEN)/2,
                   szFace, szMessage, 48);

        Blend(80, 560, 160, 250);
    }
public:
    KDDrawWindow(void)
    {
        lpdd      = NULL;
        lpddsprimary = NULL;
    }

    ~KDDrawWindow(void)
    {
        if ( lpddsprimary )
        {
            lpddsprimary->Release();
            lpddsprimary = NULL;
        }

        if ( lpdd )
        {
            lpdd->Release();
            lpdd = NULL;
        }
    }
}
```

```
        }

    }

    bool CreateSurface(void);
};

bool KDDrawWindow::CreateSurface(void)
{
    HRESULT hr;

    hr = DirectDrawCreate(NULL, &lpdd, NULL);
    if (hr!=DD_OK)
        return false;

    hr = lpdd->SetCooperativeLevel(m_hWnd,
        DDSCL_FULLSCREEN | DDSCL_EXCLUSIVE);
    if (hr!=DD_OK)
        return false;

    hr = lpdd->SetDisplayMode(640, 480, 32);
    if (hr!=DD_OK)
        return false;

    DDSURFACEDESC ddsd;
    memset(&ddsd, 0, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);
    ddsd.dwFlags = DDSD_CAPS;
    ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

    return lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL)
        ==DD_OK;
}

void inline Blend(unsigned char *dest, unsigned char *src)
{
    dest[0] = (dest[0] + src[0])/2;
    dest[1] = (dest[1] + src[1])/2;
    dest[2] = (dest[2] + src[2])/2;
}

void KDDrawWindow::Blend(int left, int right,
    int top, int bottom)
{
    DDSURFACEDESC ddsd;

    memset(&ddsd, 0, sizeof(ddsd));
    ddsd.dwSize = sizeof(ddsd);

    HRESULT hr = lpddsprimary->Lock(NULL, &ddsd,
```

```
DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL);
assert(hr==DD_OK);

unsigned char *screen = (unsigned char *)
ddsd.lpSurface;

for (int y=top; y<bottom; y++)
{
    unsigned char * pixel = screen + y * ddsd.lPitch
    + left * 4;

    for (int x=left; x<right; x++, pixel+=4)
        if ( pixel[0]!=255 || pixel[1]!=255 ||
            pixel[2]!=255 ) // non white
    {

        ::Blend(pixel-4, pixel);      // left
        ::Blend(pixel+4, pixel);      // right
        ::Blend(pixel-ddsd.lPitch, pixel); // up
        ::Blend(pixel+ddsd.lPitch, pixel); // down
    }
}

lpddsprimary->Unlock(ddsd.lpSurface);
}

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE,
                    LPSTR lpCmd, int nShow)
{
    KDDrawWindow win;

    win.CreateEx(0, szProgram, szProgram,
                WS_POPUP,
                0, 0,
                GetSystemMetrics( SM_CXSCREEN ),
                GetSystemMetrics( SM_CYSCREEN ),
                NULL, NULL, hInst);

    win.CreateSurface();
    win.ShowWindow(nShow);
    win.UpdateWindow();

    return win.MessageLoop();
}
```

What's shown above is a simple DirectDraw program that really draws directly into a screen display buffer. You may have noticed that we are reusing the KWindow class without repeating the code for window creation, simple message handling, and message loop.

With DirectDraw, we have more data to hold for each window in a KDDrawWindow instance, which is derived from the KWindow class, namely a pointer to a Direct Draw object and a pointer to a DirectDrawSurface object; both are initialized in the CreateSurface function. CreateSurface switches the screen resolution to 640 × 480 in 32-bpp mode and creates one primary DirectDraw surface. The destructor releases the interface pointers used. The OnDraw function displays a small hint message on the top-left corner of screen and a big blue “Hello World” message in the center of screen; both use normal GDI calls, just as in our previous example. What's different is the Blend routine called after that.

The KDDrawWindow::Blend routine starts by locking the screen display buffer, returning an address through which the screen buffer can be accessed directly. Before DirectDraw, direct access to the screen buffer was not possible using the GDI API, or even within GDI, because it's fully controlled by graphics device drivers which GDI replies on. For our example, which uses 32-bit-per-pixel mode, each pixel occupies 4 bytes. The pixel address is calculated using the following formula:

```
pixel = (unsigned char *) ddsd.lpSurface +  
y * ddsd.lPitch + x * 4;
```

The routine scans through a rectangle area of the screen top to bottom, left to right, searching for nonwhite pixels (the background color is white). When a nonwhite pixel is found, the pixel is blended with its neighbors on the left, right, top, and bottom directions. The blending sets the pixel value to the average of two pixels. [Figure 1-1](#) shows the result of blending 'Hello, World!' smoothly on a white background.

Figure 1-1. Blending text with background using DirectDraw.



Don't be discouraged if you've not touched the DirectDraw API yet; we will cover it in full detail [@Chapter 18](#).

1.2 ASSEMBLY LANGUAGE

People are much happier moving up the ladder, socially or even technically. So our profession has moved from machine code to C/Win32 API, to C++/MFC, to Java/AWT (Abstract Window Toolkit, classes for building graphics user interface in Java)/JFC (Java Foundation Classes, a new set of user interface classes which improve AWT), leaving only a few poor guys to implement the link backwards.

It's a good thing we can move forward in increasing productivity, but it's a sad thing that every time we move up the ladder, we quickly accept the new step as the only standard, and forget what's underneath. It's not strange nowadays to open a book on Visual C++ to find only pure MFC stuff inside or to hear questions like "How can I do this in MFC?"

Every time we add a layer of abstraction, a new layer of indirection is added. Someone has to implement each layer using the layers underneath, which is ultimately the assembly language. Even if you're not one of those people, having a deep understanding of assembly language gives you lots of advantages in your professional life. Assembly language helps you debug problems, understanding the working of the underlying OS (for example, just imagine that you got an exception in kernel32.dll). Assembly language helps you optimize your code to the highest performance; just try to understand why memory is implemented in such a complicated way. Assembly language exposes features of the CPU that are normally not accessible in high-level languages—for example, Intel's MMX (Multi-Media Extension) instructions.

We will focus on the Intel CPU in this book for assembly language, hopefully extending this to other CPUs in later editions. To gain basic knowledge of the Intel CPU, read a copy of the *Intel Architecture Software Developer's Manual*, which can be downloaded from the developer's home web page (developer.intel.com). From now on, we will assume you have at least some basic knowledge of the Intel CPU and its assembly language.

People usually say that a 16-bit program is running in 16-bit address mode and a 32-bit is running in 32-bit address mode. This may not be true on the Intel CPU; both 16-bit and 32-bit programs run in 48-bit logical address mode. For every memory access, you can have a 16-bit segment register and a 32-bit offset component. So the logical address is 48-bit. The Intel CPU does have a 16-bit module and a 32-bit module. In the 16-bit module, a segment is limited to 64 kB, and data and code pointers have a default 16-bit offset; in a 32-bit module, a segment is limited to 4 GB, and data and code pointers have a default 32-bit offset. But you can overwrite the bit-ness per instruction by adding a prefix (0x66 for operand, 0x67 for address). That's the trick to allow a 16-bit module to access 32-bit registers, or a 32-bit module to access 16-bit registers. A module in the Intel CPU term is not a process or an EXE/DLL module in the Windows world. A Windows EXE/DLL can have mixed 16-bit and 32-bit modules inside. If you're running Windows 95, check out dibeng.dll. It's a 16-bit DLL having several 32-bit segments inside to achieve 32-bit performance.

In terms of address, the difference between 16-bit and 32-bit code is that a 16-bit program normally uses the segmented model, where the whole address space is divided into several segments; a 32-bit program normally uses a flat address, where everything is in one 4 GB segment. For Win32 processes, the CPU CS (Code Segment), DS (Data Segment), SS (Stack Segment) and ES (Extra Segment) segment registers are mapped to the same virtual address 0. The benefit is that we can easily generate a few lines of machine code in a data array and call it as a function. To do that in Win16 code requires mapping a data segment into a code segment, using the latter plus the offset to access the code in the data segment. Because all the four major segment registers are mapped to the same virtual address 0, a Win32 program normally uses only a 32-bit offset as the full address. But at assembly level, the segment register can still be used together with an offset to form a 48-bit address. For example, the FS

segment register, which is also a data segment register provided by the Intel CPU, does not map to virtual address 0. Instead, it maps to the starting point of a per-thread data structure maintained by the operating system, through which the Win32 API routine accesses important per-thread information, such as the last error code (through SetLastError/GetLastError), exception handler chain, thread local storage, etc.

At the assembly-language level, the Win32 API uses a standard calling convention which passes parameters on the stack from right to left. So from the Windows message dispatcher, a call to a window message handle such as:

```
unsigned rslt = WindowProc(hWnd, uMsg,  
                           wParam, lParam);
```

would be:

```
mov  eax, lParam  
push eax  
mov  eax, wParam  
push eax  
mov  eax, uMsg  
push eax  
mov  eax, hWnd  
push eax  
call WindowProc  
mov  rslt, eax
```

Of the Intel Pentium series CPU features that are not accessible from C/C++, one instruction is very interesting to performance-minded programmers. That's the RDTSC (Read Time Stamp Counter) instruction, which returns the number of clock cycles passed since the booting of the CPU in 64-bit unsigned integer, through the EDX and EAX 32-bit general register pair. This means you can time your program in 5-nanosecond precision on a Pentium 200-MHz machine for 117 years.

As of now, the RDTSC instruction is not even supported by the Visual C++ inline assembler, although the code optimizer seems to understand that it modifies EDX and EAX registers. To use the instruction, insert its machine-code form 0x0F, 0x31. Here is our CPU clock cycle stopwatch class using RDTSC instruction:

```
// Timer.h  
#pragma once  
  
inline unsigned __int64 GetCycleCount(void)  
{  
    _asm  _emit 0x0F  
    _asm  _emit 0x31  
}  
  
class KTimer  
{  
    unsigned __int64 m_startcycle;
```

```
public:  
  
    unsigned __int64 m_overhead;  
  
    KTimer(void)  
    {  
        m_overhead = 0;  
        Start();  
        m_overhead = Stop();  
    }  
  
    void Start(void)  
    {  
        m_startcycle = GetCycleCount();  
    }  
  
    unsigned __int64 Stop(void)  
    {  
        return GetCycleCount()-m_startcycle-m_overhead;  
    }  
};
```

The KTimer class stores timing information in a 64-bit unsigned integer, because the 32-bit version lasts less than a second on a 200-MHz machine. The Get Cycle Count function returns a 64-bit unsigned integer representing the current CPU clock count. The result generated by the RDTSC instruction matches the C 64-bit function return value convention. So GetCycleCount is merely a single machine instruction. The Start function reads the starting clock count; Stop reads the end clock count and returns their difference. To be more accurate, we have to compensate for the time used by the RDTSC instructions and store its reading. The KTimer class constructor does this by starting and stopping the timer once to measure the amount of overhead, which can then be deducted later.

Here is a sample program that uses KTimer to measure your CPU clock speed and the time it takes to create a solid brush:

```
// GDISpeed.cpp  
#define STRICT  
#define WIN32_LEAN_AND_MEAN  
  
#include <windows.h>  
#include <tchar.h>  
  
#include "..\..\include\timer.h"  
  
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE,  
                    LPSTR lpCmd, int nShow)  
{  
    KTimer timer;  
    TCHAR mess[128];
```

```
timer.Start();
Sleep(1000);
unsigned cpuspeed10 = (unsigned)(timer.Stop()/100000);

timer.Start();
CreateSolidBrush(RGB(0xAA, 0xAA, 0xAA));
unsigned time = (unsigned) timer.Stop();

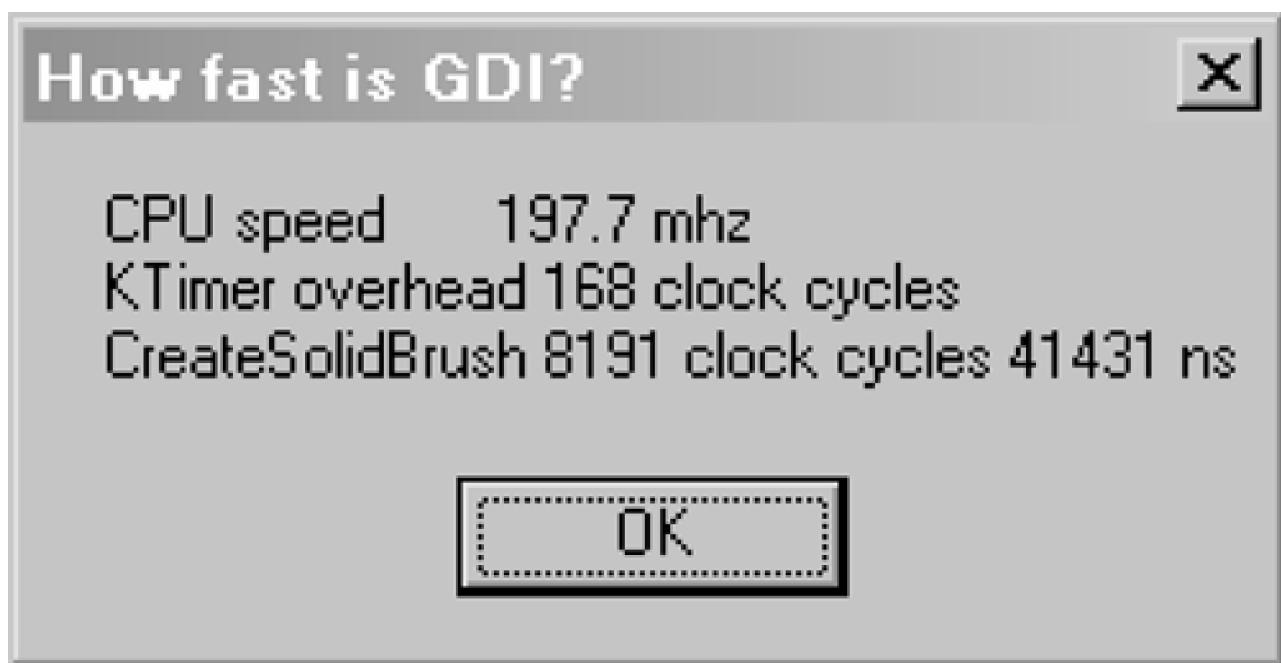
wsprintf(mess, _T("CPU speed      %d.%d mhz\n")
         _T("KTimer overhead %d clock cycles\n")
         _T("CreateSolidBrush %d clock cycles %d ns"),
         cpuspeed10 / 10, cpuspeed10 % 10,
         (unsigned) timer.m_overhead,
         time, time * 10000 / cpuspeed10;

MessageBox(NULL, mess, _T("How fast is GDI?"), MB_OK);

return 0;
}
```

The measurement result is displayed in a message box, as shown in [Figure 1-2](#). Now we know with confidence that creating a solid brush costs about 40 microseconds on a 200-MHz Pentium CPU system.

Figure 1-2. Using the Intel RDTSC instruction to measure performance.



1.3 PROGRAM DEVELOPMENT ENVIRONMENT

To develop programs for Microsoft Windows effectively, you need a set of machines, tools, books, documents, training courses, on-line resources, and friends. While so much stuff may seem to be overwhelming, you need only a few basic pieces to get started; the rest is meant to enable you to do more advanced programming or help you to be more effective.

Compared with the old DOS program development environment, or the 16-bit Windows development environment, the 32-bit Windows programming environment has advanced greatly.

Development and Testing Systems

A minimum development system nowadays may be a 200-MHz Pentium with 64 MB of RAM, a CD ROM drive, 8 GB of hard disk space, and a network link.

You need high CPU clock speed to reduce the time spent waiting for the compiler and linker to turn your source code into binary code, unless you want an excuse to take frequent coffee breaks or check your stocks. An Intel Pentium CPU or compatible is really the minimum CPU to have to take advantage of the more advanced instructions—for example, the RDTSC. It's much better to have Pentium Pro, Celeron, Pentium III CPU, or compatibles, which provide more instructions, more hardware optimization, and a better, larger memory cache. If you or your boss can afford it, get a machine with dual CPUs. You need at least that to make sure your program does not slow down on dual-CPU machines—for example, because of having multiple threads allocating from the same system heap.

RAM size may be more important than CPU speed. If you're working with a 16-MB or 32-MB system, your system RAM is constantly full. The CPU is spending lots of its clock cycle swapping out an inactive page, swapping in an active page, making the whole system run at disk speed.

Hard disk space is also important as compilers, the Software Development Kit (SDK), and the Driver Development Kit (DDK) grow in size. The development tools come with large help files, include files, and sample source code. The compiler generates huge files for pre-compiled header files, debugger symbols files, and browser databases. You can easily go to web pages to download tons of tools and documents. Besides all these fixed usages of the hard disk, the operating system is swapping RAM with the hard disk.

A network connection is required for accessing, sharing, or backing up information. Micro soft's kernel-level debugger WinDbg requires two machines linked with a serial cable or network cable. Version control systems or defect tracking systems usually run over a network.

If you're using Microsoft kernel debugger WinDbg to debug kernel mode drivers, you need two machines, one for running the program under testing and one for running the debugger program. [Chapter 3](#) of this book will show you how can you make use of WinDbg debugger extension DLL on a single machine to examine Windows NT/2000 GDI internal data structures.

Windows NT 4.0 or Windows 2000 may be a good operating system to install on your development system compared with Windows 95 or Windows 98. Microsoft Windows NT 4.0/2000 is getting more and more mature as a premium operating system, more books are published on it, and more tools are available.

Beside development system(s), you also need testing systems or access to them. Your products normally target Windows 95/98/NT/2000 users. So you have to make sure they work perfectly on all of them. For software whose performance is critical, testing systems with different configurations are also important. You may double the speed of a piece of code on a Pentium II CPU, which actually runs slower on a Pentium CPU that is several years old.

Compilers

Once you've got a development system, you need a compiler to convert your program in human-readable form to CPU-preferred form. A compiler now is much more than a bare-bones compiler. It's an integrated development environment with a syntax highlighting editor, header files, library source code and binary files, help files, a compiler, linker, a source-level debugger, wizards, components, and all the other tools.

If you're doing development solely at the Win32 API level, you're lucky to have several choices. You can use Microsoft Visual C++, Borland C++, Delphi, or Microsoft Visual Basic. But if you want to do development with the Windows NT/2000 DDK, you don't have much choice because a Microsoft Visual C++ compiler is assumed.

There are several current versions of Microsoft Visual C++ compilers. You should normally upgrade to the latest version with the latest service pack installed to take advantage of new features, enhancements, and defect fixes, unless your organization decides to delay the upgrades because of incompatibility or risks. Sample code in this book is compiled with Microsoft Visual C++ version 6.0. If you want to compile with earlier versions, the workspace and project files may give you a few warnings.

Along with the operating system and compiler come tools you normally would not notice unless someone mentioned them to you.

Windows NT/2000 debug symbols can be very useful in understanding how the system works, chasing defects which manifest only in system code, or just appreciating internal symbol names used in Microsoft source code. On VC6.0 CD, Microsoft puts a few debug symbol files in the \vc60\vc98\debug directory, for a few important system DLLs like gdi32.dll, user32.dll, etc. For a complete set of debug symbol files, turn to your Windows NT/2000 CD and check under the \support\debug directory. On newer versions of Windows 2000, the size of these debug symbols grows so much that Microsoft has to put them on a separate support-tools CD. When installing debug symbol files, check the CPU type (i386 or alpha), build type (checked or free), and build number. They need to exactly match the OS you're testing. Having debug symbols for system files is enough reason for moving your development system from Windows 95/98 machines to Windows NT/2000 machines. Now you can use the Visual C++ debugger context-sensitive menu option "Go To Disassembly" more often and not get lost. For example, if you set a breakpoint on CreateSolidBrush and choose to go to assembly mode when it hits, Visual C++ debugger will show the symbolic name _Create SolidBrush@4, instead of the unreadable hex-decimal address. So it's a function accepting 4 bytes of parameters, or just a single 32-bit value. A few lines below, you will see a call to _CreateBrush@20, a routine accepting 5 parameters. Generalization: GDI is merging different API calls into a more uniform call. Chasing into it will lead to _NtGdiCreateSolidBrush@8, which invokes a software interruption, 0x2E. If you want to step into the interruption routine, Visual C++ debugger will not allow that. GDI has to call kernel mode code in win32k.sys to create the solid brush. You don't really need to be an assembly-language expert to learn all these; the debug symbols are like lighthouses guiding your journey. [Figure 1-3](#) illustrates how the Visual C++ debugger uses debug symbol files.

Figure 1-3. Debug symbol files guide your debugging.

The screenshot shows a debugger's disassembly window with the title "Disassembly". It displays assembly code for two functions:

- CreateSolidBrush@4:**

```
77F42059 xor eax, eax
77F4205B push eax
77F4205C push eax
77F4205D push eax
77F4205E push dword ptr [esp+10h]
77F42062 push eax
77F42063 call _CreateBrush@20 (77f41f9b)
77F42068 ret 4
```
- NtGdiCreateSolidBrush@8:**

```
77F4206B mov eax, 102Ah
77F42070 lea edx, [esp+4]
77F42074 int 2Eh
77F42076 ret 8
```

At the bottom of the window, there is a partial view of another function: **NtGdiCreateStockObject@4**.

There are two ways Microsoft compilers store debug information. The old format uses a single .dbg file per module. The newer format uses two files, one .dbg, and another .pdb. The .dbg file is referenced by the module's binary file. For example, gdi32.dll references "dll\gdi32.dbg" in its debug directory, which tells the debugger to load \$SystemRoot\$\symbols\dll\gdi32.dbg.

Debugger is not the only tool that understands how to use debug symbols. In fact, it does not even know how to do that itself. Debugger uses the image help API to load and query debug information, a simple modular design. Anyone using the image help API can have the same access; this includes the dumpbin utility or even your own program.

Dependency walker (depends.exe) lists all the DLLs your module imports implicitly in a nice recursive way. It helps you to get a clear understanding of how many modules will be loaded into your program when it's running, how many functions are imported. Check this on a simple MFC program; you will be amazed how many functions are imported without your knowledge. You can use it to check if a program will run on the initial version of Windows 95, which does not have system DLLs like urlmon.dll or provide so many exported functions in ole32.dll.

Microsoft Spy++ (spyxx.exe) is a nice, powerful tool to spy on Windows, messages, processes, and threads. If you ever wonder how common dialog boxes, like the File Open dialog box, are structured during runtime, or why a certain message you're expecting is not sent, it's worth a try.

WinDiff (windiff.exe) is a simple, useful tool to find out what has been changed between two versions of a source file or a source code tree. It also helps to check the difference between localized versions of resource files and the original versions.

Pstat.exe is a tiny text-based program listing lots of information about the processes, threads, and modules running on your system. It lists all the processes and threads in the system, giving the time spent on their user mode and kernel mode code, working set, and page fault count. In the final part of list, pstat shows all the system DLLs and device drivers loaded into kernel mode address space, with their name, address, code and data size and version string. For that, please note that GDI engine implementation win32k.sys is loaded to address 0xa0000000; your display driver is loaded to address 0xbef7000, etc.

Process Viewer (pview.exe) is another little tool showing process-related information using a graphical user

interface. For example, it shows how much RAM is allocated for each module within a process.

Other tools include dumpbin.exe for the exam PE file format, profile.exe for simple performance profiling, nmake.exe for compile using a makefile, and rebase.exe for changing the base address of a module to avoid expensive runtime reallocation.

Sometimes you will be forced to look at something that sounds trivial and un attractive: the header files. Someone may say header files are really not designed for human comprehension, but rather for machine consumption. Yes, you're right. But the machine is very precise and stubborn. It follows the header files to the exact word without knowing what online-help or books may have told programmers. There are cases where the documentation is wrong, so that you have to turn to the header file for the final answer. For example, check how TBUTTON is defined from your on-line help. It's a structure with 6 fields, right? If you define the TBUTTON constant structure using 6 fields each, the compiler will give you an error message. Check the header file commctrl.h in the Win32 API; TBUTTON has an extra two-byte reserved field, making it a structure with 7 fields.

You can also use header files to learn how macro STRICT affects compilation, how the Unicode and non-Unicode code versions of API get mapped to exported functions, how much new stuff is added by Windows 2000, etc.

If you can read header files, source code will definitely make reading more interesting. The C runtime library source is a must-read to understand how your module gets started and shuts down, and how malloc/free and new/delete do suballocation from the system heap. You also get to know why the inline version of memcpy may be slower than the function-call version in certain cases. The MFC source code is something very interesting to read and step into. The message-handling part, which bridges C++ with C-based Win32 SDK, is a must-read. The ATL (Active Template Library) source code contrasts sharply with the MFC source code. The CWndProcThunk class, where a few machine instructions are used to jump from C world into C++ world, is also a must-read.

Within the proper Microsoft Visual Studio, there are some better-known useful features. Under the file menu, you can use the studio to read an html page with syntax highlighting and to open an executable to browse through its resources. Under the edit menu, you have lots of text-search options; you can set different kinds of breakpoints with a location, data access, or window message. Under the project menu, you can choose to generate an assembly listing or to link debug information in the release build. Under the debug menu, you can choose to stop at certain exceptions when they occur and check the loaded module list.

Microsoft Platform Software Development Kit

Microsoft Platform Software Development Kit (SDK) is a collection of SDKs to enable program development to interface with current and emerging Microsoft technologies. It's the successor to the Win32 SDK and includes BackOffice SDK, Active X/Internet Client SDK, and the Direct X SDK. Best of all, Microsoft Platform SDK is free and regularly gets updated. Microsoft Platform SDK and other SDKs were last spotted at the following address:

<http://msdn.microsoft.com/downloads/sdks/platform/platform.asp>

So what's in SDK really? Platform SDK is actually a huge collection of header files, library files, online documents, tools, and sample sources. If you already have a compiler like Microsoft Visual C++, you can still benefit from the latest platform SDK. For example, your compiler may not have the latest header file and libraries allowing you to use the new APIs added to Windows 2000. Downloading the Platform SDK and telling the compiler to use the new header files and libraries will solve the problem.

We can say that Visual C++ studio is a nicely packaged set of tools to develop Win32 programs using the Microsoft Visual C++ compiler, while Platform SDK is an extensive collection of tools to develop Win32 programs using an

external C/C++ compiler.

Visual C++ studio centers around a C++ compiler, with its C/C++ runtime libraries, ATL and MFC. Platform SDK does not provide a compiler, header files for C/C++ functions, or libraries for C/C++ runtime library functions. This allows third parties to provide an alternative C/C++ compiler, C/C++ header files, and library, giving programs a development environment other than Microsoft's solution—even a Pascal development environment, if you can change the Windows API header files to Pascal form. To compile any program with Platform SDK, you must have a compiler and minimum C runtime library first.

Platform SDK comes with dozens of tools, some of which have already been added to Visual Studio. The qgrep.exe program is a command-line text-searching tool, similar to Visual Studio's "Find in Files" button. The API monitor (apimon.exe) is a quite powerful Windows API monitoring tool. It works like a special-purpose debugger. It loads a program, intercepts calls to the Windows API, records the API call events with time counters, and traces API calls with parameters and return values. Once something goes wrong, the API monitor pops up a DOS window, allowing you to disassemble code, examine registers, set breakpoints, and single-step through your code. The memsnap.exe program gives a quick snapshot of memory usage for running processes. It displays working set size and pooled and nonpooled kernel memory usage. The process walker (pwalk.exe) displays a detailed breakdown of a process's virtual memory usage in user address space. It shows how the user address space is broken down into hundred of regions, giving their state, size, size, section, and module name. You can double-click a memory region to get a hexadecimal dump of the region. The sc (sc.exe) program provides a command-line interface to the service control manager.

The object viewer (winobj.exe) is a very useful program. It allows you to browse through all kernel objects currently active in the system in a tree structure, which includes events, mutexes, pipes, memory-mapped files, device drivers, etc. For example, you can find PhysicalMemory listed under device. So there is a device driver allowing you to access physical memory. You can use the following line to create a handle to physical memory:

```
HANDLE hRAM = CreateFile("\\\\.\\PhysicalMemory",...);
```

The most exciting place in Platform SDK is the sample source-code repository, if you don't mind reading some old-fashioned Windows programs written in C. You won't find C++, MFC, ATL, or heavy usage of C runtime functions. Even the COM and DirectX programs are written in C using function pointer tables, not in C++ using virtual function. Also included is the complete source code for quite a few SDK tools like windiff and spy. Keep in mind, while reading those programs, that they were first written in the early 90s; some started as Win16 API programs. There are lots of places where you would say they show bad programming habits, like lack of code reuse, overuse of global variables, little error checking, and heavy influence of the Win16 API. But still, there are lots of things to learn from the source code. [Table 1-1](#) lists some programs that may be related to what we are talking about in this book.

Oh, we forgot to mention the single most useful tool in Platform SDK: a full-featured multiwindow source-level debugger, the WinDbg. Unlike the debugger that comes with Visual Studio, WinDbg can be used both as a user program debugger and a kernel mode debugger for Windows NT/2000. If you want to use it as kernel mode debugger, you need to set up two machines, linked with serial or network cables. WinDbg also allows examination of crash dumps. We will talk more about WinDbg in [Chapter 3](#).

Speaking of tools, Numega provides quite a few professional tools. Bounds Checker verifies API calling and detects memory and resource leaks. SoftICE/W is an amazing system-wide debugger, allowing you to debug both user mode and kernel mode code, 16-bit and 32-bit code, on a single machine. With it, you can step from a user mode program into kernel mode code and back to a user mode program. TrueTime is a performance profiler that helps you identify performance bottlenecks in your code. Intel's vTune and C++ compiler is for anyone who wants to fine-tune

performance and take advantage of Intel MMX or SIMD (Single Instruction Multiple Data) instruction extensions.

Microsoft Driver Development Kit

Microsoft Visual C++ and Platform SDK allow you to write normal user-level programs, programs like WordPad or even Microsoft Word. To make the operating system work, especially with tons of devices like hard disks, display cards, and printers, you need another kind of program: device drivers. Device drivers are loaded into kernel address space. Where Win32 API functions are not available, the Win32 API data structures are also gone. Replacing them are kernel system calls and device driver interfaces. To write Windows device drivers, you need the Microsoft Driver Development Kit, which can be freely downloaded from Microsoft, including DDKs for Windows 95/98/NT4.0/2000. Here is the address:

<http://www.microsoft.com/ddk/>

Table 1-1. Sample Source Code of Interest in Platform SDK

Path to Program	Reason to Read
\graphics\directx	2 MB of DirectX sample programs
\graphics\gdi\complexscript	Display text of complex script: Arabic, Hebrew, and Thai
\graphics\gdi\fonts	Extensive usage of font API
\graphics\gdi\metafile	Enhanced metafile, loading, displaying, editing, and printing
\graphics\gdi\printer	Printing API, lines, pens, brushes
\graphics\gdi\setdisp	Dynamic screen-resolution switching
\graphics\gdi\showdib	Device-independent bitmap handling
\graphics\gdi\textfx	Shows path to create text effects
\graphics\gdi\wincap32	Screen capture, hooks
\graphics\gdi\winnt\plgblt	Demonstrates PlgBlt call
\graphics\gdi\winnt\wxform	Demonstrates world transformation
\graphics\gdi\video\palmap	Video (DIBs) format translation
\sdktools\aniedit	Animated cursor editor
\sdktools\fontedit	Bitmap font patcher
\sdktools\image\drwatson	Dr. Watson error trapper. It demonstrates symbol-table access, Intel instruction disassembling, simple debugger, access to process list, stack walking, crash dumping, etc.
\sdktools\imageedit	Simple bitmap editor
\sdktools\winnt\walker	Process virtual memory space walker
\winbase\debug\deb	Sample Win32 debugger
\winbase\debug\wdbgexts	Sample WinDbg debugger extension
\winbase\winnt\service	Service API

Note

This definitely is *not* a book on device driver development, for which there are several good books on

the market. But those books normally focus on the mainstream device driver development, like IO driver and file-system driver. This is a book on Windows graphics programming, a book with depth. We want to explore how GDI/DirectDraw calls are implemented in the GDI engine and finally passed to device drivers. As such, display drivers (including DirectDraw support), font drivers, and printer drivers are relevant to our topics. The kernel device driver is also a good way to by-pass the user mode API system to do something that cannot be done in the Win32 API. [Chapter 3](#) discusses how a simple kernel mode driver can help you to understand the GDI engine.

Similar to Platform SDK, a DDK is a huge collection of header files, library files, online documents, tools, and sample driver source code. The header files include both Win32 API header files and device driver header files. For example, wingdi.h documents the Win32 GDI API, while winddi.h documents the interface between the GDI engine and display or printer drivers. For DirectDraw, ddraw.h documents the DirectDraw API, while ddrawint.h specifies Windows NT DirectDraw driver interface. The library files are classified into free build or checked build, the two flavors of OS builds. Here you can find import libraries for system kernel DLLs such as win32k.lib for win32k.sys. The help directory contains detailed specifications of device driver interfaces and device driver development guidelines. The sample driver source-code directory is definitely a gold mine. You can find 2 MB of source code for the s3 VirGE display driver with GDI, DirectDraw, and DirectDraw 3D support in the \src\video\displays\s3virge directory.

DDK needs a C compiler to generate binary code, similar to Platform SDK. To make the generation of drivers easier, DDK provides a project building tool (build.exe) that builds a whole hierarchy of source code with different flavors. If you have the source, build.exe can probably build the whole OS from a command line. Building device drivers needs some special compile and linker options that are not handled by Visual C++ Studio. So the most convenient way to build a device driver is still the command-line build tool.

With DDK come more tools. The command-line program break.exe allows you to attach a debugger to a process. Debugger wizard (dbgwiz) helps configure WinDbg. Global Flags (gflags) lets you change dozens of system flags. For example, you can enable pool and heap tagging, marking memory allocation with its owner to identify memory leakage. Pool monitor (poolmon.exe) monitors kernel memory allocation/deallocation if pool tagging is enabled. Registry dumper (regdmp) dumps the registry to a text file.

Microsoft Development Network Library

The Microsoft Development Network (MSDN) Library is a massive collection of references for Microsoft Windows developers. It contains gigabytes of documentation, technical articles, knowledge-base articles, sample source code, magazine articles, books, specifications, and anything else you can imagine you may need to develop solutions for Microsoft Windows. It contains documentation for Platform SDK, DDK, Visual C++, Visual Studio, Visual Basic, Visual J++, etc.

The MSDN Library is virtually everything Microsoft wants you to know to write more code for its operating systems. A newer version of Microsoft Visual Studio uses the MSDN Library as a help system, which indirectly calls for upgrading your hard disk and adding a CPU to your system. As far as Windows graphics programming is concerned, [Table 1-2](#) lists the parts of the MSDN Library that are closely related.

With the MSDN library CDs so easily accessible and with a searchable MSDN Library web page (msdn.microsoft.com), every book on Windows Programming needs to justify why readers need to buy it. This book

does not compete with the massive information provided by the MSDN Library; it builds on top of the MSDN Library. In this book, you will not see pages of Win32 or DDK API specifications that you can find in the MSDN Library. The first part of this book tries to show how GDI and DirectDraw are implemented by drawing the missing link between the Win32 graphics API and the DDK graphics driver interface. The second part of this book tries to explain GDI and the DirectDraw API with more much concrete and precise information. We will refer to the MSDN Library from time to time. If you're reading the book without access to the MSDN Library, you may consider printing the sections of the MSDN Library mentioned above.

Table 1-2. Graphics Programming Related Parts in MSDN Library

Platform SDK\Graphics and Multimedia Services\Microsoft Direct X
Platform SDK\Graphics and Multimedia Services\GDI
DDK Documentation\Windows 200 DDK\Graphics Drivers

Table 1-3. Graphics Programming Related Articles from MSDN Library

Specifications\Applications\True Type Font Specifications
Specifications\Platforms\Microsoft Portable Executable and Common Object Form Specification
Specifications\Technologies and Languages\The UniCode Standard, Version 1.1
Technical Articles\Multimedia\Basics of DirectDraw Game Programming
Technical Articles\Multimedia\Getting Started with Direct3D: A Tour and Resource Guide
Technical Articles\Multimedia\Texture Wrapping Simplified
Technical Articles\Multimedia\GDI*.* (dozens of good articles)
Technical Articles\Windows Platform\Memory\Give Me a Handle, and I'll Show You an Object
Technical Articles\Windows Platform\Windows Management\Window Classes in Win32
Backgrounders\Windows Platform\Base\The Foundations of Microsoft Windows NT System Architecture

Besides the three big chunks of SDK/DDK documents, the MSDN Library provides lots of useful information regarding Windows Graphics programming in technical articles, knowledge-base articles, specifications, and other forms. While reading them, it's important to check the date they were written and the platform they were written for. You will find useful information mixed together with out-of-date junk. [Table 1-3](#) provides a partial reading list.

1.4 WIN32 EXECUTABLE FILE FORMAT

Lots of people may remember *Algorithms + Data Structures = Programs*, the title of a book written by N. Wirth, the father of the Pascal language family, of which Delphi is a modern member. But we should also know that compiled binary code is a data structure in itself, which the operating system operates on when code is loaded into memory for execution. For Win32 platforms, this data structure is called the Portable Executable, or PE, file format.

Understanding the PE file format helps greatly with Windows programming. It helps you understand how source code is turned into binary code, where global variables are stored, and how are they initialized, including how shared variables work. Every DLL in the Win32 system is in the PE format. So understanding the PE format helps you understand how dynamic linking works, how import references are resolved, and how to avoid dynamic rebasing of DLLs. The basic technique of API hooking depends heavily on knowledge of import table details. Understanding the PE format also helps you understand how virtual memory space is structured in the Win32 environment. There are a few places where knowledge of the PE file format will be needed in this book, so we briefly discuss the PE file format and its loaded form in RAM here.

Programmers write source code in C, C++, assembly code, or other languages, which are translated by the compiler into object files in .OBJ file format. Each object file contains global variables, initialized or uninitialized, constant data, resources, execution code in machine language, symbolic names for linking, and debugging information. The object files of a module are linked by a linker with several libraries, which are themselves a combined form of object files. The commonly used libraries are C/C++ runtime libraries, MFC/ATL libraries, and import libraries for Win32 API or Windows kernel services. The linker resolves all the interreferences among the object files and libraries. For example, if your code calls a C++ library function “new,” the linker finds the address of “new” in the C++ runtime library and puts it into your code. After that, the linker merges all initialized global variables into one section, all un initialized variables into another section, all execution code into yet another section, etc.

The reason why different parts of object files are grouped into different sections is twofold: protection and optimal usage of resources. Constant data and executable code are normally read-only. It helps the programmer to detect program errors if the operating system can detect any attempt to write to these portions of memory. In order to do so, the linker marks constant data and executable code sections as read-only. The executable code section needs to be marked as executable. Global variable sections, initialized or uninitialized, certainly would be readable/writeable. The Windows operating system code uses DLLs heavily. All Win32 programs having a graphics user interface use gdi32.dll, for example. To make optimal usage of RAM, the executable code section of gdi32.dll is stored in RAM only once in the whole system. Different processes get a view of its code through a memory-mapped file. This is possible because the execution code is read-only, which means it should be the same in all processes. Global data can't be shared among processes unless they are specially marked as shared data.

Each section has a symbolic name, which can be referred to in linker options. Code or data in a section shares the same attributes. Memory is allocated in pages to a section. So the minimum runtime cost for a section is one page, which is 4 kilobytes on the Intel CPU. [Table 1-4](#) lists the commonly seen sections.

Table 1-4. Common Sections in the PE File

Name	Contents	Attributes
.text	Execution code	Code, Execute, Read
.data	Initialized Global Data	Initialized Data, Read, Write
.rsrc	Resource	Initialized Data, Read Only
.bss	Uninitialized Global Data	Read, Write
.rdata	Constant Data	Initialized Data, Read Only
.idata	Import Directory	Initialized Data, Read, Write
.edata	Export Directory	Initialized Data, Read Only
.reloc	Relocation Table	Initialized Data, Discardable, Read Only
.shared	Shared Data	Initialized Data, Shareable, Read, Write

Execution code and global data do not have so much structure within themselves in the PE file. No one really wants to help hackers to hack into his/her code. But for other information, the operating system needs to do some searching from time to time. For example, when a module is loaded, the loader needs to search the import function table to patch up the address of imported functions; when a user calls GetProcAddress, a search of the export function table is needed. The PE file reserves 16 so-called directories for these kinds of purposes. The commonly seen directories are import, bound import, delayed import, export, relocation, resource, and debug information.

Combine the sections and directories together, add a header or two, and we have a PE file, as illustrated in [Figure 1-4](#).

Figure 1-4. Portable Executable File Layout.

IMAGE_DOS_HEADER

DOS Stub Program

IMAGE_NT_HEADER

'PE' File Signature

IMAGE_FILE_HEADER

IMAGE_OPTIONAL_HEADER R32

Section Table

IMAGE_SECTION_HEADER []

.text section for binary code

.data section for initialized data

.reloc section for relocation table

.rdata section for constants

.rsrc section for resource

• • •

A PE file starts with a DOS .EXE file header (IMAGE_DOS_HEADER structure), because Microsoft wants to make sure you can still run a Win32 program in a DOS box. Immediately following the IMAGE_DOS_HEADER is a tiny DOS program, which generates a software interruption just to print an error message, followed by a program exit.

Following the stub program is the real PE file header (IMAGE_NT_HEADERS). Note that the stub program is not of fixed length, so to find the offset of the IMAGE_NT_HEADERS structure, use the e_lfanew field in IMAGE_DOS_HEADER. The IMAGE_NT_HEADERS structure starts with a 4-byte signature that should equal IMAGE_NT_SIGNATURE. If not, the file could be an OS/2 file or a VxD file. The IMAGE_FILE_HEADER structure stores the machine identifier of the target CPU, the number of sections in the file, time-stamp of the build, pointer to the symbol table, and size of the optional table.

The IMAGE_OPTIONAL_HEADER structure is not really optional. You will see it in every PE file, because it's too important to be omitted. The information stored there includes the suggested base address of the module, sizes of code and data, base addresses of code and data, heap and stack configuration, OS version and subsystem requirements, and the table of directories.

A PE file stores endless addresses to reference functions, variables, names, tables, etc. Some of them are stored as virtual addresses, which could be used as an address directly once the module is loaded into memory. If the module can't be loaded into the suggested base address, the loader will patch the data to make it right. But most addresses are stored as relative to the start of a PE file header. Such an address is called an RVA (relative virtual address). Please note that an RVA is not an offset within a PE file before it's loaded into RAM. The reason is that the PE file normally packs all sections using 32-byte alignment, while the OS uses CPU page alignment. For the Intel CPU, each page is 4096 bytes. The RVAs are calculated assuming sections are page-aligned to minimize runtime cost.

The following is a simple C++ class to handle simple tasks in dealing with Win32 modules loaded into memory. The constructor shows how to get pointers to IMAGE_DOS_HEADER and IMAGE_NT_HEADERS structures. The GetDirectory routine shows how to get a pointer to a directory's data. We will continue to develop this class to make it more useful.

```
class KPEFile
{
    const char      * pModule;
    PIMAGE_DOS_HEADER pDOSHeader;
    PIMAGE_NT_HEADERS pNTHHeader;

public:
    const char * RVA2Ptr(unsigned rva)
    {
        if ( (pModule!=NULL) && rva )
            return pModule + rva;
        else
            return NULL;
    }
}
```

```
KPEFile(HMODULE hModule);

const void * GetDirectory(int id);

PIMAGE_IMPORT_DESCRIPTOR GetImportDescriptor(LPCSTR pDIIName);

const unsigned * GetFunctionPtr(PIMAGE_IMPORT_DESCRIPTOR
    plImport, LPCSTR pProcName);

FARPROC SetImportAddress(LPCSTR pDIIName, LPCSTR pProcName,
    FARPROC pNewProc);

FARPROC SetExportAddress(LPCSTR pProcName, FARPROC pNewProc);
};

KPEFile::KPEFile(HMODULE hModule)
{
pModule = (const char *) hModule;

if ( IsBadReadPtr(pModule, sizeof(IMAGE_DOS_HEADER)) )
{
    pDOSHeader = NULL;
    pNTHHeader = NULL;
}
else
{
    pDOSHeader = (PIMAGE_DOS_HEADER) pModule;

    if ( IsBadReadPtr(RVA2Ptr(pDOSHeader->e_lfanew),
        sizeof(IMAGE_NT_HEADERS)) )
        pNTHHeader = NULL;
    else
        pNTHHeader = (PIMAGE_NT_HEADERS) RVA2Ptr(pDOSHeader->
            e_lfanew);
}
}

// returns address of a PE directory
const void * KPEFile::GetDirectory(int id)
{
    return RVA2Ptr(pNTHHeader->OptionalHeader.DataDirectory[id].
        VirtualAddress);
}
```

Now that we have a basic conceptual understanding of the PE file format, let's look at a few real applications instead of covering all the details.

Import Directory

When you use a Win32 API function, such as LoadLibraryW, the binary code generated looks like:

```
DWORD __imp__LoadLibrary@4 = 0x77E971C9;  
call dword ptr[__imp__LoadLibraryW@4]
```

The strange thing is that the compiler introduces an internal global variable here, using an indirect instead of a direct call instruction. The compiler does have good reason to do this. The linker does not know the address of LoadLibraryW@4 for certain during linking time, although it can put in a guess based on one version of kernel 32.dll (specified in the bound import information directory). So most of the time, the module loader needs to find the right address of the imported function and patch the module image. There may be more than one call to the same function like LoadLibraryW. For performance reasons, the loader would prefer to patch the least number of places, which means one per imported function. This introduces one location to store the address of one import function, which normally has the internal name form __imp__xxx. Import function addresses can either be in a separate section, normally named as .idata, or merged with .text section to save space.

One module normally imports from multiple modules, with multiple functions imported from each module. In the PE file, the import directory points to an array of IMAGE_IMPORT_DESCRIPTOR structures, each corresponding to one imported module. The first field of IMAGE_IMPORT_DESCRIPTOR holds an offset to the hint/name table. Its last field holds an offset to the import address table. The two tables are of the same length, with each entry corresponding to one imported function.

The hint/import table entry could contain an ordinal value if the most significant bit is on, or an offset to a 16-bit hint value followed by the name of the imported function otherwise. So clearly, the hint/name table can be used to search the export directory of the module we're importing from.

In the original PE file, the import address table could contain the same information as the hint/name table, offsets to hint followed by the function name. In this case, the loader finds the address of the imported function and patches the import address table entry with the address. So when the PE file is loaded, the import address table is really a table of imported function addresses. The linker could also bind the module with a certain DLL to initialize the table with addresses of imported functions from a certain version of DLL. In the latter case, the import address table is a table of bound import function addresses. In both cases, the import function table contains internal variables like __imp__LoadLibrary@4.

Now let's try to implement KPEFile::SetImportAddress. It is supposed to change the address of an imported function within a module to a new address, returning the original function address.

```
// returns PIMAGE_IMPORT_DESCRIPTOR for an imported module  
PIMAGE_IMPORT_DESCRIPTOR KPEFile::GetImportDescriptor(  
    LPCSTR pDIIName)  
{  
    // first IMAGE_IMPORT_DESCRIPTOR  
    PIMAGE_IMPORT_DESCRIPTOR pImport = (PIMAGE_IMPORT_DESCRIPTOR)  
        GetDirectory(IMAGE_DIRECTORY_ENTRY_IMPORT);  
  
    if ( pImport==NULL )  
        return NULL;
```

```
while ( pImport->FirstThunk )
{
    if ( strcmp(pDlName, RVA2Ptr(pImport->Name))==0 )
        return pImport;

    // move to next imported module
    pImport++;
}

return NULL;
}

// returns address of __imp__xxx variable for an import function
const unsigned * KPEFile::GetFunctionPtr(
    PIMAGE_IMPORT_DESCRIPTOR pImport, LPCSTR pProcName)
{
    PIMAGE_THUNK_DATA pThunk;

    pThunk = (PIMAGE_THUNK_DATA) RVA2Ptr(pImport->
        OriginalFirstThunk);

    for (int i=0; pThunk->u1.Function; i++)
    {
        bool match;

        if ( pThunk->u1.Ordinal & 0x80000000 ) // by ordinal
            match = (pThunk->u1.Ordinal & 0xFFFF) ==
                ((DWORD) pProcName);
        else
            match = strcmp(pProcName, RVA2Ptr((unsigned)
                pThunk->u1.AddressOfData)+2) == 0;

        if ( match )
            return (unsigned *) RVA2Ptr(pImport->FirstThunk)+i;
        pThunk++;
    }

    return NULL;
}

FARPROC KPEFile::SetImportAddress(LPCSTR pDlName,
    LPCSTR pProcName, FARPROC pNewProc)
{
    PIMAGE_IMPORT_DESCRIPTOR pImport =
        GetImportDescriptor(pDlName);

    if ( pImport )
    {
        const unsigned * pfn = GetFunctionPtr(pImport, pProcName);
```

```
if ( IsBadReadPtr(pfn, sizeof(DWORD)) )
    return NULL;

// read the original function address
FARPROC oldproc = (FARPROC) * pfn;

DWORD dwWritten;

// overwrite with new function address
WriteProcessMemory(GetCurrentProcess(), (void *) pfn,
    & pNewProc, sizeof(DWORD), & dwWritten);

return oldproc;
}

else
    return NULL;
}
```

SetImportAddress needs support from two more functions. GetImportDescriptor searches through the module's import directory to find the IMAGE_DESCRIPTOR structure for the module from which the function is imported. With it, GetFunction Ptr searches the hint/name table to locate the function imported, returning the address of the corresponding entry in the import address table. For example, if the imported function is MessageBoxA, which is imported from user32.dll, GetFunctionPtr should return the address of __imp__MessageBoxA. Finally, SetImportAddress reads the original function address and overwrites with the new function address, using Write ProcessMemory.

After calling SetImportAddress, all calls within the module to the specified imported function will go to the new function, instead of the original imported function. As you can see, SetImportAddress is essential to basic API hooking. Here is a simple example of using the KPEFile class to take over the message box display:

```
int WINAPI MyMessageBoxA(HWND hWnd, LPCSTR pText, LPCSTR pCaption,
    UINT uType)
{
    WCHAR wText[MAX_PATH];
    WCHAR wCaption[MAX_PATH];

    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, pText,
        -1, wText, MAX_PATH);
    wcscat(wText, L"—intercepted");

    MultiByteToWideChar(CP_ACP, MB_PRECOMPOSED, pCaption,
        -1, wCaption, MAX_PATH);
    wcscat(wCaption, L"—intercepted");

    return MessageBoxW(hWnd, wText, wCaption, uType);
}
```

```
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int)
{
    KPEFile pe(hInstance);

    pe.SetImportAddress("user32.dll", "MessageBoxA",
        (FARPROC) MyMessageBoxA);

    MessageBoxA(NULL, "Test", "SetImportAddress", MB_OK);
}
```

The code overwrites the imported address of MessageBoxA in the current module to an application-provided routine MyMessageBoxA. All the calls to MessageBoxA will go to MyMessageBoxA instead, which for the demo purposes appends an extra word “intercepted” and displays the message box using MessageBoxW.

Export Directory

When your program imports a function/variable from a system DLL, the function/variable needs to be exported. To export a function/variable from a DLL, the PE file needs to contain three pieces of data—an ordinal number, an address, and an optional name. All the information regarding exported functions is gathered in an IMAGE_EXPORT_DIRECTORY, which can be accessed from the PE file header's export directory.

Although you can export both functions and variables, normally only functions are exported. That's why even field names in the PE file definition mention only functions.

An IMAGE_EXPORT_DIRECTORY structure stores the number of exported functions and the number of names. The number of names may be less than the number of functions. Most DLLs export functions by name. But some DLLs do mix export by name with export by ordinal—for example, comctl32.dll. Some DLLs—for example, MFC DLL—may have thousands of exported functions, so they export everything by ordinal to save some space originally occupied by names. COM DLLs export a fixed number of well-known functions like DllRegisterServer, while providing its service interfaces or virtual function tables. Some DLLs may export nothing, relying solely on the only DLL entry point.

The more interesting information in IMAGE_EXPORT_DIRECTORY consists of RVA addresses of the function address table, function name table, and function ordinal table. The address table holds RVA addresses of all exported functions, the name table holds RVA addresses of the function name strings, and the ordinal table holds the difference between the real ordinal and the base ordinal value.

With knowledge of the export table, we can easily implement GetProcAddress. But the Win32 API already provides a quite nice implementation, although regrettably it does not have a UniCode version. Instead, let's try to implement KPEFile:: SetExport Address.

As we have seen before, SetImportAddress patches a module's import table to changes of address of one imported function within a single module. Other modules within the process are not affected, which includes modules later loaded into the process. SetExportAddress works differently. It patches the export table of a module, so that any future link to it will be affected. Here is the code:

```
FARPROC KPEFile::SetExportAddress(LPCSTR pProcName,
    FARPROC pNewProc)
```

```
{  
    PIMAGE_EXPORT_DIRECTORY pExport = (PIMAGE_EXPORT_DIRECTORY)  
        GetDirectory(IMAGE_DIRECTORY_ENTRY_EXPORT);  
  
    if ( pExport==NULL )  
        return NULL;  
  
    unsigned ord = 0;  
    if ( (unsigned) pProcName < 0xFFFF ) // ordinal ?  
        ord = (unsigned) pProcName;  
    else  
    {  
        const DWORD * pNames = (const DWORD *)  
            RVA2Ptr(pExport->AddressOfNames);  
        const WORD * pOrds = (const WORD *)  
            RVA2Ptr(pExport->AddressOfNameOrdinals);  
  
        // find the entry with the function name  
        for (unsigned i=0; i<pExport->AddressOfNames; i++)  
            if ( strcmp(pProcName, RVA2Ptr(pNames[i]))==0 )  
            {  
                // get the corresponding ordinal  
                ord = pExport->Base + pOrds[i];  
                break;  
            }  
    }  
  
    if ( (ord<pExport->Base)|| (ord-pExport->NumberOfFunctions) )  
        return NULL;  
  
    // use ordinal to get the address where export RVA is stored  
    DWORD * pRVA = (DWORD *)RVA2Ptr(pExport->AddressOfFunctions)  
        + ord-pExport->Base;  
  
    // read original function address  
    DWORD rsIt = * pRVA;  
  
    DWORD dwWritten = 0;  
    DWORD newRVA = (DWORD) pNewProc-(DWORD) pModule;  
    WriteProcessMemory(GetCurrentProcess(), pRVA,  
        & newRVA, sizeof(DWORD), & dwWritten);  
  
    return (FARPROC) RVA2Ptr(rsIt);  
}
```

SetExportAddress first tries to find the ordinal number of the specified function using the function name through a search in the function name table if the ordinal is not given. Indexing the function address table with the ordinal number gives the location where the exported function's RVA address is stored. SetExportAddress then reads the original RVA and patches it with a new RVA calculated using the new function address.

Because only the export table is patched, after calling SetExportAddress, Get Proc Address will return the address of the new function. Future loading of DLLs into the process will link with the new function.

Neither SetImportAddress nor SetExportAddress provides a complete solution to API hooking with a process. But if we use them both, we are much closer to full coverage. The idea is simple: Walk through all currently loaded modules in the process and call SetImportAddress onto every one of them. Then call SetExportAddress to patch the export table. Now we have covered both loaded modules and yet-to-be-loaded modules.

This concludes our brief introduction to the PE file format. The knowledge presented here will be used in querying user virtual address space in [Chapter 3](#) and in implementing API hooking/spying in [Chapter 4](#). If you're really interested in PE file and API spying, think about whether there are still API calls which are not covered by both SetImportAddress and SetExportAddress.

[< BACK](#) [NEXT >](#)

1.5 ARCHITECTURE OF MICROSOFT WINDOWS OS

If you buy a CPU case with a power supply, a motherboard, a CPU, some RAM, a hard disk, a CD drive, a display card, a keyboard, and a monitor, and if you assemble everything together the right way, you get a computer. But to get the computer to do anything useful, you need software.

Computer software can be roughly classified as system software and application software. System software manages the computer and related peripherals to provide support for application software, which solves real problems for users. The most fundamental system software is the operating system, which manages all of the computer's resources and provides a nice interface to the application software.

The raw machines we are working on are very primitive and awkward to program, although they are much more powerful than their predecessors. One of the operating system's main tasks is to make the raw machine easier to program through a set of well-defined system calls. The system calls are implemented by the operating system in privileged processor mode, and they define the interface between the operating system and user programs running in nonprivileged processor mode.

Microsoft Windows NT/2000 is a little bit different from traditional operating systems. The Windows NT/2000 operating system consists of two major parts: a privileged kernel mode part and a nonprivileged user mode part.

The kernel mode part of the Windows NT/2000 OS runs in the privileged processor mode that has access to all CPU instructions and the full address space. On the Intel CPU, this means running in CPU privilege level 0, which has access to 4 GB of address space, IO address space, etc.

The user mode part of Windows NT/2000 OS runs in the nonprivileged processor mode that only has access to limited CPU instructions and user address space. On the Intel CPU, the user mode code is running in CPU privilege level 3 and has access only to the lower 2 GB of per-process address space with no access to IO ports.

The kernel mode part of the Windows NT/2000 operating system provides system services and internal processes to the user mode part of the operating system. Microsoft calls this portion Executive. It's the only entry point into the operating-system kernel. For security reasons, Microsoft provides no back doors. The major components are:

- Hardware Abstraction Layer (HAL): a layer of code that isolates the OS kernel mode parts from platform-specific hardware differences.
- MicroKernel: low-level operating system functions, such as thread scheduling, task switching, interrupt and exception dispatching, and multiprocessor synchronization.
- Device Drivers: hardware device drivers, file systems, and network support that implements

user I/O function calls.

- Window Management and Graphics System: implements graphical user interface functions, such as windows, controls, drawing, and printing.
- Executive: basic operating-system services, such as memory management, process and thread management, security, I/O, and interprocess communication.

The user mode part of Windows NT/2000 normally consists of three components:

- System Processes: special system processes, such as logon process and session manager.
- Services: such as event log and schedule services.
- Environmental Subsystems: provides native operating-system services to user programs through well-defined APIs. Windows NT/2000 supports the running of Win32, POSIX (Portable Operating System System Interface, an international standard for C language level API to the operating system), OS/2 1.2 (an operating system from IBM), DOS, and Win16 programs.

Hardware Abstraction Layer

The responsibility of the Hardware Abstraction Layer (HAL) is to provide a platform-specific support for the NT kernel, I/O manager, kernel-mode debuggers, and lowest-level device drivers. Having a HAL in the OS makes Windows NT/2000 not so dependent on a single particular hardware platform or architecture. HAL provides abstractions for device addressing, I/O architecture, interrupt management, DMA (Direct Memory Access) operations, system clocks and timers, firmware, and BIOS interfacing and configuration management.

When Windows NT/2000 is installed, the actual module for HAL is system32\hal.dll. But actually, there are different HAL modules for different architectures; only one of them gets copied to the system directory and renamed as hal.dll. Check your Windows NT/2000 installation CD, and you will find quite a few HALs, such as HALACPI.DL_, HALSP.DL_, and HALMPS.DL_. ACPI here means Advanced Configuration and Power Interface.

To check what's really provided by HAL on your system, use **dumpbin hal.dll/export**. You will find exported functions like HalDisableSystemInterrupt, HalMakeBeep, HalSetRealTimeClock, READ_PORT_UCHAR, WRITE_PORT_UCHAR, etc. Routines exported by HAL are documented in the Windows 2000 DDK, under Kernel-Mode Drivers, References, Part 1, Chapter 3.0: Hardware Abstraction Layer Routines.

MicroKernel

The Windows NT/2000 MicroKernel manages the main resource of the machine, the CPU. It provides support for interrupt and exception handling, thread scheduling and synchronization, multiprocessor synchronization, and timekeeping.

MicroKernel provides its services through object-based interfaces to its client, much like the objects and handles used in the Win32 API. The main objects provided by the MicroKernel are dispatcher objects and control objects.

Dispatcher objects are used for dispatching and synchronization. They include events, mutexes, queues, semaphores, threads, and timers. Each dispatcher object has a state, which is either signaled or not signaled. MicroKernel provides routines accepting dispatcher object states as parameters (KeWaitxxx). Kernel-mode threads synchronize themselves by waiting for dispatcher objects or user mode objects that have embedded kernel mode dispatcher objects. For example, the Win32 user level event object has a corresponding MicroKernel level event object.

Control objects are used to manage kernel mode operations except dispatching and synchronization, which are covered by dispatcher objects. Control objects include asynchronous procedure call (APC), deferred procedure call (DPC), interrupt, and process.

A spin lock is a lower-level synchronization mechanism defined by the NT kernel. It's used to synchronize access to shared resources, especially in a multiprocessor environment. When a routine tries to acquire a spin lock, it "spins" until the lock is acquired, doing nothing else.

MicroKernel resides in NTOSKRNL.EXE on your system, which contains both the MicroKernel and the Executive. There are two versions of MicroKernel on your installation CD: NTKRNLMP.EX_for multiprocessor systems and NTOSKRNL.EX_for single-processor systems. Although the module has an .EXE extension, it's actually a DLL. Among hundreds of exported functions from NTOSKRNL.EXE, approximately 60 of them belong to the MicroKernel. All routines supported by the MicroKernel start with "Ke." For example, KeAcquireSpinLock lets you acquire a spin lock which enables you to access shared data in a multiprocessor-safe way. KeInitializeEvent initializes a kernel event structure, which can then be used in KeClearEvent, KeResetEvent, or KeWaitForSingleObjects. Kernel objects are explained in Windows DDK, under Kernel-Mode Drivers, Design Guide, Part 1, Chapter 3.0: NT Objects and Support for Drivers. Kernel routines are documented under Kernel-Mode Drivers, References, Part 1, Chapter 5.0: Kernel Routines.

Device Drivers

We know that the MicroKernel manages the CPU; HAL manages things like bus, DMA, timer, firmware, and BIOS. For the computer to provide real value to users, the operating system needs to interface with lots of different kinds of devices, like the video display, mouse, keyboard, hard disk, CD ROM, network card, parallel ports, serial ports, etc. The operating system needs device drivers to talk to these hardware devices.

Most of the device drivers on Windows NT/2000 are kernel mode drivers, except Virtual Device

Drivers (VDD) for MS-DOS applications and Windows 2000 user-mode printer drivers. Kernel mode device drivers are DLLs loaded into kernel address space, depending on hardware configuration and user settings.

The Windows NT/2000 operating system's interface to device drivers is layered. User applications call API functions like Win32 CreateFile, ReadFile, WriteFile, etc. The calls are translated to calls to I/O system services provided by the Windows NT/2000 Executive. The I/O manager with the Executive sets up I/O request packets (IRPs), passing them through possibly layered drivers to physical devices.

Windows NT/2000 defines four basic types of kernel mode drivers, which have different structures and functionality:

- Highest-Level Driver: mostly file system drivers like the File Allocation Table (FAT) file system (inherited from DOS), NT file system (NTFS), and CD-ROM file system (CDFS) drivers, and the NT network server or redirector. File system drivers may implement a physical file system on a local hard drive; they may also implement a distributed or networked virtual file system. For example, some source-code version-control systems are implemented as a virtual file system. Highest-level drivers depend on lower-level drivers to function.
- Intermediate Drivers: such as virtual disk, mirror, or device-type-specific class drivers, drivers in the network transport stack, and filter drivers. They provide either value-added features or per-class processing for devices. For example, there is a class driver for parallel-port communication. Intermediate drivers also depend on support from underlying lower-level drivers. There may be multiple intermediate drivers in a driver stack.
- Lowest-Level Drivers, sometimes called device drivers: such as the PnP hardware bus driver, legacy NT device drivers, and network interface controller (NIC) driver.
- Mini-Drivers: customization module for generic drivers. A mini-driver is not a full driver. It resides inside a generic driver, which becomes its wrapper, to customize it for specific hardware. For example, Microsoft defines a printer UniDriver, which is an actual printer driver. Printer vendors can develop a printer mini-driver which will be loaded by UniDriver to make it print to the printer provided by the vendor.

A device driver may not always correspond to a physical device. A device driver is an easy way for programmers to write a module that can be loaded into kernel address space. With your module in kernel address space, you can do useful things that would not be possible otherwise. With well-defined file operations in the Win32 API, your user mode application can easily communicate with your kernel mode driver. For example, www.sysinternals.com provides several very useful NT utilities which use NT kernel mode device drivers to achieve registry, file system, and I/O port monitoring. [Chapter 3](#) of this book shows a simple kernel mode driver that reads data from kernel address space. It is used extensively in analyzing Windows graphics subsystem data structures.

While most device drivers are part of the I/O stack controlled by the Executive I/O manager and have similar structures, some device drivers are exceptions. Device drivers for the Windows NT/2000 graphics engine—for example, display driver, printer driver, video port driver—use a different structure and a direct calling sequence. Windows 2000 even allows printer drivers to run in user mode. We will say more about display drivers and printer drivers in [Chapter 2](#).

Most of the modules loaded into kernel mode address space are device drivers. The **drivers** tool in Windows NT/2000 DDK can list them in a DOS box. You will find tcpip.sys for networking, mouclass.sys for mouse, kbdclass.sys for keyboard, cdrom.sys for CD-ROM, etc.

For complete discussions on Windows 2000 device drivers, read the Windows 2000 DDK, Kernel-Mode Drivers, Design Guide and References.

Window Management and Graphics System

In the earlier versions of Microsoft Windows NT, security was a major concern for OS design, so window management and graphics system ran in user address space. This led to so many performance problems that Microsoft made a major change in design by moving the windowing and graphics system from user mode to kernel mode starting with Windows NT 4.0.

Window management supports the foundation of the Windows graphic user interface by implementing window class, window, window messaging, window hooks, window property, menu, title bar, scrollbar, cursor, virtual key, clipboard, etc. It's basically the kernel counterpart of USER32.DLL, which implements what's defined in the winuser.h portion of the Win32 API.

Graphics system implements the real GDI/DirectDraw/Direct3D drawing, calling on a physical device or memory-based device. It relies on graphics device drivers like display drivers and printer drivers. Graphics system is the backbone for GDI32.DLL, which implements what's defined in the wingdi.h portion of the Win32 API. Graphics system also provides strong support for display drivers and printer drivers, by providing a fully featured rendering engine for several standard format bitmap surfaces. We will cover the details of the graphics system in [Chapter 2](#).

Windowing and the graphics system are packed into a single huge DLL, win32k.sys, of around 1.6 MB. If you look at the exported functions from win32k.sys, you can find graphics system entry points—for example, EngBitBlt, PATHOBJ_bMove To—but not window management entry points. The reason is that window management entry points are never called by other parts of the OS kernel components, while functions in graphics system need to be called by graphic device drivers. GDI32.DLL and USER32.DLL calls WIN32K.SYS through system service calls.

Executive

Microsoft defines Windows NT/2000 Executive as the collection of kernel-mode components that form the base Windows NT/Windows 2000 operating system. Besides HAL, Micro-Kernel, and

Device Drivers, Executive also includes the Executive Support, Memory Manager, Cache Manager, Process Structure, Interprocess Communication (LPC, local procedure call and RPC, remote procedure call), Object Manager, I/O Manager, Configuration Manager, and Security Reference Monitor.

Each component within the Executive provides a set of system services which can be used from user mode through interrupts, except Cache Manager and HAL. Every component also provides an entry point that can be called only by modules running in kernel address space.

Executive Support provides a set of functions callable from kernel mode, which normally starts with the prefix "Ex". Kernel memory allocation is a main functionality provided by Executive Support. Windows NT/2000 uses two dynamically growing memory regions, called pools, to manage dynamic memory allocation in kernel mode address space. The first is nonpaged pool, which is guaranteed to be resident in physical RAM all the time. Critical routines like interrupt services can use nonpaged pool without worry about generating paging interrupts. The second pool is paged pool, which is much bigger but can be swapped out to the hard disk when physical RAM is low. For example, storage for Win32 device-dependent bitmaps is allocated from paged pool using the family of ExAllocatePoolxxx routines. Executive Support also provides an efficient fixed-size block memory allocation scheme called look-aside lists, using routines like ExAllocatePagedLockasideList. Multiple look-aside lists are allocated from the pools when the system boots up. Executive Support provides a rich set of atomic operations, such as ExInterlockedAddLargeInteger, ExInterlockedRemoveHeadList, InterlockedCompareExchange, etc. Other Executive Support functionality includes fast mutex, callback, throw exception, time translation, Universally Unique Identifier (UUID) creation, etc.

Memory Manager supports virtual memory management, balance set management, process/stack swapping during thread swapping, modified page writer, mapped page writer, system cache, page file, virtual memory to physical memory mapping, etc. Memory Manager provides routines like MmFreeContiguousMemory, MmGet PhysicalAddress, MmLockPageableCodeSection, etc. For details of Memory Manager, check [Chapter 5](#) of *Inside Windows NT*, second edition.

Cache Manager provides data caching for Windows NT/2000 file-system drivers. Cache Manager routines have the prefix "Cc." Cache Manager exports routines like CclsThereDirtyData and CcCopyWrite. For details, check [Chapter 8](#) of *Inside Windows NT*, second edition.

Process Structure routines are responsible for creating and terminating kernel mode system thread, process/thread notification, and process/thread query. For example, Memory Manager could use PsCreateSystemThread to create a kernel thread to manage dirty page writing.

Object Manager manages common behavior of objects exposed from the Executive. The Executive supports creation of directory, event, file, key, section, symbolic link, and timer objects, through routines like ZwCreateDirectorObject and ZwCreate File. Once created, Object Manager routine ObReferenceObject/Ob De refer ence Count updates reference count, ObjReferenceObjectByHandle validates object handle and returns the pointer to the object's body.

I/O Manager translates I/O requests from the user mode program or other kernel mode components into properly sequenced calls to various drivers on driver stacks. It provides a very rich set of routines. For example, IoCreateDevice initializes a device object for use by a driver, IoCallDriver sends an I/O request packet to the next-lower-level driver, and IoGetStackLimits checks the current thread's stack boundary.

Windows NT/2000 Executive also contains a small runtime library support, which is parallel to the C runtime library but much smaller. The kernel runtime library supports double-link list, Unicode conversion, bit operations, memory, large integer, registry access, time conversion, string manipulation, etc.

On Windows NT/2000, Executive and MicroKernel are packed into a single module NTOSKRNL.EXE, which exports more than 1000 entry points. Entry points exported by NTOSKRNL.EXE normally have distinguishing double-letter prefixes that indicate the components they belong to—for example, “Cc” for cache manager, “Ex” for Executive Support, “Io” for I/O manager, “Ke” for MicroKernel, “Ob” for object manager, “Ps” for process structure, “Rtl” for runtime library, “Dbg” for debugging support, etc.

System Services: Native API

The rich functionality provided by the kernel part of the Windows NT/2000 operating system is finally exposed to user mode modules through a single narrow gate, which is interrupt 0x2E on the Intel CPU. It's served by an interrupt service routine KiSystemService, which is in NTOSKRNL.EXE but not exported. Because the service routine is running in kernel mode, the CPU switches itself to privileged execution mode automatically, making the kernel address space addressable.

Although a single interrupt is used, more than 900 system services are provided by Windows NT/2000 through a service index number in EAX register (on the Intel CPU). NTOSKRNL.EXE keeps a table of system services named KiServiceTable; so does WIN32K.sys in W32pServiceTable. System service tables are registered by calling KeAddSystemServiceTable. When KiSystemService services a system service call, it checks if the service index is valid and the expected parameters accessible, then dispatches to the right routine handling a particular system service.

Let's look at a few simple system services, using the Microsoft Visual C++ debugger with the help of the Windows 2000 debug symbol files. If you step into Win32 GDI and call CreateHalftonePalette in assembly code, you will see:

_NtGdiCreateHalftonePalette@4:

```
mov    eax, 1021h  
lea    edx, [esp+4]  
int    2Eh  
ret    4
```

The Win32 USER function GetDC is implemented as:

_NtUserGetDC@4:

```
mov    eax, 118bh  
lea    edx, [esp+4]  
int    2Eh  
ret    4
```

The Win32 KERNEL function CreateEvent is more complicated. CreateEventA calls CreateEventW, which then calls NtCreateEvent from NTDLL.DLL. NtCreateEvent is implemented by:

_NtCreateEvent@20:

```
mov    eax, 1Eh  
lea    edx, [esp+4]  
int    2Eh  
ret    14h
```

The Windows NT/2000 system calls are almost completely hidden from programmers. SoftICE/W from Numega provides an ntcall command that lists some kernel system calls. For more information on system calls, refer to Mark Russinovich's article "*Inside the Native API*," on www.sysinternals.com. We will cover more details about GDI system calls in [Chapter 2](#).

System Processes

The Windows NT/2000 operating system employs a few system processes in order to manage login, services, and user processes. You can find list of system processes in task manager, or you can use the Task List tool (**tlist**) which comes with Platform SDK.

When Windows NT/2000 is running, there are basically three hierarchies of processes. The first hierarchy contains a single system process, the system idle process, which always has process id 0. The second hierarchy holds all other system processes. It starts with a process named system, which is the parent of the session manager process (smss.exe). The session manager process is the parent of the Win32 subsystem process (csrss.exe) and the logon process (winlogon.exe). The third hierarchy starts with the program manager process (explorer.exe), which is the parent of all user processes.

Here is an annotated Windows 2000 process tree, as displayed by **tlist -t** command:

System Process (0)	System Idle Process
System (8)	

smss.exe (124)	Session Manager
csrss.exe (148)	Win32 Subsystem server
winlogon.exe (168)	logon process
services.exe (200)	service controller
svchost.exe (360)	
spoolsv.exe (400)	
svchost.exe (436)	
mstask.exe (480)	SYSTEM AGENT COM WINDOW
lsass.exe (212)	local security authentication server
explorer.exe (668)	Program Manager
OSA.EXE (744)	Reminder
IMGICON.EXE (760)	

The Process Walker tool (**pwalker.exe**) displays more per-process information. With Process Walker, you can find that the System Idle Process has a single thread with 0 as starting address. It may well be a mechanism for the operating system to account for idle time instead of being a real process. The System Process has a valid starting address in kernel address space and dozens of threads all with a starting address in kernel address space. So the System Process is also the home for kernel mode system threads. If you translate the thread starting addresses in the System Process to symbolic names, you will find interesting names like Phase1Initialization, ExpWorkerThread, Exp Worker Thread BalanceManager, MiDereferenceSegmentThread, MiModified Page Writer, Ke BalancedSetManager, FsRtlWorkerThread, etc. Although all the threads listed here are created by the Executive, other kernel components can create kernel threads, too. Both the Idle Process and the System Process are pure kernel mode components which do not have any modules in user mode address space.

Other system processes like session manager, logon process, etc., are user mode processes, started from a PE format execution file. For example, you can find smss.exe, csrss.exe, and winlogon.exe in the system directory.

Services

Microsoft Windows NT/2000 supports a special application type known as a service program, which is normally a console program controlled by the Service Control Manager (SCM). A service program can provide several services. Unlike normal user programs, services can be scheduled to start automatically at system boot time before user logon.

Several service programs are started when the operating system boots up. To list the service programs and services currently running on your system, use Task List (tlist .exe) with the **-s** option. Here is a sample list of service programs and services:

200 services.exe Svcs: AppMgmt, Browser, dmserver,
Dnscache, EventLog, LanmanServer,
LanmanWorkstation,

LmHosts, Messenger, PlugPlay,
ProtectedStorage, seclogon,
TrkWks

212 lsass.exe Svcs: PolicyAgent, SamSs
360 svchost.exe Svcs: RpcSs
400 spoolsv.exe Svcs: Spooler
436 svchost.exe Svcs: EventSystem, Netman, NtmsSvc,
RasMan, SENS, TapiSrv
480 mstask.exe Svcs: Schedule

Of particular interest to this book is the spooler service, which handles print jobs on a local machine and printing to printers over a network. We will talk more about spooler service in [Chapter 2](#).

Environment Subsystems

When Windows NT was initially designed, not many Win32 programs had been written to run on it. So Microsoft thought it was important for Windows NT to support environments to run DOS programs, Win16 programs, OS/2 programs, POSIX (UNIX-style interfaces) programs, and Win32 programs on the same operating system. Windows NT/2000 provides different environment subsystems to run these totally different kinds of programs.

An environment subsystem is a set of processes and DLLs that exposes a subset of the operating-system services to application programs written for the specific subsystem. Each subsystem has a single subsystem process, which manages interaction with the operating system. The DLLs are mapped into application processes to allow interfacing with the subsystem process or direct interaction with the kernel portion of the operating system through OS system service routines.

The Win32 environment subsystem is gradually becoming the major subsystem supported by the Windows NT/2000 family of operating systems. Its environment subsystem process (csrss.exe) is used to handle all window management and graphic display in the user address. Win32 application programs have to issue local procedure calls to its subsystem process for window management and display, which causes performance problems. Starting from Windows NT 4.0, Microsoft moved the major portion of the Win32 environment subsystem into a kernel mode DLL, win32k.sys, together with all the graphical device drivers. Win32 subsystem DLLs should be very familiar to Windows programmers. We have KERNEL32.DLL, which handles virtual memory, I/O, heap, process, thread, and synchronization; USER32.DLL, which handles window management and message passing; GDI32.DLL, which handles graphic display and printing; ADVAPI32.DLL, which manages registry, etc. Win32 subsystem DLLs allow direct access to the OS kernel system services, and provide extra useful features not provided by OS system services. The enhanced metafile (EMF) is an example of a Win32 API feature not directly supported by win32k.sys.

The remaining two environment subsystems, OS/2 subsystem and POSIX subsystem, rely on the Win32 environment subsystem to run, although they used to be the major players in the initial design of Windows NT.

The Win32 environment subsystem now becomes a vital part of the operating system, which is always running. The OS/2 and POSIX subsystems are started only when needed by a program requiring such subsystems.

[< BACK](#) [NEXT >](#)

1.6 SUMMARY

In this chapter we talked briefly about the basics of Windows programming, using the C++ language. We touched upon skeleton Windows programs using C++, a little assembly language, the program development environment, the Win32 executable format, and the architecture of the Microsoft Windows NT/2000 operating system.

Starting in [Chapter 2](#), we will focus on Windows graphics programming under Windows NT/2000, although when needed we will develop some small tools to help out in our exploration.

Further Reading

To get a deeper understanding of the topics we covered in this book, there are lots of resources to turn to.

Jeffrey Richter's *Programming Applications for Microsoft Windows*, 4th ed., is a masterpiece on core Windows programming, covering kernel object, process, thread, memory, memory-mapped file, DLL, file system, device I/O, and structure exception handling.

David A Solomon's *Inside Windows NT*, 2d ed., is a must-read on Windows NT architecture and internals. It's a book targeted at a general audience, not just programmers. There is no source code and no companion CD with the book. The descriptions of many Windows NT internal data structures do not mention data types or field offsets.

John Robbins' *Debugging Applications* should be on every Windows programmer's bookshelf.

If you're interested in Executive system service calls, check *Undocumented Windows NT* and *Windows NT/2000 Native API Reference*.

Matt Pietrek's *Windows 95 System Programming Secrets* is a classic on Win32 programming in the Windows 95 era. It has excellent coverage on the PE file format and Windows API spying. The book spends 12 pages on the Windows 95 GDI.

Charles Petzold's *Programming Windows*, 5th ed., is a classic on the Win32 API. The book focuses more on the Windows 95/98 part of the Win32 API, instead of the WinNT/2000 part of API. You will find sample code in the book in the pure classic SDK style.

Brent Rector and Joseph Newcomer's *Win32 Programming* is a heavyweight book on the Win32 API.

Besides books, web pages like www.codeguru.com, www.codeproject.com, or

www.msdn.microsoft.com are good online resources. System Internals' web page, www.sysinternals.com, has lots of insightful articles, tools, and source code helping you to explore the system.

Intel has a developer's web page if you want to learn more about Intel CPUs, performance fine-tuning, AGP bus, or Intel C++ compiler, etc. Adobe has a developer's web page for anyone talented enough to do photo image plug-ins and filters for Adobe applications. Video display card vendors also have good developer web pages.

Sample Programs

The complete programs described in this chapter can be found on the CD that comes with the book, as listed in [Table 1-5](#).

Table 1-5. Sample Programs for Chapter 1

Project Location	Description
Sample\Chapt_01\Hello1	"Hello, World," starting a browser.
Sample\Chapt_01\Hello2	"Hello, World," drawing on the desktop.
Sample\Chapt_01\Hello3	"Hello, World," class for simple window.
Sample\Chapt_01\Hello4	"Hello, World," blending using DirectDraw.
Sample\Chapt_01\GDISpeed	Using assembly code for performance measurement.
Sample\Chapt_01\SetProc	Simple API hooking through overwriting PE file import/export directories.

[< BACK](#) [NEXT >](#)

Chapter 2. Windows Graphics System Architecture

The Graphics system is an indispensable part of a modern operating system, such as Microsoft Windows NT/2000, which relies more and more on the intuitive graphical user interface to be accessible to the average user.

[Chapter 1](#) ended with a brief discussion of Windows NT/2000 operating system architecture. This chapter zooms in on the graphics system parts of the operating system. We will be covering graphics system components and their relationships, in addition to major components like the GDI API, DirectDraw API, OpenGL API, the graphics engine, the display driver, the printing and spooling system, and printer drivers. We will also discuss the vertical layers in the Windows graphics system, namely, the user mode system DLLs supporting the system calls, kernel mode engine, and third-party-provided graphics device drivers. This chapter ends with a simple printer driver that generates HTML pages as output.

2.1 WINDOWS GRAPHICS SYSTEM COMPONENTS

The Windows Application Programming Interface—in other words, the Win32 API—is a huge set of interrelated functions providing different kinds of services to application programs. From the programmer's viewpoint, the Win32 API can be divided into several groups of services:

- Window Base Services, commonly known as kernel services, which include Microsoft clustering, debugging, error handling, dynamic load library (DLL), process, thread, file, I/O, international features, interprocess communication, performance monitoring, removable storage, security, etc.
- User Interface Services, commonly known as user services, which include windowing, message queue, dialog box, control, common control, common dialog box, resource, user input, shell, etc.
- Graphics and Multimedia Services, including broadcast architecture, color management, DirectX, GDI, multimedia, video for windows, still image, OpenGL, Windows Media, etc.
- COM, OLE, and Active X Services, including Component Object Model (COM), automation, Microsoft Transaction Server, Object Linking and Embedding (OLE), etc.
- Database and Messaging Services, including Data Access Objects (DOA), the Structured Query Language (SQL) server, Messaging API (MAPI), etc.
- Networking and Distributed Services, including active directory, message queue, networking, remote procedural call, routing and remote access, Systems Network Architecture (SNA) server, synchronization manager, Telephony API (TAPI), etc.
- Internet, Intranet, and Extranet Services, including indexing service, Internet Explorer, Microsoft agent, NetShow, scripting, site server, etc.
- Setup and System Management Services, including configuration, setup, systems management, etc.

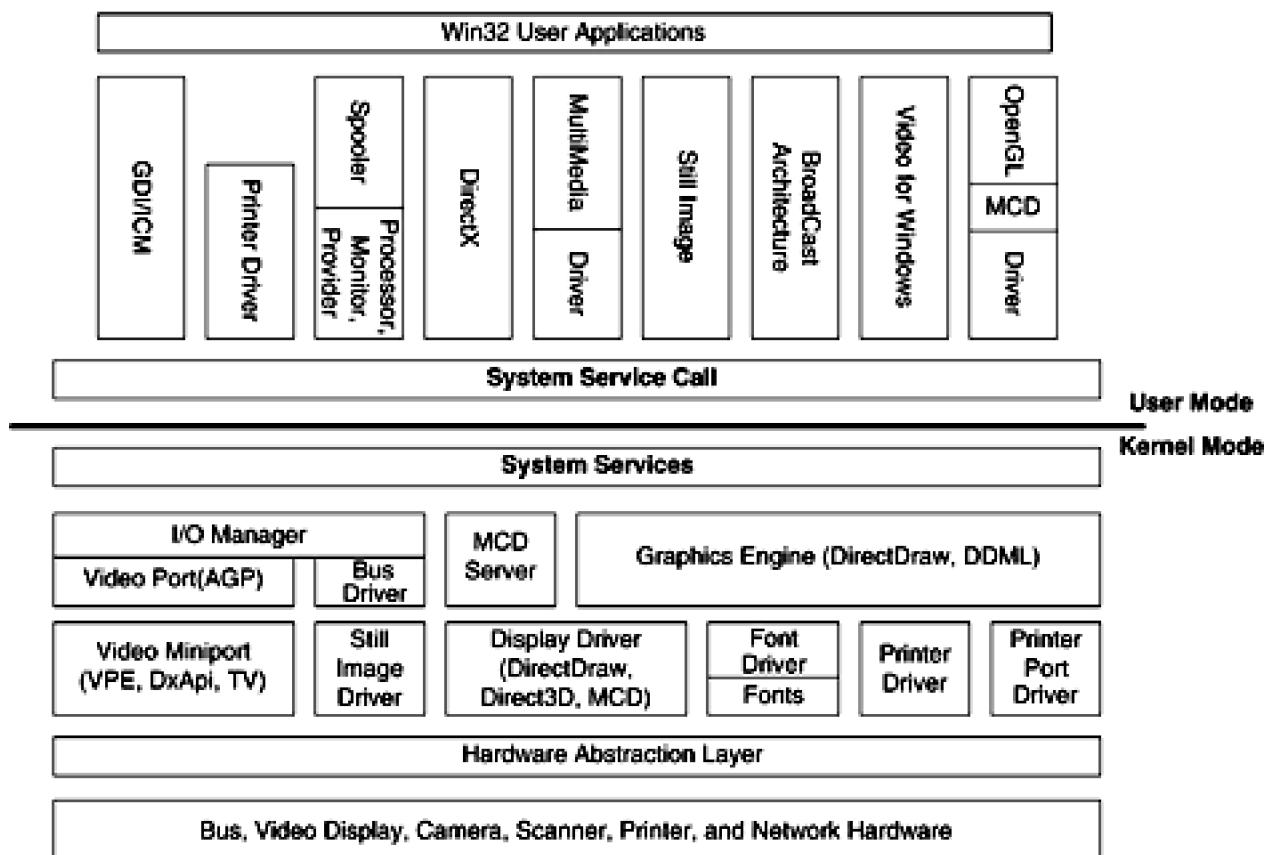
Each group of services is supported by a set of operating-system components. They include Win32 environment subsystem DLLs, user mode drivers, OS system services, and kernel mode drivers. Each group is well worth several good books to uncover the deep knowledge needed to use those services effectively. For example, Jeffrey Richter's *Programming Applications for Microsoft Windows*, 4th ed. (originally called *Advanced Windows*), is a masterpiece on windows base services; while Charles Petzold's *Programming Windows*, 5th ed., focuses on user interface services and GDI services.

The Graphics and Multimedia Services group of the Win32 API is so huge that it may well take several books to cover adequately. The focus of this book is a very important subset of the Graphics and Multimedia Services, namely GDI and DirectDraw. Let's now take a closer look at the components that support Graphics and Multimedia Services.

The Win32 graphics APIs are implemented on several platforms: Windows 95/98, WinCE, Windows NT, and the latest Windows 2000. It used to be that NT-based systems provided the best support for GDI, being a true 32-bit implementation, while Windows 95-based systems provided the best support for game programming. But the new Windows 2000 operating system provides the best of both worlds. Windows 2000 makes significant changes to provide hardware acceleration for DirectX/OpenGL, the new still-image API, broadcasting architecture, user mode printer driver, etc. In this book we will focus on the Windows 2000 graphics and multimedia system architecture, while pointing out the differences with Windows 95/98 and Windows NT 3.5/4.0 from time to time.

Looking top-down at [Figure 2-1](#), as you've seen with the overall operating-system design, the Windows NT/2000 graphics and multimedia system is a layered system. The topmost box contains application programs, which interface with a set of 32-bit user mode system DLLs through the Win32 API. The system DLL layer contains familiar DLLs like GDI32.DLL for the graphics device interface, USER32.DLL for the user interface and window management, KERNEL32.DLL for window base services, etc. Most of the modules in this layer are provided by the operating system, but a few components in this layer rely on help from some user mode drivers provided by hardware vendors. Below this is the system service call gate, which invokes system calls that are served by system service routines in the kernel mode portion of the operating system. The Windows NT/2000 Executive, which lives in kernel address space, provides the graphics engine, I/O manager, video port driver, etc., to support the graphics and multi media system. It needs support from vendor-supplied device drivers, which talk to an array of hardware devices like the bus, video display, and printer with the help of the OS hardware abstraction layer.

Figure 2-1. Windows 2000 graphics/multimedia system architecture.



Now let's scan through the user mode portion horizontally. Graphics Device Interface (GDI) and Image Color Management (ICM) provide a device-independent graphics programming interface to application programs. If you're doing printing output, GDI eventually needs to talk to a printer driver, which could be running in user mode in Windows 2000. User mode printer drivers rely heavily on functions provided by the graphics engine, for which they call GDI to pass calls back to the graphics engine. Printing jobs are controlled by the spooler system process, which relies on vendor-customizable components like the print processor, printer monitor, and printer provider.

DirectX adds a relatively new set of Win32 system DLLs, which implement the DirectX COM interfaces. For actual interfacing with DirectX implementation in kernel address space, DirectX goes through GDI. DirectX is made up DirectDraw, Direct Sound, DirectMusic, DirectInput, DirectPlay, DirectSetup, AutoPlay, and Direct3D. This book covers only the DirectDraw part of DirectX.

We will cover GDI and the DirectDraw portion of DirectX in much more detail. For the moment, let's just quickly go through the components we will not be covering in this book.

Multimedia

The multimedia part of the Win32 API is a continuation from multimedia support started in Windows 3.1. It contains the MCI (media control interface), audio output, multimedia file I/O, joystick, and the multimedia timer. The MCI is defined to control all linear playback media, with functions like load, pause, play, record, stop, resume, etc. There are three types of audio output supported: CD audio, MIDI (musical instrument digital interface), and waveform audio. The Win32 multimedia API is

defined in mmsystem.h, with winmm.lib and winmm.dll being its import library, and Win32 system DLL. Winmm.dll relies on installable user mode device drivers for each actual multimedia device. The single major exported function for a multimedia driver, which is itself a 32-bit DLL, is DriverProc. DriverProc handles messages from the multi media system, such as DRV_OPEN, DRV_ENABLE, DRV_CONFIGURE, DRV_CLOSE, etc.

Note

To check the multimedia drivers available on your system, open mmdriver.inf under the %SystemRoot%\system32 directory. You can find around a dozen multimedia drivers. For example, mmdrv.dll is for low-level wave, MIDI (musical instrument digital interface), and AUX support (auxillary output device); msacm32.drv is Microsoft Audio Compression Manager; and ir32_32.dll is the Indeo codec, a video compressor/decompressor developed by Intel, which uses a wavelet compression algorithm with MMX support.

If you wonder how Windows NT/2000 user mode drivers can control hardware like your speaker, they can't do it alone. User mode multimedia drivers rely on a class of kernel mode drivers called kernel streaming drivers, which are capable of controlling hardware directly.

The Win32 multimedia component is gradually being replaced by corresponding components in DirectX, with better features and higher performance. For example, DirectSound handles waveform-audio sound capture and playback; DirectMusic can capture and play back digital sound samples, including MIDI; DirectInput supports a whole range of input devices including mouse, keyboard, joystick, and other game controllers, as well as force-feedback devices.

One multimedia function is widely used by non-multimedia Windows applications to get high-resolution timer readings, namely, the timeGetTime() function. It can offer at best 1 ms accuracy, usually better than GetTickCount() (1 ms on Windows 95, 15 ms on Windows NT/2000). For Win32 programs, QueryPerformanceCounter is orders of magnitude more accurate than both timeGetTime and GetTickCount if the CPU supports a high-resolution performance counter. On machines with the Intel Pentium CPU, the high-resolution performance counter is the CPU clock-cycle count we mentioned in [Chapter 1](#). So on a 200-MHz CPU, its accuracy is 5 nanoseconds. But calling Query PerformanceCounter itself is not so high performance; a system call is made to the OS kernel to return the performance counter.

Video for Windows

Similar to Win32 multimedia, Win32 Video for Windows has a long history dating back to Windows 3.1. Video for Windows (VFW) provides Win32 API support for video data processing. To be more specific, it supports AVI (audio-video interleaved) file read, write, positioning, and editing, the video

compression manager, video capture, and the DrawDib API. Many of VFW's features have been superseded by Direct Show within DirectX.

The DrawDib API contains functions like DrawDibDraw, DrawDibGetBuffer, DrawDibUpdate, etc. It's similar to the Win32 StretchDIBits call, but provides extra features like calling the right decoder, data streaming, and supposedly higher performance. The first two benefits are from installable multimedia drivers that support different data streams; the third benefit is certainly no match for DirectDraw. Win32 VFW is supported by header file vfw.h, library file vfw32.lib, and runtime DLL mservfw32.dll. The VFW implementation relies on Win32 multimedia implementation.

Note

The DrawDib functions are still advertised as providing high-performance image-drawing capabilities which do not rely on GDI and write directly to video memory. While it sounds good, it may no longer be true, especially with Windows NT/2000. On Windows NT/2000, where direct video access is available only through kernel mode driver or DirectX, DrawDibDraw does use the GDI function to draw normal Window DIB, which makes it slower than the GDI DIB drawing function.

Still Image

The Still Image (STI) API is a new Microsoft interface for acquiring digital still images from devices like scanners and digital cameras and is available only on Windows 98 and Windows 2000. Clearly STI is a replacement for the older TWAIN standard. (You may wonder why it's not called DirectImage; perhaps it will be soon.) Because it's new, Microsoft has the luxury of using COM interfaces for STI instead of traditional Win32 API function calls. Microsoft STI is made up of a still-image event monitor, vendor-supplied user mode still-image mini-drivers, and a scanner and camera control panel. The still-image event monitor monitors system-wide still image devices and their events. It also keeps track of registered still-image applications, which can be started when an event is detected. A still-image mini-driver detects events from a particular device, notifying the event monitor of activities. It also passes image data from the kernel mode driver to user mode. The scanner and camera control panel allows the user to associate still-image devices with still-image-aware applications. The scanner/camera control panel applet (sticpl.dll), still-image monitor (stimon.dll, stisvc.exe) and imaging applications all use STI COM object (CLSID_Sti), which implements the IStillImage interface, whose instance is created by StiCreateInstance. The STI COM object is implemented by sti.dll, which uses IStiDevice and IStiDeviceControl COM interfaces to control still-image mini-drivers. Windows 98/2000 STI API is supported by header file sti.h, library file sti.lib, the DLLs and EXE mentioned above, and user mode and kernel mode still-image device drivers.

OpenGL

The last user mode component in [Figure 2-1](#) is OpenGL. OpenGL is a programming standard for 2D/3D graphics developed by Silicon Graphics, Inc. Its main purpose is to render 2D/3D objects into a frame buffer. OpenGL allows a programmer to describe objects using a group of vertices, each with its coordinate, color, normal, texture coordinate, and edge flag. This allows a description of points, line segments, and 3D surfaces, using OpenGL function calls. The OpenGL graphic control allows specification of transformation, lighting equation coefficients, antialiasing methods, and pixel update operators. The OpenGL render goes through several stages to finally render the data into a frame buffer. The first stage, evaluator, approximates curves and surfaces by evaluating polynomial commands. The second stage, vertex operation and primitive assembly, transforms, lights, and clips vertices. The third stage, rasterization, produces a series of frame-buffer addresses and associated values. The last stage, per-fragment operation, does depth buffering, alpha blending, masking, and other pixel operations in the final frame buffer.

The Microsoft OpenGL implementation, as seen on Windows NT/2000, adds some extra features to the OpenGL standard. It implements the full set of OpenGL commands, OpenGL Utility (GLU) library, OpenGL Programming Guide Auxiliary Library, Window extension (WGL), per-window pixel format, and double buffering. OpenGL uses three header files in the gl subdirectory of your compiler include directory, including gl.h, glaux.h, and glu.h. The window extension (WGL) part is defined in the GDI header file wingdi.h. OpenGL uses library files opengl.lib and gdi32.lib and runtime DLLs opengl32.dll and gdi32.dll.

To improve OpenGL performance, its implementation allows for vendor-supplied drivers to perform vendor-specific optimization and direct hardware access. Microsoft provides mini-client architecture (MCD) to facilitate OpenGL drivers. Open GL.dll loads mcd32.dll, the OS-supplied client-side DLL, and an optional vendor-supplied user mode OpenGL driver. You can find your OpenGL driver by searching for OpenGLDrivers in the registry. The MCD client and OpenGL user mode driver uses the GDI function ExtEscape to send commands to the graphics engine and driver in kernel mode. A display driver providing OpenGL optimization is needed to support its MCD portion, with the help of a kernel MCD server in mcdrv32.dll.

It's quite common nowadays for graphics card vendors to support DirectDraw, Direct3D, and OpenGL hardware acceleration in a single package. It's always interesting to see different designs for similar purposes, in this case, GDI and OpenGL. Initially, GDI started as a simple graphics programming interface for the hardware the PC industry had during that time, namely, 16- or 256-color EGA or VGA display cards, and monochrome printers. Gradually, GDI added support for device-independent bitmaps, color printers, vector fonts, TrueType fonts, OpenType fonts, 32-bit logical coordinate space, gradient fill, alpha channel, multiple monitor support, multiple terminal support, and so on. GDI is still evolving to GDI+. GDI can really run a small device like a palmtop PC as well as a powerful machine like a workstation. Performance, backward compatibility, and device independence may be the main design goals behind GDI, or the Windows API in general. On the contrary, OpenGL is designed to be a high-end 2D/3D graphics package for creating photorealistic scenes. OpenGL's heavy usage of floating-point calculation needs a high-end machine with lots of

memory and horsepower. Various effects like lighting, blending, antialiasing, and fogging would not be effective on a 256-color VGA monitor. Although OpenGL is designed to be device-independent, it's mainly designed to render onto a frame buffer. So OpenGL still has difficulty printing on high-resolution printers. Actually, on Windows NT/2000, GDI is offering an OpenGL printing solution by recording OpenGL commands into a special EMF format and then playing them back to a high-resolution printer device.

Because of the complexity of building a 2D/3D scene, OpenGL is a higher-level graphics language than GDI. An OpenGL program normally builds a scene in three-dimensional space using vertices, line segments, and polygon surfaces, defines attributes, lighting sources, and viewing angles, and then has the OpenGL engine do all the detailed work. In comparison, GDI is a much more procedural-level language, where the application needs to specify the exact command parameters and sequence to construct a picture. If you want to create a 3D scene, GDI does not help you to calculate 3D depth and remove hidden surfaces. Even Direct3D Immediate Mode is a lower level API when compared with OpenGL.

Windows Media

Windows Media is a new addition to the Win32 Graphics/Multimedia system. It contains Windows Media Services, Windows Media Encoder, Windows Media Player Control, and Windows Media Format SDK.

Windows Media Services provides ActiveX controls and COM APIs to allow Web authors to add streaming media to their web sites and to control, monitor, and managing media broadcasting.

Windows Media Encoder is primarily responsible for encoding different types of media content into the Windows Media Format stream or file, which can then be delivered through Windows Media Services. Advanced Streaming Format (ASF) is a file container which can contain data in different media formats.

Windows Media Player Control is an ActiveX control for adding multimedia playback to applications and web pages.

Windows Media Format SDK is a development kit for reading, writing, and editing Windows Media audio, video, and script files.

OS Kernel Mode Components

The user mode graphics and multimedia components have two ways to communicate with the operating system kernel. For GDI, DirectDraw, Direct3D, and OpenGL, user mode calls go through gdi32.dll, which provides the interface to hundreds of system service calls. For interaction with video port drivers and multimedia drivers, user mode calls use the normal file I/O API provided by Windows base services.

File I/O system services are handled by the I/O manager in the kernel mode Executive, which calls corresponding drivers. GDI, DirectDraw, Direct3D, and OpenGL calls go through the graphics engine, which passes them to individual device drivers.

The actual operating-system modules involved include NTOSKRNL.EXE (system service dispatching, I/O manager), WIN32K.SYS (graphics engine), MCDSVR32.DLL (MCD server), and HAL.DLL (hardware abstraction layer).

The Windows NT/2000 kernel Executive, NTOSKRNL.EXE, is the most important piece in the OS kernel. In the graphics system, it's mainly responsible for dispatching graphics-system service calls to the graphics engine, because the graphics engine happens to use the same system service-call mechanism used by other more hard-core system services. The hardware abstraction layer, HAL, provides some service routines to the graphics device driver for things like reading and writing hardware registers. It helps other components of the kernel to be more platform independent. For more discussion on the Executive and HAL, refer to [Chapter 1](#).

Kernel Mode Drivers

The Windows NT/2000 graphics and multimedia system relies on quite a number of vendor-supplied drivers to interface with the final hardware.

The most important driver is the display driver, which should handle GDI, DirectDraw, Direct3D, and MCD for OpenGL support. The display driver is always paired with a video miniport driver, which handles interaction with things like hardware ports. The video miniport driver also needs to support VPE (video port extension to DirectX) and DxApi miniport.

A font driver is a lesser-known kind of graphics driver, which supplies font glyphs to the graphics engine. For example, ATM (Adobe Type Manager) uses atmfd.dll as a font driver. Font files are loaded into kernel address space by the graphics engine and font drivers.

The printer driver is similar to the display driver with some more functions to implement. Unlike other drivers, printer drivers normally do not talk directly with printer hardware. Instead, they write the printer-ready data stream back to the spooler in user mode. The spooler passes the data to the print processor and then to the port monitor, which then uses file I/O to call the kernel mode I/O driver. Windows 2000 allows the printer driver to be either user mode DLL or kernel mode DLL.

Other kernel mode drivers used by the graphics and multimedia system include multimedia drivers like the sound-card driver, and still-image drivers like the scanner driver and camera driver. Windows 2000 DDK provides detailed documentation on kernel streaming drivers (audio, video, video capture) and still-image drivers.

The quality of the kernel mode device driver is crucial to the stability of the whole operating system. A kernel mode driver has read, write access to the whole kernel mode address space and all privileged CPU instructions. So a buggy kernel mode driver could easily damage important data

structures maintained by the operating system, causing the whole system to crash. Therefore, it is critical that any application that includes kernel mode drivers, such as antivirus products, be thoroughly tested to minimize this risk. With Windows 2000, Microsoft ships a Driver Verifier Manager tool (**Verifier.exe** in the system directory) with the OS to help developers verify their drivers.

This section covered the architecture of the Windows NT/2000 graphics and multi media system, a sophisticated yet structured hierarchy of DLLs, user mode drivers, kernel mode DLLs, and kernel mode drivers. The control flow is much more complicated, as in printing, where control cycles between user mode code and kernel mode code several times. We will leave most of the details to the MSDN library, DDK, and other references, while focusing on a few major components important to common Windows applications. In the remaining sections of this chapter, we will unfold what's in GDI, DirectDraw, the display driver, and the printing system including the printer driver.

[< BACK](#) [NEXT >](#)

2.2 GDI ARCHITECTURE

Graphics Device Interface (GDI) is an API designed by Microsoft to provide application programs with a device-independent interface to graphic devices, such as video display, printer, plotter, or fax machine. The GDI interface has been improved considerably between the Win16 API, used in Windows 3.1, and the Win32 API, implemented on Windows 95, 98, NT, and 2000.

The Windows NT/2000 operating systems have a pure 32-bit graphic engine, so the GDI API implemented contains more features than the Windows 95/98 operating systems, which are still based on a 16-bit graphic engine evolved from the Windows 3.1 engine. The exception may be that Windows 95 supports ICM (Image Color Management), while Windows NT 4.0 does not. The new Windows 2000 now supports ICM version 2.0. Windows 98 and Windows 2000 even add new features like alpha blending to GDI.

Microsoft is still planning a new enhancement to Win32 GDI, code-named GDI+. GDI+ provides a better object-oriented interface to the graphics system with many more features.

Exported Functions from GDI32.DLL

GDI provides hundreds of functions callable from Windows programs. Most of these functions are exported by a Win32 subsystem DLL GDI32.DLL. The window management module, USER32.DLL, is a heavy user of GDI functions to draw the tiny details of menus, icons, scrollbars, title bars, and frames of each window. Some of its drawing functions are exported by USER32.DLL to make them available to application programs. Windows 2000 GDI32.DLL alone exports 543 entry points. The dump bin utility, which comes with DevStudio, is a simple tool to list the exported functions of a module. Here is a partial listing of the output generated by dumpbin gdi32.dll /export:

543 number of functions

543 number of names

ordinal hint RVA name

```
1 0 000278B9 AbortDoc
2 1 00027E19 AbortPath
3 2 0001FE0B AddFontMemResourceEx
4 3 0001CE3D AddFontResourceA
5 4 0001FCCC AddFontResourceExA
6 5 00020095 AddFontResourceExW
7 6 0001FE4F AddFontResourceTracking
8 7 00020085 AddFontResourceW
9 8 00026D4E AngleArc
...
533 214 00028106 WidenPath
534 215 00031B4C XFORMOBJ_bApplyXform
535 216 0000F9FE XFORMOBJ_iGetXform
536 217 00031A98 XLATEOBJ_cGetPalette
```

```
537 218 00031AB4 XLATEOBJ_hGetColorTransform
538 219 00031AA6 XLATEOBJ_iXlate
539 21A 0002BD2A XLATEOBJ_piVector
540 21B 000014F9 bInitSystemAndFontsDirectoriesW
541 21C 0000143B bMakePathNameW
542 21D 000015AA cGetTTFFromFOT
543 21E 00026A1F gdiPlaySpoolStream
```

GDI Function Areas

With so many functions, we definitely need to a way to classify Win32 GDI API to understand the structure of GDI. The MSDN library classifies the GDI API into as many as 17 areas, providing quite a good picture of the functionality of GDI.

- *Bitmap*: functions dealing with the creation and drawing of device-dependent bitmaps (DDB), device-independent bitmaps (DIB), DIB sections, pixels, and flood filling.
- *Brush*: functions handling the creation and modification of GDI brush objects.
- *Clipping*: functions handling a device context's drawable region.
- *Color*: palette management.
- *Coordinate and Transformation*: functions dealing with mapping modes, logical to device coordinate mapping, and world transformation.
- *Device Context*: functions dealing with device context creation, attribute query and setting, and GDI object selection.
- *Filled Shapes*: functions drawing a closed area and its perimeter.
- *Fonts and Texts*: functions dealing with the installation and enumeration of fonts on a system and their use to draw text strings.
- *Lines and Curves*: functions drawing straight lines, elliptic curves, or Bezier curves.
- *Metafile*: functions dealing with the generation and playback of Windows-format metafiles or enhanced metafiles.
- *Multiple Display Monitors*: functions enabling the use of multiple display monitors on a single system. These functions are actually exported from user32.dll.
- *Painting and Drawing*: functions dealing with managing a paint message and dirty region for a window. Some of those functions are exported from user32.dll.
- *Path*: functions dealing with combining a series of lines and curves into a GDI object called path and using it in drawing.
- *Pens*: functions dealing with line-drawing attributes.

- *Printing and Print Spooler*: functions dealing with sending GDI drawing commands to hard-copy devices, such as line printers and plotters, and managing such tasks smoothly. Spooler functions are provided by the Win32 spooler, which has several system-provided DLLs and vendor-customizable modules.
- *Rectangles*: functions manipulating a RECT structure, provided by user32.dll.
- *Regions*: functions dealing with describing a set of points using a region GDI object and operations on it.

Besides these well-documented, classified GDI functions, many functions are unaccounted for. Some are documented in DDK, some are undocumented but used by some system DLLs, and others are not used. The following is a rough classification of these functions:

- *User Mode Printer Driver*: functions to support Windows 2000's new user mode printer driver design. These functions are basically sub functions for the kernel mode GDI engine entry points, documented in DDK. For example, the Windows 2000 user mode printer driver may call GDI entry point EngTextOut, which is implemented by a win32k.sys entry point by the same name.
- *OpenGL*: functions to support OpenGL WGL implementation—for example, SwapBuffers, SetPixelFormat, and GetPixelFormat, which are all documented in Windows OpenGL documents.
- *EUDC*: functions to support end-user-defined characters, to allow users to add new characters to fonts. EUDC is documented in the international features section of Platform SDK, under Window Base Services. GDI exports functions like EnableEUDC, EudLoadLinkW, etc.
- *Other system DLL support*: functions known to be used only by other system DLLs. For example, USER32.DLL calls GDI function GdiDlInitialize, GdiPrinterThunk, GdiProcessSetup, etc.; DDRAW.DLL calls GdiEntry1, GdiEntry2, etc.; Spooler service SPOOLSV.EXE calls GdiGetSpoolMessage and GdiInitSpool; WOW32.DLL calls GdiQueryTable and GdiCleanCacheDC.
- *Other undocumented functions*: functions undocumented and not known to be used—for example, GdiConvertDC, GdiConvertBitmap, SetRelAbs, etc.

[Figure 2-2](#) summarizes our understanding of GDI client side architecture. The topmost layer is the documented or undocumented API; underneath it are hundreds of functions grouped into different function areas. On the bottom layers are routines making system calls.

Figure 2-2. GDI function areas.

Documented Win32 GDI API		Un/Semi-documented API
Region	Rectangle (user32.dll)	Other Undocumented API
Pen	Printing (spooler)	Other system DLL Support
Path	Painting & Drawing (user32.dll)	DirectDraw/Direct3D Support
Multiple Display Monitor (user32.dll)	Metafiles (mf3216.dll)	Open GL support
Lines & Curves	Filled Shapes	EUDC
Fonts & Texts	Device Context	User Mode Printer Driver
Coordinate & Transformation	Color	
Clipping	Brush	
Bitmap (msimg32.dll)		

GDI System Service Call Routines

GDI System Service Calls

Among Win32 subsystem DLLs, GDI32.DLL is a relatively small module. Windows 2000 GDI32.DLL is only 223 kilobytes in size, smaller than comdlg32.dll, wow32.dll, icm32.dll, advapi32.dll, user32.dll, and kernel32.dll. GDI implements most of its features by calling the GDI engine through Windows NT/2000 system calls, or native APIs.

Microsoft does not provide public documentation on Windows NT/2000 system calls. Although there are tools that display some of the system calls (Numega SoftICE/W) and third-party documents on system calls (Mark Russinovich's article in www.sysinternals.com/ntdll.htm), there are no public documents and tools on windows graphics and window management system calls or system calls supported by the graphics engine.

With the help of debug symbol files and the image help API, it's not hard to write a program to enumerate all the symbols within a DLL like GDI32.DLL. Those symbols include exported function names, internal function names, and even global variable names. Image help API has a function SymEnumerateSymbols, which can call a user-supplied callback function for each symbol within the module. Given a symbol, we can find its address within the module image and read the binary code starting from the address. Matching the binary code with the system call pattern, we can find all the functions which make system calls.

The SysCall program on the CD is such a program, listing all routines making system calls in a Win32 subsystem DLL. You can list system calls from USER32.DLL, NTDLL.DLL, or GDI32.DLL. Here is a partial list of the 351 system calls (Windows 2000) made from GDI32.DLL, sorted by the system call index number:

```

syscall(0x1000, 1) gdi32.dll!NtGdiAbortDoc
syscall(0x1001, 1) gdi32.dll!NtGdiAbortPath
syscall(0x1002, 6) gdi32.dll!NtGdiAddFontResourceW
syscall(0x1003, 4) gdi32.dll!NtGdiAddRemoteFontToDC
syscall(0x1004, 5) gdi32.dll!NtGdiAddFontMemResourceEx
syscall(0x1005, 2) gdi32.dll!NtGdiRemoveMergeFont

```

```
syscall(0x1006, 3) gdi32.dll!NtGdiAddRemoteMMIInstanceToDC
syscall(0x1007, 12) gdi32.dll!NtGdiAlphaBlend
syscall(0x1008, 6) gdi32.dll!NtGdiAngleArc
...
syscall(0x1125, 11) gdi32.dll!NtGdiTransparentBlt
syscall(0x1126, 2) gdi32.dll!NtGdiUnloadPrinterDriver
syscall(0x1128, 1) gdi32.dll!NtGdiUnrealizeObject
syscall(0x1129, 1) gdi32.dll!NtGdiUpdateColors
syscall(0x112a, 1) gdi32.dll!NtGdiWidenPath
syscall(0x11e5, 3) gdi32.dll!NtUserSelectPalette
syscall(0x1244, 3) gdi32.dll!NtGdiEngAssociateSurface
syscall(0x1245, 6) gdi32.dll!NtGdiEngCreateBitmap
syscall(0x1246, 4) gdi32.dll!NtGdiEngCreateDeviceSurface
syscall(0x1247, 4) gdi32.dll!NtGdiEngCreateDeviceBitmap
syscall(0x1248, 6) gdi32.dll!NtGdiEngCreatePalette
...
syscall(0x1280, 1) gdi32.dll!NtGdiEngCheckAbort
syscall(0x1281, 4) gdi32.dll!NtGdiHT_Get8BPPFormatPalette
syscall(0x1282, 6) gdi32.dll!NtGdiHT_Get8BPPMaskPalette
syscall(0x1283, 1) gdi32.dll!NtGdiUpdateTransform
356 total syscalls found
```

The listing shows the system call index number, the number of parameters required, and the module and the function name of the routine making the system call. The SysCall program also shows function addresses, which we don't have the space to show in a single line.

The central part of the SysCall program is the KImageModule class. Based on the Win32 image help API, an API to process images of Win32 executable files instead of graphical images, the class handles the loading and unloading of modules and their debug symbols files, translation between names and addresses, and enumeration of symbols. Listing of the system call routine is implemented by enumerating all symbols within a module and checking for the fixed pattern of a system service calling routine.

From Win32 GDI API to GDI Engine System Service Calls

With the two listings—the list of exported functions from GDI32 and the list of system service calls made by GDI32, both in alphabetical order—it's easy to figure out or at least make an educated guess as to how Win32 GDI functions are mapped to win32k.sys system calls. For example, the printing function AbortDoc definitely calls NtGdiAbortDoc, which is system call 0x1000; the user mode printer driver support function EngBitBlt is just an alias for NtGdiEngBitBlt, since they have the same address.

Certain Win32 API functions have a simple version, which is easier to use, and an extended version with more features. For example, AddFontResource and Add Font ResourceEx form such a pair. You can guess that Microsoft would not provide two system calls for the two of them. It's a simple matter to make AddFontResource call AddFontResoureEx. For functions with text-string parameters, Win32 API normally has an ANSI string version which ends with "A" and a UNICODE version ending with "W." The Windows NT/2000 system call has only the UNICODE version, for the OS is UNICODE based. You may have noticed that the three AddFontResource XXX calls have only a single corresponding system call, NtGdiAddFontResourceW.

If you look at feature areas in comparing the GDI exported functions list and the GDI system service calls, you will

find that a certain area of GDI functionality is implemented purely in user mode GDI client, without making simple pass-through calls to the GDI engine. One good example is Windows metafile and enhanced metafile handling, which does not have any trace in the GDI engine system calls. The same applies to functionalities based on EMF—for example, print EMF despooling functions like GdiStartDocEMF, GdiStartPageEMF, GdiPlayPageEMF, etc.

Various Win32 API calls to get and set device context attributes—for example, GetBkMode, SetTextColor, etc.—are also missing in system calls. Their closest system calls may be the generic NtGdiGetDCDword and NtGdiGetAndSetDCDword. We will learn later that some device context attributes are kept in user mode memory for easy access, while others are stored in the kernel mode data structure.

To summarize, Win32 subsystem DLL GDI32.DLL implements Win32 GDI API by mostly morphing Win32 API calls to system service calls, which are implemented by the GDI graphics engine in win32k.sys. A few areas like metafile, enhanced metafile, and EMF print despooling are true new features provided by GDI32.DLL without direct help from the GDI engine. GDI client DLL also supports implementation for other system components like DirectDraw, Direct3D, OpenGL, and printing and spooling.

[< BACK](#) [NEXT >](#)

2.3 DIRECTX ARCHITECTURE

While the GDI API has been making most application programmers happy with its features and performance, Microsoft struggled for quite some time to make game programmers feel the same way. What game programming wants is performance, which is not what a device-independent API like Windows GDI was designed for. Microsoft tried DrawDIB API (part of video for Windows), WinG (a small library to speed up bitmap display), WinToon (an animated sprite engine), Game SDK, and finally came up with DirectX.

DirectX is an API provided by Microsoft to build a new generation of high-performance computer game and multimedia applications. DirectX also comes with a DirectX DDI interface, which defines the features to be implemented in hardware vendor-supplied display drivers. So DirectX serves two very important purposes. On the API level, DirectX allows game/application developers to get a powerful device-independent API without comprising performance. Application programmers can take advantage of hardware advances without worrying about interfacing with the hardware directly. On the device-driver level, DirectX allows hardware vendors to concentrate on hardware innovations and bring them easily to market with a thin layer of Direct X-aware drivers. The DirectX DDI interface provides hardware vendors with guidelines to hardware advancement that can easily be integrated into DirectX.

DirectX Components

DirectX is composed of several major components that address different areas of game and multimedia programming. Here is what DirectX looks like currently:

- DirectDraw provides a two-dimensional, high-performance interface to the display card, which supports direct video memory access, fast blitting (blit: bit block transfer), back buffering, and flipping, palette management, clipping, overlay and color keying. DirectDraw can be seen as a subset of GDI specially designed for display monitors with performance enhancements.
- DirectSound accelerates wave-audio sound (digital samples) capture and playback, with low-latency mixing and direct access to sound devices.
- DirectMusic converts message-based musical data to wave samples, either through hardware or in a software synthesizer. The wave samples are then streamed to DirectSound.
- DirectPlay facilitates communication between multiple players in multiple-player games over a modem link or network. It provides a uniform way for DirectX applications to communicate with each other independent of the actual protocol, transport, and online services.
- Direct3D provides two levels of 3-D games API—Direct3D Immediate Mode and Direct3D Retained Mode. Direct3D Immediate Mode is a low-level 3D API ideal for experienced developers porting existing games and multimedia applications to DirectX. Direct3D Retained Mode is a high-level API allowing easy implementation of 3D graphical application, which is based on Direct3D Immediate Mode. Direct3D provides switchable depth buffering, flat and Gouraud shading, multiple lights and light types, material and texture support, transformation, and clipping. Direct3D Retained Mode development has been stopped, waiting to be replaced by something newer.
- DirectInput supports interactive input devices including mouse, keyboard, joystick, force-feedback device, and other game controllers.

- DirectSetup provides a simple API for installation of DirectX components. Game and multimedia applications can also make use of the AutoPlay feature, which starts an installation program or game automatically from a CD when it's inserted.
- DirectShow enables the playback of compressed audio and video content in various formats, including MPEG, QuickTime, AVI, and WAV. New formats can be added by adding plug-ins called filters, which are managed by the Direct Show filter graph manager.
- DirectAnimation supports building animation in a variety of environments such as HTML, VBScript, Jscript, Java, and Visual C++. It supports vector graphics, images and sprites, 3D geometry, video, and sound in a unified single animation API. DirectAnimation also provides several client Media Play Controls that expose methods and properties for manipulating multimedia playback from a web page or application.

[Figure 2-3](#) shows the architecture of DirectX, with several small components omitted for lack of space. On the lowest level, DirectX calls GDI to call system services. Based on these system services, DirectDraw, DirectSound, DirectMusic, Direct3D Immediate Mode, and Direct3D Retained Mode are built, each of them exposing its functionalities through a set of COM interfaces. DirectShow and Direct Animation are built on top of these more basic DirectX components, which also depend on various filters. On the topmost layers are various games, multimedia applications, Java applets, web pages, etc.

[Figure 2-3. Main DirectX architecture.](#)

DirectX games, multimedia applications, Java applets, HTML pages, etc.

Browser Plug-ins	DirectAnimation Client Controls	
Media Player Controls	DirectAnimation (danim.dll)	

DirectShow Filter Graph Manager		
Source Filter	Transform Filter	Renderer Filter
(more)	(more)	IDirect3DRMIMaterial2
IDirect3DRMLight	IDirect3DExecuteBuffer	IDirect3DRMLight
IDirect3DRMDevice3	IDirect3DDDevice	IDirect3DRMDevice3
IDirect3DRM3	IDirect3D3	IDirect3DRM3
(more)	(more)	(more)
IDirectMusicComposer	IDirect3DLight	IDirect3DRMMaterial2
IDirectMusicCollection	IDirect3DExecuteBuffer	IDirect3DRMLight
IDirectMusicLoader	IDirect3DDDevice	IDirect3DRMDevice3
IDirectMusic	IDirect3D3	IDirect3DRM3
(more)	(more)	(more)
IDirectSoundCapture	IDirect3D	Direct3D
IDirectSound3DBuffer	Immediate Mode	Retained Mode
IDirectSoundBuffer	(d3dim.dll)	(d3drm.dll)
IDirectSound	DirectMusic	Mode
DirectDraw (ddraw.dll, ddrawex.dll)	DirectSound (dsound.dll)	(d3dm.dll)

GDI & other OS Services

Each component of DirectX is presented to application programmers through one or more Win32 subsystem DLLs with easily recognizable names. For example, ddraw.dll and ddrawex.dll implement DirectDraw API; d3dim.dll implements Direct3D Immediate Mode API; and d3drm.dll implements Direct3D Retained Mode API.

Unlike the traditional Win32 API that is composed of hundreds of C or Pascal-like functions, DirectX uses Microsoft's Component Object Model (COM) interfaces to expose DirectX API. A COM interface is an ordered group of semantically related functions, with predetermined parameter types and function return types. In the C programming paradigm, a COM interface can be thought of as a function table; in the C++ world, a COM interface is an abstract base class.

COM interfaces are implemented by COM classes. But the COM rationale is to separate the implementation strictly from the interface, so client-side programs can create only COM class instances, also called COM objects, in a controlled manner and operate on them through COM interfaces.

Once a COM interface is published, it's frozen. That is to say, the interface definition can't be changed, although you can freely change its implementation. The only way to expose more features to applications is designing and publishing new interfaces. That's the reason you will see interface names like IDirectDraw, IDirectDraw2, and IDirectDraw7.

Figure 2-3 shows a few COM interfaces supported by several DirectX components. Most DirectX components define far too many interfaces to be listed here.

DirectDraw Architecture

The focus of this book is two-dimensional Windows graphics programming—in other words, GDI and DirectDraw. We leave the rest of DirectX to MSDN documents, other books, and online resources. Let's take a closer look now at DirectDraw architecture.

DirectDraw can be seen as a specialization of GDI. The first step of this specialization is limiting its target to computer display cards, not printers, plotters, or other imaginable graphics devices. The second step is reducing the feature set supported by GDI. DirectDraw does not have direct support for mapping mode, world transformation, font and text, lines and curves; it deals only with bitmaps. The last step is implementing the limited feature set with provisions for hardware acceleration and adding features that are critical to high-performance game and multimedia programming.

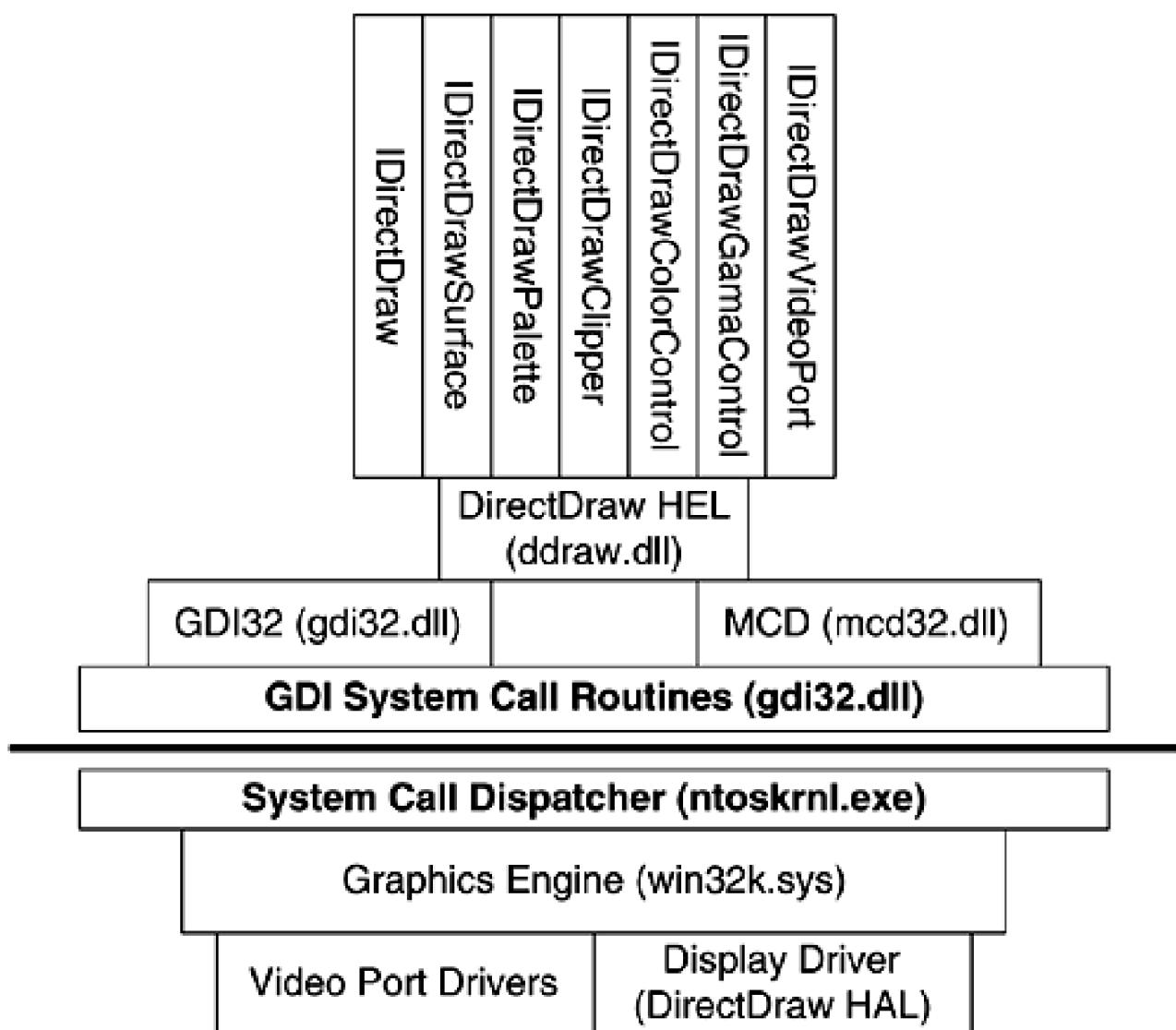
DirectDraw implements seven major interfaces, two of them having multiple versions.

- IDirectDraw is the basic DirectDraw interface, from which other DirectDraw objects can be created. The latest version is IDirectDraw7. The IDirect Draw interfaces support the creation of other DirectDraw objects, surface management, display modes, display status, memory allocation, etc. Calling Direct DrawCreate creates a DirectDraw object that supports various IDirect Draw interfaces.
- IDirectDrawSurface is where the DirectDraw drawing all occurs. Its latest version is IDirectDrawSurface7. IDirectDrawSurface supports surface management activities like capability query, locking, unlocking, setting palette and clipper, etc. Surface locking maps video display memory to an application's virtual address space, which allows direct access to and attaching of the GDI device context to the DirectDraw surface and using GDI calls on it. Most importantly, it supports surface blitting and flipping with color keying with hardware acceleration. For more complicated drawing, you can either implement it yourself or turn to GDI for help.
- IDirectDrawPalette interface supports the creation and direct manipulation of the color palette for 256-color displays.
- IDirectDrawClipper interface manages clipping for DirectDraw surfaces using clip lists, which are represented by the GDI API's RGNDATA structure. As DirectDraw provides no support to create those clip lists, you can use GDI's rich set of region operations to help.
- IDirectDrawColorControl interface adjusts color for surfaces and overlays through adjusting color brightness, contrast, hue, saturation, sharpness, and gamma.
- IDirectDrawGammaControl interface manages red, green, and blue gamma ramps, which map the frame buffer color values to colors that will be passed to the hardware digital-to-analog (DAC) converter.
- IDirectDrawVideoPort interface enables channeling live video data from a hardware video port to a DirectDraw surface. It gives programmers a way to control the hardware via the video port.

Figure 2-4 shows the architecture of DirectDraw, both the user mode and kernel mode components. The user mode component of DirectDraw is ddraw.dll, which is related to gdi32.dll and mcd32.dll (OpenGL). DirectDraw calls goes through GDI32.DLL to make a system service call, which is served first by the system call dispatcher in kernel mode address space. The dispatcher passes the call to the graphics engine (win32k.sys) and then to either third-party provided display drivers or video port drivers. DirectDraw is not an across-the-board enhancement for GDI, since its focus on video display and DirectDraw's limited function set make DirectDraw an application relying on GDI for

support, especially in curve drawing, region manipulation, fonts, and texts. Having a deep understanding of how GDI is implemented also helps to simulate GDI using DirectDraw features.

Figure 2-4. DirectDraw architecture.



Microsoft documents and other documents often picture DirectDraw alongside GDI, both having direct access to the hardware through a device-dependent abstraction layer. Some books even claim that with DirectDraw you don't need GDI any longer. In reality, DirectDraw API is implemented by DDRAW.DLL, which relies on GDI32.DLL to interface with the GDI engine and then DirectDraw device drivers. DDRAW.DLL imports several undocumented GDI exported functions, almost every one from GdiEntry1 through GdiEntry15. In [Section 2.1](#), we mentioned a program SysCall to list the system service calls made in system DLL like GDI32.DLL. If you look at the GDI32 system service call list, you will find dozens of DirectDraw and Direct3D system service calls—for example, NtGdiDdCreateSurface, NtGdi D3dTextureSwap, and NtGdiD3dDraw-Primitives2. Apparently, GDI itself has no use for those functions. So the only explanation is that GDI exposes those DirectDraw/Direct3D system service calls to DDRAW.DLL and other DirectX DLLs through undocumented entry points.

DirectDraw is implemented in several layers. The top layer supports the DirectDraw COM interfaces, standard COM exported functions (DllGetClassObject, etc.), and special DirectDraw create routines (DirectDrawCreate, etc.) The middle layer is the hardware emulation layer (HEL), which simulates all or some of the DirectDraw features not supported by the hardware. The bottom layer, called the hardware abstraction layer (HAL), talks directly with the display hardware.

But where are DirectDraw HEL and DirectDraw HAL? Actually, the DirectDraw hardware emulation layer is the major part of DDRAW.DLL, a 32-bit user mode DLL. You can figure this out in several ways. First look at the size of DDRAW.DLL, which is 248 kilobytes, a little bit larger than GDI32.DLL. This tells you that DDRAW.DLL is much more than a thin API layer. Second, look at the DDRAW import function list; you will find GDI functions like CreateDIBSection, StretchDIBits, PatBlt, BitBlt, etc. So DDRAW is doing some serious drawing using GDI functions. Last, use a tool which can list names in the debug information files—for example, Visual C++ debugger. You will find names like HELBlt, HELInitializeSpecialCases, and generalAlphaBlt. You can also find lots of names with “mmx,” which refers to Intel CPU’s multimedia extension instruction set. So DDRAW.DLL provides DirectDraw HEL with special optimization for MMX. Implementing DirectDraw HEL in user mode makes sharing code between Windows NT/2000 and Windows 95/98 much easier. It also has little trouble using floating-point instructions and MMX instructions, which are not supported very well in OS kernel mode. Needless to say, DirectDraw crashes the system less often with more user mode code.

DirectDraw HAL is nothing other than the device driver provided by the display card vendor that supports the DirectDraw DDI interface. Please remember that the DirectDraw API layer and HEL, which are both in user mode, do not have direct access to DirectDraw HAL, which is in kernel mode on Windows NT/2000. You have to go through GDI system service call routines, which are served by code in the GDI engine (WIN32K.SYS), before finally reaching DirectDraw HAL.

We will present many more details about DirectDraw in this book. [Section 3.6](#) explores the DirectDraw internal data structure. [Section 4.3](#) sheds more light by spying on DirectDraw interfaces [Chapter 18](#) is all about DirectDraw.

[< BACK](#) [NEXT >](#)

2.4 PRINTING ARCHITECTURE

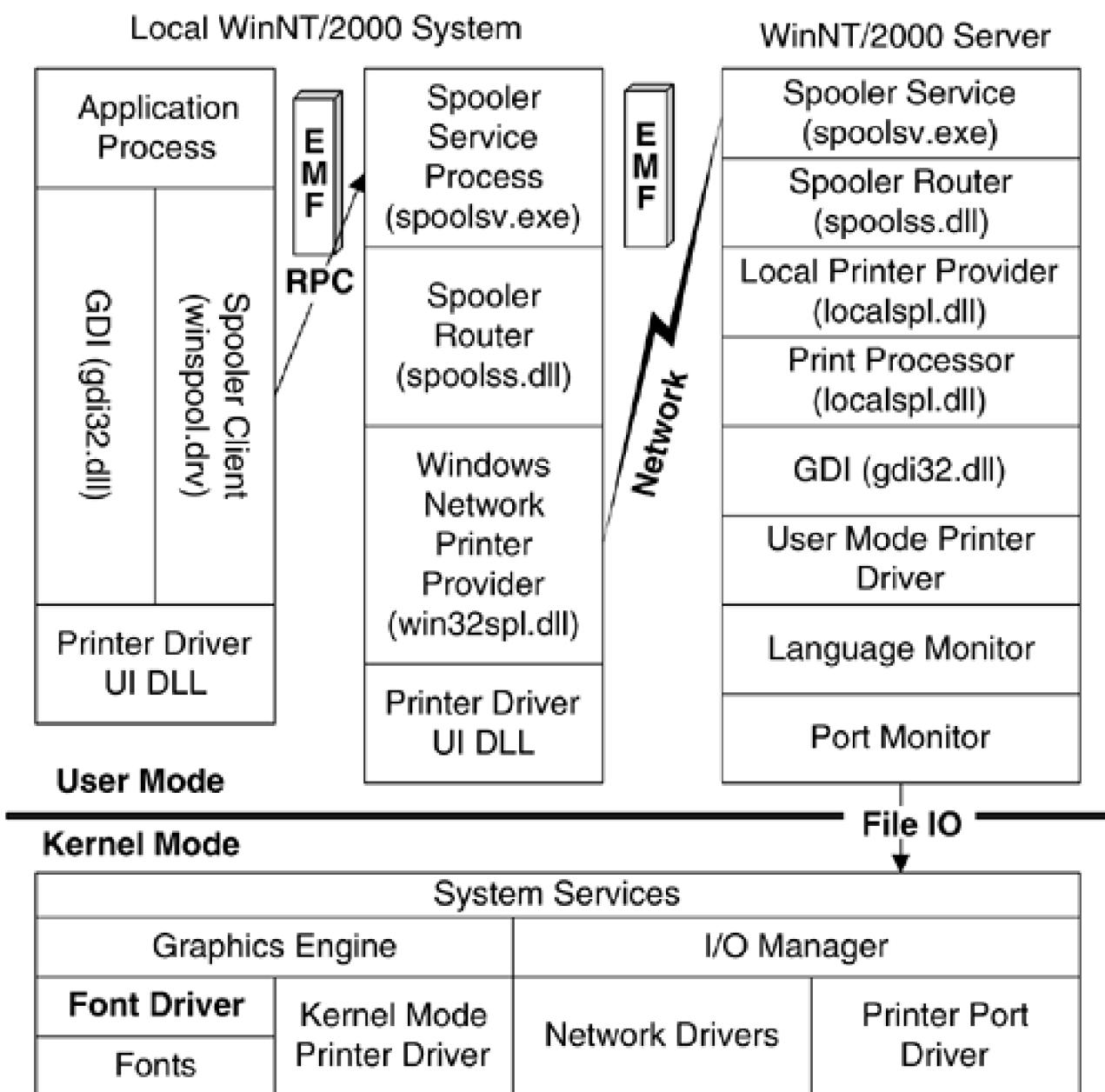
The Win32 GDI API is supposed to be a device-independent API which can draw lines, curves, bitmaps, and texts to any graphics device you have a driver for. But printers are a special class of graphic devices that need special treatment. Printers differ from other graphic devices in several ways.

- Users do not print a page at a time; they print a whole document, with special formatting requirements like quality mode, paper size, duplex, n-up, banner, multiple copies, etc. GDI provides a printer-specific API to handle the page boundary within a document, and a DEVMODE structure to specify all print settings. The application needs to pay attention to issues like pagination and paper margins.
- Printers use much higher resolution (300 to 2400 dpi) than do screen displays (75 dpi to 120 dpi), which means there is much more data to process and not enough memory to render a page at a time. The GDI engine allows a printer driver to accept data a small band at a time through EMF (enhanced metafile) spooling.
- Printers are normally slow and are shared by a group of people, and they may not be directly attached to the PC. The Windows OS provides a sophisticated spooler to ensure that the application can finish its share of printing as soon as possible, that multiple jobs can share a printer, and a group of people can share a printer in a locally shared environment, across a network, or even through a URL address.
- Printers speak different languages—PCL for HP printers, ESC/P for Epson printers, Postscript for Postscript printers, and HPGL for plotters. This is totally different from display cards, which are all raster based. Microsoft provides several “universal” drivers that can be customized by hardware vendors to meet their special hardware requirements.

Windows NT/2000 printing architecture centers around a print spooler, with support from GDI and a set of printer drivers. User applications call API entry points exported by GDI and the spooler client DLL to start a printing job. GDI and the spooler, with help from printer drivers, handle the print job and send the data to any of a variety of hard-copy devices, including laser printers, ink-based printers, plotters, and fax machines.

Drawing commands are sent to GDI through GDI API calls, which are normally saved in an enhanced metafile (EMF). The EMF and another file with the current printing setting are passed to the spooler system service process (spools.exe). At this point, the application is done with the printing of the document. The user is free to use the application to continue other tasks, while the spooler is making sure the document actually gets printed. First the spooler needs to route the job to a printer provider, which handles the actual printer. The printer could be a local printer, which is handled by the local printer provider (localspl.dll), or it could be accessed over a Windows network, which is handled by the Windows network printer provider (win32spl.dll). If the printer is on a remote machine, OS network services are used to send the spool files to the remote machine, where they enter its spooler as a job for a local machine. [Figure 2-5](#) illustrates the architecture of the Windows NT/2000 printing system.

Figure 2-5. Windows NT/2000 printing architecture.



Finally, the local print provider gets the print job, which is passed to a print processor. The print processor checks the format of the spool file. For EMF files, it plays each page back to GDI, which breaks down the GDI commands into drawing primitives defined by the DDI interface (device driver interface) and passes the drawing primitives to the printer drivers. The printer drivers render the drawing primitives into raw data in printer language—for example, PCL, ESC/P, or Postscript. The raw data are sent to the spooler's print processor again.

When the print processor sees the raw data, which is ready to be sent to the printer, it passes them to a language monitor. A language monitor passes the data to a port monitor, which uses the OS file system API to write the data to the hardware port. Any more firmware on the printer side is not an issue for us; we just assume that you get your document perfectly printed after this long chain of software components.

Let's take a look now at the Windows NT/2000 printing system's components in more detail.

The Win32 Spooler Client DLL

The Win32 spooler client DLL (winspool.drv) exposes the spooler API to the user applications. The user application

uses the spooler API to query printers, print jobs, change printer settings, query printer settings, load the printer driver user interface DLL to display printer specific settings property sheets, and do other things. For example, you can use OpenPrinter, WritePrinter, and ClosePrinter, which are part of the spooler API, to send raw data to a printer without doing the normal GDI drawing, printer driver rendering routine.

The spooler API is defined in the header filer winspool.h, with winspool.lib being its library file. So winspool.drv is loaded into the application process if printing is required. The spooler client DLL helps GDI to determine how a print job should be handled. For normal jobs, GDI generates an EMF file and passes it to the spooler client, which uses a remote procedure call to send the job to the spooler system service process.

Spooler System Service Process

The Windows NT/2000 spooler is implemented as a system service, a process having special privileges and responsibilities in the system. The spooler service is started when the operating system is started. That's why sometimes your printer starts printing automatically during a system reboot if you have an unfinished printing job when the machine is shut down.

You can use **net stop spooler** to stop the spooler process, and **net start spooler** to restart the spooler process. Once the spooler is stopped, you will find your control panel printer applet stops working.

The spooler service exports an RPC (remote procedure call) interface to the spooler client DLL, which the user application can use to manage printers, printer drivers, and printing jobs. The spooler service itself is a small EXE file (spoolsv.exe). It passes most of its calls to printer providers through a spooler router.

The spooler service is a system component that can't be replaced.

Spooler Router

The printer you're printing to may be your local printer, may be somewhere on a Microsoft network, may be on a Novell server, or may be anywhere in the wide world, having only a URL address. The spooler system service uses the spooler router DLL (spool ss.dll) to hand the printer job to a printer provider, which knows where to send the job.

Spoolss.dll provides exported functions similar to winspool.drv. You can find AddPrinter, OpenPrinter, EnumJobW, etc., in both of them. It is quite common in printer spoolers for the same call to be passed from one module to another, to still another, until it finally reaches its destination.

The job of the router is quite simple: to find the right print provider and pass along the information. It does that through the printer name or printer handle information which comes with the printing job with the help of lots of printer-related settings in the system registry.

When a user application calls OpenPrinter to the spooler client DLL winspool.drv, the call passes to the spooler system service (spoolsv.exe). It calls the spooler router, which calls each printer provider's OpenPrinter, until one of them supplies a handle that means the printer provider recognizes the printer name. This handle is returned to the application so that subsequent calls can be directed to the correct printer provider.

The spooler router is a system DLL that can't be replaced.

Print Provider

The print provider is responsible for directing printing jobs, either to the local machine or to a remote machine. It also manages print-job queue operations, such as starting, stopping, and enumeration of jobs.

Unlike the spooler service process and the spooler router, there can be multiple print providers in the system. Hardware vendors can even write their own printer provider using Windows NT/2000 DDK.

The operating system provides several print providers:

- Local Print Provider (localspl.dll). Handles local print jobs or jobs sent from remote clients to local machines. Ultimately, every job gets handled by a local print provider, which passes the job to a print processor. Windows 2000 puts the default print processor into the local print provider DLL.
- Windows Network Print Provider (win32spl.dll). Passes print jobs to a remote Win32 server.
- Novell Netware Print Provider (nwprovau.dll). Passes print jobs to Novell Netware print servers. Because Novell servers do not handle EMF files, print data must be rendered into raw data before being sent to a Novell server.
- HTTP Print Provider (inetpp.dll). Passes print jobs to URL addresses.

All print providers must implement some required routines as specified in DDK, so that the spooler router can treat them the same way. Other routines are optional. Windows 2000 DDK provides source code for a sample print provider in the `src\print\pp` directory. The main entry point of a print provider is `InitializePrintProvider`. Other entry points are exposed through a function table returned by it.

The local print provider needs to implement the full set of print provider functions, including despooling print jobs, invoking the printer driver user interface DLL, and calling print processors to process the job.

Print Processor

Print processors are responsible for converting a print job's spooled file into raw data format which can be sent to printers. They are also called to control print job pausing, resuming, and canceling requests. A print processor is called by the local printer provider (localspl.dll).

A print job's spooled file is normally in EMF format on Windows NT/2000. GDI helps record an application's drawing request into EMF format and write it quickly to disk to allow the application to return to normal operation. Spool files are normally stored in the `$SystemRoot$\spool\printers` directory. For an EMF spool job, the spooler generates a pair of files. The file with `.shd` extension contains job setting information like printer name, document name, port name, and a copy of the `DEVMODE` structure. The file with `.spl` has an undocumented header, some embedded fonts, and one EMF page for each page in the document to be printed.

Windows 2000 includes two default print processors:

- Window Print Processor (in localspl.dll) supports several spool formats including NT EMF, RAW, and TEXT.
- Macintosh Print Processor (in sfmpsprt.dll) supports PSCRIPT1 spool format.

The EMF spool format is the normal spool file format for all Windows applications. An EMF spool file is usually much smaller than the raw printer-ready data. Generating an EMF spool is mainly done by GDI with minimum printer driver involvement. If you are printing over a network, the spool EMF data sends smaller data over the network. It also frees your client machine to do its normal job, while the server machine is busy converting EMF to raw printer data. During EMF generation, GDI queries the remote machine for font availability. If certain fonts are missing on the remote machine, they will be embedded in the .spl spool file, shipped to the remote machine, and installed.

If you choose RAW data type, the printer-ready data will be generated on the client machine instead of the printer server, which means higher loads on a network, but lower load on server memory and CPU usage. A spool file in Postscript file format for a Postscript printer is considered RAW data because no conversion is needed for the printer.

A TEXT format spool file consists solely of ANSI text. A print processor is responsible for rendering the text string into a format the printer can accept. It asks GDI and the printer driver to do so. The TEXT format spool file may be useful for DOS applications.

The PSCRIPT1 spool format and the sfmpsprt print processor are not for Postscript printers. A PSCRIPT1 spool file is in the Postscript format, and the sfmpsprt the print processor converts it to the RAW format for a printer. So sfmpsprt is actually a Postscript interpreter.

A printer processor does not have direct access to those spool files, and their formats are not documented. Printer processors rely on GDI and the spooler client (winspool.drv) API to render them into raw data.

You are not limited to the two print processors provided by Microsoft. Microsoft provides documentation and a working sample print processor with Windows NT/2000 DDK. In the Win2000 DDK src\print\genprint directory, there is a sample EMF print processor. You can compile the code, copy its binary to the \$SystemRoot\$\system32\spool\prtprocs\w32x86 directory, write a small program to call AddPrintProcessor to install it, and then play with your own copy of print processor in a debugger. Windows NT 4.0 DDK provides a more complete print processor source code that handles EMF, RAW, and TEXT spool formats.

The main entry points of a print processor are OpenPrintProcessor, PrintDocumentOnPrinterProcessor, and ClosePrintProcessor. OpenPrintProcessor initializes a print processor for a print job; PrintDocumentOnPrintProcessor handles a print job; and ClosePrintProcessor frees the memory allocated by OpenPrintProcessor.

For an EMF spool file, Windows NT 4.0 GDI provides a single function Gdi PlayEMF to render a whole document. Windows 2000 provides a much richer, yet still protective API to let the print processor query for individual pages within an EMF file, change the order of EMF page rendering, combine several logical pages into one physical page, and add world coordinate transformation in playing back EMF files. For example, within PrintDocumentOnPrintProcessor, you can use:

- GdiGetPageCount to get the number of pages in the document, which waits for the whole document to be spooled in EMF.
- GdiStartPageEMF to start the rendering of a physical page.
- GdiGetSpoolPageHandle to randomly seek the last EMF page.
- GdiPlayPageEMF to play four logical pages each into one-fourth of a physical page.

- GdiEndPageEMF to finish the rendering of a physical page, which actually calls the printer driver.

What this sequence achieves is called reverse-order *n*-up printing—that is, printing the document in back-to-front order, with *n* (= 4 here) logical pages mapped into a single physical page. With this Windows 2000 print processor design, you don't have to implement these document-formatting features in each and every printer driver. You just need one print processor, which can be used as a preprocessor for all compatible print drivers.

The EMF print processor for Windows NT 4.0 is in a separate DLL, winprint.dll. Windows 2000 merges its functionality into localspl.dll. You can find PrintDocument OnPrintProcessor in localspl.dll's exported function list. To change the print processor setting for a printer driver, go to the printer driver's property page and select the Advanced tab; you will find a "Print Processor ..." button.

There are several ways data may go from a print processor. For RAW spool data, the print processor uses a WritePrinter call (check Windows NT 4.0 winprint sample, raw.c). In this case, the data goes directly to the language monitor. For TEXT spool data, the print processor does a new StartDoc call and sends the GDI drawing command to the printer driver. The printer driver is responsible for calling WritePrinter. For EMF spool data, the print processor calls GdiEndPageEMF, which uses GDI's EMF playback mechanism to send recorded draw commands to the printer driver. Again, the printer driver calls WritePrinter.

Language Monitor and Port Monitor

Print monitors are responsible for directing raw print data from the spooler to the right port driver. There are two types of print monitors—a language monitor and a port monitor.

Here language does not mean English or even C++. It means the different kinds of printer job languages, like PJL, understandable by printer firmware. The main purpose of a language monitor is to provide a full-duplex communication channel between the print spooler and the printers, linked through a cable capable of bidirectional communication. The data path from computer to printer is mainly for sending print data to the printer. The back channel from printer to computer is for providing feedback information. The spooler, printer drivers, and even the user applications may be interested in the exact features and status of the printer (for example, on-printer RAM installed, RAM available, printer options installed, ink level, etc). A language monitor may be able to supply this information through the standard DeviceloControl call. The second purpose of a language monitor is to insert printer control commands to the printer data stream.

A port monitor works underneath a language monitor to provide a communication path between the spooler and the kernel mode port drivers which actually access the hardware I/O port linked to the printers. Being user mode DLLs, a port monitor does not have access to hardware directly. It uses the normal file API, CreateFile, WriteFile, ReadFile, and DeviceloControl to communicate with the drivers in the OS kernel.

Port monitors are also responsible for managing the logical printer ports on your computer; for example, localmon.dll supports all your local machine's COM and LPT ports. So when your application writes data to LPT1, it's not talking to a hardware port driver directly. It's actually talking to a pipe created by the spooler, managed by a port monitor. If you use the SDK tool Winobj, you can find that "\DosDevices\lpt1" is a symbolic link for "\Device\NamedPipe\Spooler\lpt1".

Windows 2000 provides quite a few printer monitors: pjmon.dll for lots of HP printers supporting PJL job-control language; tcpmon.dll for monitoring the network port; faxmon.dll for the fax driver; and sfmon.dll. DDK also provides sample source code for language/port monitors to vendors to implement their own print monitors.

A Peek into the Spooler Process

This section gives a brief introduction to the Windows NT/2000 printing architecture, mainly focusing on the spooler part. We will say more about printing API in [Chapter 17](#), and more about printer drivers in [Section 2.7](#).

If you really want to see for yourself what's in the spooler system service process, where all these dramas take place, you can easily do so with Visual C++ studio. Follow these simple steps:

- Press Ctrl+Alt+Del keys at the same time; the Windows Security dialog box will pop up. Select Task Manager.
- In the process list, select the spooler service process (spoolsv.exe), right-mouse click, and select the “Debug” option; now you're debugging the spooler system service process.
- Watch the symbol files for dozens of modules to get loaded, or just check the VC 6.0 module list; you can find the spooler client DLL, spooler service, router, print providers, print processors, language monitors, port monitors, and other modules not mentioned here.
- Start a print job from the control panel; you will find that the printer driver user interface DLLs and the user mode printer drivers get loaded and threads get created and finished. For example, UNIDRV.DLL is the Microsoft UniDriver that is widely used as a user mode printer driver under Windows 2000; UNIDRVUI.DLL is its user interface DLL.
- If you close Visual C++ Studio, thereby closing the spooler service process, remember to restart it using **net start spooler**.

[Figure 2-6](#) shows some of the modules loaded into the spooler service process after a print job, using Visual C++ 6.0's module list feature. Fifty-five modules are loaded into that process. Vendor-provided printer-driver components are normally loaded during printing and then unloaded after the job is done.

Figure 2-6. Modules loaded into the spooler service process.

Module	Path	Orc
tcpmon.dll	D:\WINNT50\system32\tcpmon.dll	40
usbmon.dll	D:\WINNT50\system32\usbmon.dll	41
msfaxmon.dll	D:\WINNT50\system32\msfaxmon.dll	42
sfmpspri.dll	D:\WINNT50\system32\spool\prtprocs\w32x86\sfmpspri...	43
rnr20.dll	D:\WINNT50\system32\rnr20.dll	44
winrnr.dll	D:\WINNT50\system32\winrnr.dll	45
nwprovau.dll	D:\WINNT50\system32\ nwprovau.dll	46
mpr.dll	D:\WINNT50\system32\mpr.dll	47
win32spl.dll	D:\WINNT50\system32\win32spl.dll	48
clbcatq.dll	D:\WINNT50\system32\clbcatq.dll	49
oleaut32.dll	D:\WINNT50\system32\oleaut32.dll	50
inetpp.dll	D:\WINNT50\system32\inetpp.dll	51
icmp.dll	D:\WINNT50\system32\icmp.dll	52
UNIDRMUI.DLL	D:\WINNT50\system32\spool\drivers\w32x86\3\UNID...	53
UNIDRV.DLL	D:\WINNT50\system32\spool\drivers\w32x86\3\UNID...	54
mscms.dll	D:\WINNT50\system32\mscms.dll	55

[Close](#)

[< BACK](#) [NEXT >](#)

2.5 GRAPHICS ENGINE

In [Section 2.1](#) we showed a diagram of the Windows NT/2000 graphics system architecture. In it is a big block named Graphics Engine. In the previous discussions on GDI, DirectDraw, and Direct3D, we mentioned that they all call system service call routines provided by GDI32.DLL, which are served by the graphics engine. Now let's take a closer look at the Windows NT/2000 graphics engine, the backbone of GDI, gateway to the graphics device drivers, and support base for those drivers.

The Windows NT/2000 graphics engine lives in a kernel mode DLL which also implements OS Windows Management functionalities—that is, WIN32K.SYS. During the initial implementation of Windows NT, security was a major concern for operating-system design, which favors a smaller, less involved, and more stable kernel component design. Before Windows NT 4.0, the graphics engine and window management was a user mode DLL as part of the Win32 subsystem process (csrss.exe). If an application makes a window management or graphics drawing call, a local procedure call (LPC) is made from the application to the Win32 subsystem process. The latter calls the graphics engine or window management in its thread and returns the result to the application. Significant CPU and memory resources were used in the process and thread switching in this design. Windows NT 4.0 and the new Windows 2000 move the graphics engine and window management down to kernel mode. Now USER32.DLL or GDI32.DLL make only system service calls, which are dispatched by NT OS KRNL to WIN32K.SYS without process and thread switching.

So WIN32K.SYS can be seen as the kernel backbone implementation of two important modules in Windows OS: USER32.DLL and GDI32.DLL. WIN32K.SYS itself is a huge DLL (1640 KB, Windows 2000), even larger than NTOSKRNL.EXE (1465 KB, Windows 2000). The internal architecture of WIN32K.SYS is barely known to the outside world. Microsoft needs to document only one thing: the DDI interface which the graphics device driver relies on. WIN32K.SYS exports around 200 functions in comparison with NTOSKRNL.EXE, which exports around 1200 functions. On www.sysinternal.com, there is a complete listing of the Windows 2000 beta 1 OS kernel source code tree, developed from assert messages from the OS debug build. But it's only for NTOSKRNL.EXE; nothing similar exists for WIN32K.SYS.

Luckily, the Microsoft NT/2000 DDK document on the DDI interface—that is, the interface between the graphics engine and the graphics device drivers—is very well written. Based on the DDK documentation and our own learning, [Figure 2-7](#) shows how the graphics engine may look architecturally.

Figure 2-7. Windows 2000 graphics engine architecture.

Graphics Engine			
Graphics Engine System Service Table			
GDI API Layer			DirectDraw
MCD	MCD Server	Direct3D	
Halftone	Floating Point Simulation	Bitmap Font Driver	
Raster Font Driver	TrueType Font Driver	Scaler	
Render Engine			
GDI Handle Manager			

Third Party Drivers			
Printer Driver	Font Driver Fonts	Video Miniport (VPE, DxApi, TV)	Display Driver (DirectDraw, Direct3D, MCD)

There are several layers in the graphics engine architecture. The top layer is the system service table, which forms its sole entrance from user mode applications. Below it, DirectDraw, Direct3D, and OpenGL have much more direct paths to the display drivers. For normal GDI calls, there is a GDI API layer that transforms GDI constructs into primitives understandable by the DIB render engine and the graphics device drivers. The GDI API layer uses the GDI handle manager to manage internal data structure, the render engine to render DDI primitives to bitmap surfaces supported by GDI, the scaler to implement transformation, three font drivers for fonts supported by GDI, and other components. Unlike Windows 95/98, Windows NT/2000's render engine is powerful enough to render all DDI primitives to surfaces using standard DIB formats without the help of graphics device drivers. For non-standard bitmap surfaces, the graphics engine calls the device drivers supporting them to render DDI primitives. These drivers may call back to the graphics engine, either to query for more information, to ask the graphics engine to further break down commands, or even to ask the render engine for help in drawing the commands.

Now let's examine the Windows 2000 graphics engine in more detail.

Graphics Engine System Services

In [Section 2.2](#) on GDI we mentioned that GDI32.DLL provides hundreds of small routines that make system service calls. These system service call routines are used by Win32 subsystem DLLs, namely GDI, DirectDraw, Direct3D, and OpenGL, to call the graphics engine living in the OS kernel. We even showed a program to list all the system service calls made from those DLLs.

WIN32K.SYS is the actual module serving these system calls. It has a table of system service routines, which include graphics engine system services together with the Window Management system services. During the initialization of WIN32K.SYS, the table is registered with the OS system service dispatcher. Thus, system service calls quickly get dispatched to the graphics engine.

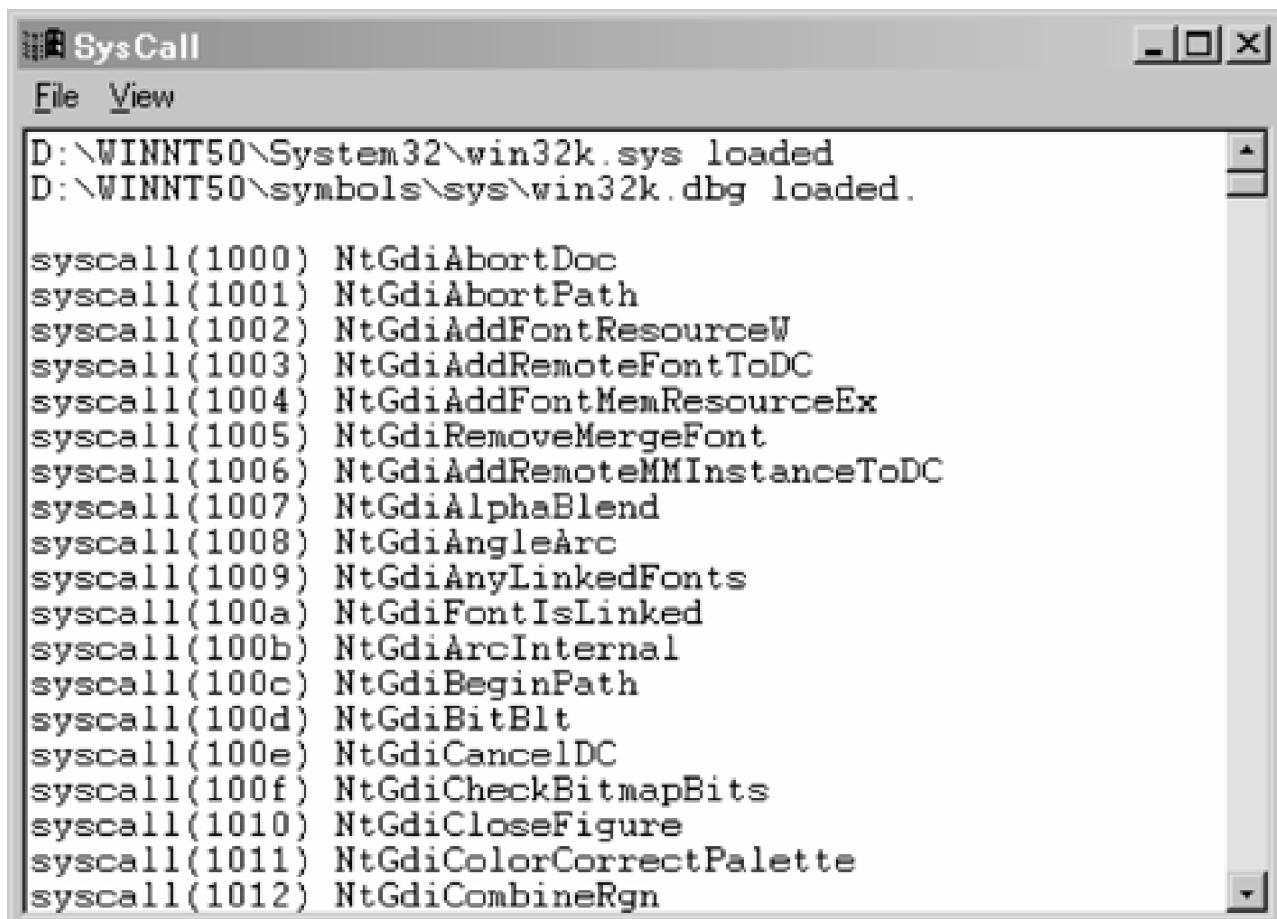
If you have Windows NT/2000 debug symbol files installed, you can examine symbols in WIN32K.SYS using the dumpbin tool. Dumpbin is a quite powerful tool, which can be used on Win32 PE files, object files, and debug

symbols. The following commands show how to list all the symbols in win32k.sys and search for the word "Service":

```
dumpbin symbols\sys\win32k.dbg /all > tmpfile  
grep Service tmpfile
```

You will find an interesting name, “_W32pServiceTable”. It is the starting address for a table of pointers to all WIN32K.SYS system service routines. Our SysCall program presented in [Section 2.2](#) can actually enumerate the contents of this table, translate them to symbolic names, and dump them into an edit window. So run the Sys Call program, choose under the “View” menu “System Call Tables” and then “Win32k .sys system call table,” and you will see the names of the 639 (Windows 2000) graphics engine and the window management system service routines. [Figure 2-8](#) shows a listing of Graphics Engine system services using the SysCall program.

Figure 2-8. Listing graphics engine system services.



For the SysCall program, listing the contents of the system service table is a much easier task than finding all the places making system service calls. It just needs to continuously read addresses, starting from W32pServiceTable from the win32k.sys image and translate them into symbolic names with the help of its debug symbol file.

Note

For the Hard Core Kernel fan club: Another major provider of system services for Windows NT/2000 is the Executive in NTOSKRNL.EXE, whose services are called from the Win32 subsystem DLL NTDLL.DLL. They provide the backbone support for Win32 base services, commonly known as

kernel services, mainly exported from KERNEL 32.DLL. The Executive uses services with identifiers lower than 0x1000, while services with higher identifiers are used by the graphics engine and the window manager. The SysCall program supports listing system service calls from NTDLL.DLL and the system service table in NTOSKRNL.EXE.

As we have already seen in the GDI system service call routines (routines making interrupt 0x2E calls), this list of graphics engine system service routines (routines servicing individual interrupt 0x2E calls) is not a surprise, except that the list from WIN 32K.SYS is nicely ordered according to the service index, and it's complete. Micro soft uses the same names for the corresponding GDI32.DLL and WIN32K.SYS routines. For example, NtGdiAbortDoc with the service index 0x1000, the first nonkernel system service, is in both tables.

Except for a few cases, the graphics engine services start with NtGdi; the window management services start with NtUser. In most cases, it's quite easy to trace a Graphics Engine system service routine to certain functions in GDI, DirectDraw, Direct3D, OpenGL, or user printer driver support. In other cases, the system service routine may be for internal use only. Here are a few examples:

- NtGdiAbortDoc is clearly implementing the GDI printing API function AbortDoc.
- NtGdiDdBlt can be traced to the DirectDraw IDirectDrawSurface interface.
- NtGdiDoBanding and NtGdiGetPerBandInfo are for printing banding.
- NtGdiCreateClientObj and NtGdiDeleteClientObj seem strange. After reading [Chapter 3](#), you will know them better.
- NtGdiGetServerMetafileBits and NtGdiGetSpoolMessage definitely help with the printer spooler.

Graphics Render Engine

Now let's jump to the foundation of the Windows NT/2000 graphics engine, the graphics render engine (GRE). After you see it, it's much easier to figure out how the whole graphics engine works.

With Windows NT/2000, Microsoft provides a full-feature render for all standard-format DIB formats, which include 1-, 4-, 8-, 16-, 24-, and 32-bits-per-pixel DIB. If an output device uses one of those DIB formats, the graphics engine does not need help from the device drivers to do any line drawings, area fillings, bitmaps, or text output. On the contrary, a graphics device driver can rely on the graphics engine to implement GDI drawing calls. Device drivers can do their own rendering if they choose, for performance like DirectDraw/Direct3D, or for special hardware design. This is significant in that it reduces the complexity of normal graphics device drivers significantly, increases the stability of the operating system, and reduces product time to market both for hardware vendors and for Microsoft.

GRE is used both by the graphics engine itself and by graphics device drivers; it forms the major part of functions exported by the graphics engine. They are well documented in Windows NT/2000 DDK under the title "GDI Functions for Graphics Drivers." The GRE API is quite different from the WIN32 GDI API. Here are some major concepts used by the GRE and the graphics engine in general:

- **GDI-Managed Bitmaps.** All standard DIB formats are supported, including 1-, 4-, 8-, 16-, 24-, and

32-bits-per-pixel uncompressed DIBs, and 4- or 8-bits-per-pixel run length encoding (RLE) compressed DIBs. A scan line in a DIB can be either top-down or bottom-down. The memory for DIB bits can be allocated either from kernel address space or from user process address space. The EngCreateBitmap function, exported from WIN32K.SYS, creates a GDI-managed bitmap and returns a handle to it.

- **Coordinate Space.** To improve drawing accuracy without using floating-point numbers, GRE may use fractional coordinates in 28.4 fixed-point format. This means the upper 28 bits are the signed integer part, while the lower 4 bits represent the fractional part. This is called FIX coordinates, used in lines and curves drawing. Other parts of API uses a 28-bit signed integer. All drawing calls are pre-transformed, so there is no more window and viewport, extend and world transformation in GRE. The largest DIB surface is 2^{27} by 2^{27} pixels, which is 1.42 km by 1.42 km in 2400 dpi, or 0.88 mile by 0.88 mile in 2400 dpi.
- **Surfaces.** A GDI-managed bitmap is only one type of surface fully manageable by GRE. A device driver can create device-managed surfaces in either DIB or non-DIB format using EngCreateDeviceSurface. It then controls the rendering on such surfaces. A device-dependent bitmap, as we know in Win32 API, is an example of a non-DIB device-managed surface.
- **Hooking and Punting.** By default, GRE does all rendering on GDI-managed DIB surfaces. But the device driver may decide to hook out a few drawing calls to be implemented by driver-supplied routines. The flHook flag in EngAssociateSurface determines what calls the driver wants to hook out. For example, the driver can provide a DrvBitBlt routine and turn on the HOOK_BITBLT flag in calling EngAssociateSurface. So DrvBitBlt gets the bitblting calls. But when DrvBitBlt gets called, it may determine that the operation is too complicated to handle. In this case, the driver can punt back to GDI by calling EngBitBlt.
- **Rendering Primitives.** The graphics engine merges and breaks down the rich set of Win32 GDI drawing calls into only a handful of rendering primitives, which are the drawing calls supported by the GRE. Here is a quick summary:
 - EngLineTo and EngStrokePath handle all line and curve drawing.
 - EngFillPath and EngPaint fill a closed area with a brush.
 - EngStroke and FillPath fill a closed area with a brush and strokes the perimeter with a pen.
 - EngBltBlt, EngPigBlt, EngStretchBlt, EngStretchBltROP, EngCopyBits, EngAlphaBlend, and EngTransparentBlt implement bitmap drawing API.
 - EngGradientFill does area gradient fill.
 - EngTextOut handles all text drawing calls.

Besides the simplified rendering primitives, GRE is using a brand-new set of data structures. We are not talking about GDI handles here; actually, GRE is living beneath or behind the GDI handle layer. Here are some of the C++-like class of objects used by GRE:

- A CLIPOBJ represents a clip region.
- A PATHOBJ is for both GDI path and all curve drawings converted to paths.

- A PALOBJ and XLATEOBJ are for color translation.
- A BRUSHOBJ is for both GDI brush and pen.
- A FONTOBJ is definitely a realized font.
- A STROBJ stores positions for glyphs in a text drawing call.
- A XFORMOBJ is for coordinate transformation, used for FONTOBJ.
- A SURFOBJ represents a drawing surface.

Although the GRE has a much simpler interface compared with the Win32 GDI API, implementing every little detail of it is a major task. Just imagine how many versions of bitbltting routines you need—at least 36 of them. Check the debug symbols in win32k.sys; you will find bSrcCopySRLE4D32, vSrcCopyS24D32, vSrcCopy S24D8, vSrcCopyS32D16, vSrcCopyS4D4Identify, BltLnkSrcCopyMsk32, etc. The first name, bSrcCopyRLE4D32, sounds like a routine to copy a 4-bits-per-pixel RLE compressed bitmap to a 32-bpp destination bitmap. The third name, vSrcCopy S24D8, hints at a routine to copy a 24-bpp bitmap to a 8-bpp destination bitmap. Remember, there are 16 binary raster operations and 256 ternary raster operations, not to mention quadnary raster operations that have two ternary raster operations in one.

Graphics Engine Data Structures

As with the rest of Win32 API, GDI uses GDI object handles to hide its implementation from programmers. This level of abstraction proves to be very useful in defining a somewhat unified Win32 API, having rather different implementations on Win32s, Windows 95/98, and Windows NT/2000.

As for any layer of abstraction, someone finally needs to manage these GDI handles, making them usable in both user mode system DLLs and kernel mode implementation. The GDI handle manager is the module that manages GDI handles.

A bigger question is how data structures are maintained in the GDI user mode DLL and kernel mode engine, or how Win32 GDI information regarding device con text, logical pen, logical brushes, fonts, regions, paths, etc., gets converted to or represented by Graphics Engine data structures. Understanding the underlining data structure of GDI helps the programmer to visualize how the GDI API is implemented, so that better programs can be written. [Chapter 3](#) of this book will explore this area and try to answer these questions.

Transform to Primitives

There is a visible size gap between the Win32 GDI API and the primitives supported by the GRE, which is also the DDI interface to the device drivers. The GDI API layer is responsible for transforming the Win32 GDI API calls to GRE primitives.

Here are some of the differences where transformation is needed:

- Coordinate system. The Win32 API provides a versatile coordinate system with view-port to window translation, mapping modes and world transformation, while the GRE/DDI uses device coordinates whose size is determined by the size of its drawing surface. Coordinates in the Win32 API calls need to be translated to device coordinates on the actual drawing surface.

- Elliptical curves. The GRE/DDI interface does not support elliptical curves like circle, arc, or ellipse. They need to be converted to Bezier curves. [Chapter 3](#) shows an example of how an ellipse can be represented using four Bezier curves. The graphics engine needs floating-point simulation in the conversion. The fixed-point coordinates used by the DDI interface make the final result more accurate.
- Curve to path. The DDI interface supports only straight lines and path, so all curve drawings need to be converted to path objects internally. Internally, the graphics engine has a rich set of path-object-manipulation functions.

If a surface is device managed, a device driver is required to support only a minimum of functions, which are DrvPaint, DrvCopyBits, DrvTextOut, and DrvStroke Path. For the rest of the functions, the graphics engine needs to break them down to these calls. Here are some of the things GDI does:

- For cosmetic line and curve output, the DrvStrokePath must support solid and styled cosmetic lines with solid brush and clipping. In implementing DrvStrokePath, the driver can call the PATHOBJ and CLIPOBJ service functions to break parameters down to single-pixel-wide lines and clipping rectangles. When the path or clipping is too complex, the driver can punt back to the graphics engine, which breaks down the call to single pixel lines with precomputed clipping. GDI can break styled lines or Bezier curves using straight-line approximation.
- For geometric lines, which can have width, join-style, and end-cap, if a device driver can't handle them, the graphics engine transforms drawing calls with these lines to simpler DrvFillPath or DrvPaint calls. In this case, a line is converted to an area fill.
- For filling areas, a driver must support DrvPaint. DrvPaint implementation can call the CLIPOBJ service functions to break a complex clipping region into a series of clipping rectangles.
- For bitmap block-transfer functions, the driver must support DrvCopyBits, which does block transfer to and from a standard DIB, or to device bitmap format, with an arbitrary clipping. DrvCopyBits does only plain copying without stretching, reflection, and raster operations. If the device driver supports only DrvCopyBits, the graphics engine needs to do its own simulation in its memory and send the result back using DrvCopyBits.

The relationship between the graphics engine and the device drivers is somewhat like a parent-child relationship. The graphics engine provides all the support for the device drivers; the device drivers can choose to do anything they want to outperform the graphics engine; yet they can fall back on the graphics engine whenever they have difficulty.

Font Drivers

Windows NT 4.0/2000 allows a special graphics device driver, which supplies font glyph bitmaps or outlines to the system. These drivers are called font drivers.

The Windows OS natively supports three types of fonts using different technology: raster fonts which are bitmap based, vector fonts which are based on straight lines, and TrueType fonts which are based on Bezier curve and sophisticated hinting mechanisms. As such, the graphics engine internally has three font drivers—for raster fonts, vector fonts and TrueType fonts.

Third-party vendors can provide their own font drivers. For example, a printer driver may include a font driver to supply the graphics system with information on printer device fonts. Another example is the ATM (Adobe Type

Manager) font driver (ATMFD.DLL) which comes with Windows 2000. The ATM font driver supports ATM fonts, which are based on Adobe's PostScript font technology.

[< BACK](#) [NEXT >](#)

2.6 DISPLAY DRIVERS

A display driver in Windows NT/2000 belongs to the graphics device-driver category. A graphics device driver is normally a kernel mode DLL, which is loaded into kernel address space. It's responsible for the final implementation of drawing calls from a user application on a hardware device. For Windows NT/2000, a display driver is always a kernel mode DLL. Only printer drivers are allowed to be user mode DLLs under Windows 2000.

The interface protocol between the GDI graphics engine and graphic device drivers is commonly known as Device Driver Interface (DDI). Why is such a misleading general term used for graphics drivers only? Perhaps graphics drivers were the only significant vendor-supplied drivers in the old days of Windows development.

Video Port Driver and Video Miniport Driver

Each display driver is paired with a kernel mode video miniport driver. Whenever you see a "mini" prefix for a driver, you know there is a nonmini driver which hosts the minidriver. In this case, a video miniport driver is controlled by the video port driver.

The video port driver and video miniport driver handle interaction with the video adapter card, including hardware initialization and detection, memory mapping, video register access, etc. The video miniport driver can map video registers to display the drivers' memory space to allow them to be accessible using normal memory access.

For Windows 2000, which is designed to support DirectX in the video display driver level, the video miniport driver is also responsible for supporting DirectX. For example, one of the main advantages of DirectDraw over GDI is that the user application has direct access to the screen display buffer. This is achieved by the video miniport driver, which maps the display buffer to a virtual address region accessible from user applications.

A display driver interfaces with its video miniport driver through the graphics engine's EngDeviceControl routines, which go through the NT kernel I/O manager to the video port driver and then to the video miniport driver.

Display Driver Function Areas

Although the video miniport driver provides direct access to video hardware, normally it's controlled by a display driver. A display driver has four main areas of responsibility:

- Enable and disable the graphics hardware, including mapping video memory, banking,

off-screen heap, hardware mouse pointer, hardware palette, brush caching, and DirectDraw/Direct3D/OpenGL hardware support, if available. A display driver normally passes the requests to the video port driver.

- Supply GDI engine with hardware and driver capabilities through several DDI data structures.
- Create and enable drawing surface, which can either be a GDI-managed DIB surface, a device-managed DIB surface, or a device-managed non-DIB surface.
- Implement basic or all drawing operations on a drawing surface through surface hooks, or create a GDI-managed surface and let the graphics engine draw directly into it.

Display Driver Initialization

A graphics device driver is normally a kernel mode DLL that can only import functions from the graphics engine (WIN32K.SYS). Its main entry point, which has the same position as_DIIIMainCRTStartup, is normally named DrvEnableDriver. Once a driver is loaded, its DrvEnableDriver is called by the GDI engine to create the first device context for the device. The driver does some simple version checking and returns to the GDI engine a function table that lists all the DDI functions it supports in a DRVENABLE DATA structure. Every DDI function a Windows NT/2000 graphics driver supports has a predefined index. So DrvEnableDriver needs only to return the list of supported DDI functions, each paired with a DDI function index. Windows 2000 defines a total of 89 DDI functions. For example, the corresponding closing call for DrvEnable Driver is DrvDisableDriver, which has an index of 8.

After getting the function table, the graphics engine normally will call Drv Enable PDEV to get the driver to create an instance of a physical device, which also returns some important information about the graphics hardware and driver. With Drv Enable PDEV call, the graphics engine passes the driver a copy of the DEVMODEW structure that specifies the characteristics of the graphics device. For display cards, DEV MODEW specifies display frequency, resolution, and display modes like gray scale or interlaced display. For printers, DEVMODEW is even more important with paper type, paper size, orientation, printing quality, copy count, and paper bin requirements.

DrvEnablePDEV allocates an instance of a driver-defined PDEV(physical device) structure where private driver data can be stored. It returns a handle to its PDEV structure; that is the way the GDI engine refers to this instance of physical device during subsequent calls. It fills-in a GDIINFO structure that informs the GDI engine about the device's resolution, physical size, color format, DAC bits, aspect ratio, palette size, primary color or plane order, halftoning pattern size and format, refresh frequency, etc. Another data structure for feeding information back is the DEVINFO structure, which describes the driver's graphics capabilities, default fonts, number of device fonts, and dither format. The graphics capability flags can tell the graphics engine whether the device is capable of handling Bezier curves, geometric widening, alternate and winding fills, EMF printing support, antialiased text, mirror driver, hardware font rasterizer, JPEG, PNG support, loadable hardware gamma ramp, and hardware alpha cursor support, etc. After calling DrvEnablePDEV, the

graphics engine does its internal initialization for the physical device and finally calls DrvCompletePDEV to signal that a physical device is ready to use. The closing call for a PDEV is DrvDisablePDEV, which normally frees the memory allocated for the physical device.

Before the GDI engine can draw to a display, the graphics engine calls Drv Enable Surface to let the driver create a drawing surface. If the display card uses the standard DIB format for its frame buffer, it can create an engine-managed surface using EngCreateBitmap. If not, the display driver does its own allocation for the storage needed for the surface and calls EngCreateDeviceSurface to inform the graphics engine of the size and compatible format of the surface. In either case, the device driver then calls EngAssociateSurface to specify the DDI surface drawing calls the driver wants to hook out, using functions exposed through the function table returned in the Drv EnableDriver call. After the graphics engine is done with a surface, it calls DrvDisable Surface to let the driver release the resources allocated for it.

Surface Drawing Calls, Hooking, and Punting

After a surface is successfully created, the graphics engine sends drawing calls to the device driver according to its capabilities bits and surface hooking flags. [Table 2-1](#) lists all of the hookable DDI surface rendering calls.

The table shows clearly that for every DDI rendering call, the graphics rendering engine (GRE) has one routine with the exact parameters which performs the function on a standard DIB surface. So there are several ways a graphics driver can implement the DDI rendering calls:

- If the surface uses a standard DIB surface, the driver can let GRE do all the rendering calls by not hooking the DDI drawing calls. For example, the frame buffer video display card sample that comes with Windows 2000 DDK does not hook out any function (ddk\src\video\displays\framebuf).

Table 2-1. Windows 2000 DDI Rendering Hooks

Index	Graphics Engine Function	Driver Function
HOOK_BITBLT	EngBlitBlt	DrvBitBlt
HOOK_STRETCHBLT	EngStretchBlt	DrvStretchBlt
HOOK_PLGBLT	EngPlgBlt	DrvPlgBlt
HOOK_TEXTOUT	EngTextOut	DrvTextOut
HOOK_PAINT	EngPaint	DrvPaint
HOOK_STROKEPATH	EngStrokePath	DrvStrokePath
HOOK_FILLPATH	EngFillPath	DrvFillPath
HOOK_STROKEANDFILLPATH	EngStrokeAndFillPath	DrvStrokeAndFillPath
HOOK_LINETO	EngLineTo	DrvLineTo
HOOK_COPYBITS	EngCopyBits	DrvCopyBits
HOOK_MOVEPANNING	EngMovePanning	DrvMovePanning
HOOK_SYNCHRONIZE	EngSynchronize	DrvSynchronize
HOOK_STRETCHBLTROP	EngStretchBltROP	DrvStretchBltROP
HOOK_SYNCHRONIZEACCESS	EngSynchronizeAccess	DrvSynchronizeAccess
HOOK_TRANSPARENTBLT	EngTransparentBlt	DrvTransparentBlt
HOOK_ALPHABLEND	EngAlphaBlend	DrvAlphaBlend
HOOK_GRADIENTFILL	EngGradientFill	DrvGradientFill

- If the surface is device managed, the driver can choose to hook out a few must-implement primitive hooks and let the graphics engine break out the rest into primitive calls. The driver can also hook all drawing calls to use hardware acceleration or optimized software implementation. For example, the Windows 2000 DDK s3virge sample display driver provides optimized assembly implementation for certain bitblting between the screen buffer and the off-screen DIBs. Therefore, it definitely needs to hook the DrvBlitBlt calls.
- Within a driver-provided hook function, the driver can call the graphics engine service routines to break down a complicated clipping region into a more manageable series of rectangles. The driver can also punt back to GRE when it's overwhelmed. The same s3virge driver mentioned above can punt back to the graphics engine for bitblting between two off-screen DIBs.

Other Driver Features

Besides initialization/termination and DDI drawing hooks, a display driver also needs to expose other entry points to the graphics engine for other tasks like:

- Device bitmap management: DrvEnableDeviceBitmap and DrvDisable Device Bitmap.

- Palette management: DrvSetPalette.
- Brush realization, color dithering, and ICM (image color management) support: DrvRealizeBrush, DrvDitherColor, DrvIcmCreateColorTransform, DrvIcmCheckBitmapsBits, etc.
- GDI escape function: DrvEscape, DrvDrawEscape (e.g., for passthrough Postscript data).
- Mouse management: DrvSetPointerShape, DrvMovePointer.
- Font query and font driver support: DrvQueryFont, DrvQueryFontTree, DrvQueryFontData, DrvQueryFontFile, DrvQueryTrueTypeFontTable, etc.
- Printing: DrvQuerySpoolType, DrvStartDoc, DrvEndDoc, DrvStartPage, DrvEndPage, etc. More on printing in [Section 3.7](#).
- OpenGL support: DrvSetPixelFormat, DrvSwapBuffers, etc.
- DirectDraw/Direct3D support: DrvEnableDirectDraw, DrvGet Direct Draw Info, and DrvDisableDirectDraw.

Most of those functions are optional. A graphics device driver needs to support them only when its hardware does.

Display Driver Support for DirectDraw/Direct3D

If a display driver supports DirectDraw/Direct3D, it must expose the DrvGetDirectDrawInfo entry point, through which the interaction between the graphics engine and DirectDraw hardware begins.

When a DirectDraw/Direct3D application creates an instance of a DirectDraw object, the graphics engine calls DrvGetDirectDrawInfo first to query DirectDraw support in a display driver.

DrvGetDirectDrawInfo returns DirectDraw/Direct3D hardware support information to GDI, including hardware capabilities, chunks of display memory, and a list of media formats supported. Hardware capabilities are encoded in a DD_HALINFO structure, which details hardware support for hardware bitbltting, stretching, alpha channel, clipping, color keying, overlay, palette, Direct3D support, and lots of other stuff. The display memory format information is returned in an array of VIDEOMEMORYINFO structure. Each media format is encoded with a 32-bit value called a FOURCC, which serves as a media type tag used in multimedia API. DirectDraw uses FOURCC to describe pixel formats supported by video cards and different compressed-texture pixel formats.

The graphics engine then calls DrvEnableDirectDraw to inform the driver to enable hardware for DirectDraw usage. DrvEnableDirectDraw fills three structures with callback routines for DirectDraw, DirectDraw surface, and DirectDraw palette interfaces. This is similar to the display driver's main entry point, DrvEnableDriver, that returns an indexed list of callback routines for DDI implementation.

Each of the three data structures returned by DrvEnableDirectDraw roughly corresponds to one or part of a DirectDraw interface, as we have seen from the DirectDraw API. It has a flag field that indicates the callback routines supported, and pointers to all callback routines for that interface. For example, DD_CALLBACKS is for DirectDraw implementation, which defines nine callback routines. If its dwFlags field has a DDHAL_CB32_CREATESURFACE flag, its CreateSurface field should point to a callback routine normally named as DdCreateSurface. The IDirectDraw interface has more than nine methods. Some of the methods are implemented internally in the DirectDraw Win32 client DLL; the rest of the corresponding driver level routines are returned in other structures like DD_NTCALLBACKS.

A display driver's support for Direct3D is first signaled by a few flags in the DD_HALINFO structure returned by DrvGetDirectDrawInfo. The DDCAPS_3D flag in the ddCaps.dwCaps field indicates that the driver's hardware has 3D acceleration. The flags in the ddCaps.ddsCaps field, for example, DDSCAPS_3DDEVICE, DDS_CAPS_TEXTURE, DDSCAPS_ZBUFFER, describe the video memory surface's 3D capabilities. The DD_HALINFO also has a pointer to a D3DNTHAL_CALLBACKS structure, which lists Direct3D DDI callback routines.

[Table 2-2](#) lists some of the structures used to describe the display driver's callback routines for DirectDraw/Direct3D support.

Now that we have seen both the GDI DDI interface and the DirectDraw/Direct3D DDI interface, it's easy to see that they are totally different in design. Here are some of the differences:

Table 2-2. DirectDraw/Direct3D DDI Callback Routines (Partial)

Structure	Callback Functions
DD_CALLBACKS	DdDestroyDriver, DdCreateSurface, DdSetColorKey, DdSetMode, DdWaitForVerticalBlank, DdCanCreateSurface, DdCreatePalette, DdGetScanLine, DdMapMemory
DD_SURFACECALLBACKS	DdDestroySurface, DdFlip, DdSetClipList, DdLock, DdUnlock, DdBlt, DdSetColorKey, DdAddAttachedSurface, DdGetBltStatus, DdGetFlipStatus, DdUpdate Overlay, DdSetOverlayPositions, DdSetPalette
DD_PALETTECALLBACKS	DdDestroyPalette, DdSetEntries
DD_NTCALLBACKS	DdFreeDriverMemory, DdSetExclusiveMode, DdFlipToGDISurface
DD_COLORCONTROLCALLBACKS	DdColorControl
DD_MISCCELLANEOUSCALLBACKS	DdGetAvailDriverMemory
D3DNTHAL_CALLBACKS	D3dContextCreate, D3dContextDestroy, D3dContextDestroyAll, D3dSceneCapture, D3dTextureCreate, D3dTextureDestroy, D3dTextureSwap, D3dTexture GetSurf,

D3DNTHAL_CALLBACKS3

D3dClear2, D3dValidateTextureStageState,
D3dDrawPrimitives2

- The GDI DDI interface is at a much more primitive level than the DirectDraw/Direct3D DDI interface. So the graphics engine needs to perform lots of pre-processing before calling the GDI DDI interface, while the DirectDraw/Direct3D path from the Win32 API to the display driver is much more direct and less involved.
- For GDI DDI interface support, the display driver can always get help from the graphics engine by punting back to it. For the DirectDraw/Direct3D DDI interface, software simulation is in user mode Win32 client DLL, so the display driver has nothing to fall back on in kernel mode driver level.
- The GDI DDI interface uses a simple and extensible way to query the driver's callback routines, while the DirectDraw/Direct3D DDI has an array of data structures to describe the callback routines.

We have seen how the display driver is initialized to support GDI, DirectDraw, and Direct3D, and we even got to know a little bit about the driver's main entry points and callback routines. We will come back to see how those callback routines are managed and used by the graphics engine when we explore the Windows graphics system's internal data structure in [Chapter 3](#), and spying on GDI/DirectDraw in [Chapter 4](#).

[< BACK](#) [NEXT >](#)

2.7 PRINTER DRIVERS

A display driver, as described in [Section 2.6](#), is just one class of graphics device drivers the OS supports. Another major class of graphics device drivers operates hard-copy devices like printers, plotters, fax machines, or other nontraditional devices. Graphics device drivers for hard-copy devices have the same structure, so whatever we discuss about a printer driver applies to a fax driver in principle.

The main difference between hard-copy devices is the printer language they use, which can be classified into three classes:

- Text-only devices: the traditional line printers that print only plain text. They are not so common in the Windows environment, which encourages a what-you-see-is-what-you-get graphics user interface. The device driver needs to send a text stream to them with limited formatting for line breaks and page feeds.
- Raster-based devices: include dot-matrix printers, fax machines, and most DeskJet or ink-based printers. The device driver needs to convert DDI drawing commands into a raster image and encode it in printer languages like PCL3, ESC/2, etc.
- Vector-based devices: include laser printers, plotters, Postscript printers, and high-end DeskJet printers. Although some of those devices may be rasterbased during actual printing, they all accept vector-based graphics input and convert them within the printer to raster images. The device driver for vector-based devices normally converts DDI drawing commands into commands in printer languages—for example, PCL5, PCL6, HPGL, HPGL/2 or PostScript. A vector-based device normally supports images at the same time.

A full Windows NT/2000 printer driver has several components; the first two are its major must-have components:

- A printer graphics DLL that accepts DDI drawing commands as a display driver does, transforms them into printer language, and sends the data to the printer spooler.
- A printer interface DLL that provides the user interface to the device's configurable parameters and to the spooler for printer installation, configuration, and event reporting.
- An optional printer processor that helps the spooler process to despool print jobs.
- An optional language monitor that provides bidirectional feedback to the spooler and user.
- An optional port-monitor that sends printer-ready data to hardware port drivers.

Microsoft Printer Driver Frameworks

To help vendors provide printing solutions to customers, Microsoft provides several standard drivers which printer manufacturers can use to build their own drivers by building plug-in modules instead of full-blown drivers.

- The Universal Printer Driver (Unidrv) is Microsoft's solution for non-Postscript printers—for example, dot

matrix, DeskJet, and LaserJet printers. A vendor only needs to provide a minidriver for UniDrv, which at minimum is a text-based GPD file which details printer features, options, conditional constraints, and printer commands. The UniDrv architecture allows for plug-in modules. A render plug-in can customize drawing command rendering, halftoning, and printer ready-data generation. A user interface plug-in module customizes printer property sheets, the DEVMODE structure, and print event handling.

- The PostScript Printer Driver (Pscript) is Microsoft's solution for Postscript printers. A Postscript minidriver has a text-based PPD file describing the printer's characteristics, a binary NTF file describing the printer's device fonts, a render plug-in, and a user interface plug-in.
- The Microsoft Plotter Driver is Microsoft's standard for supporting Hewlett-Packard Graphics Language (HPGL/2) compatible plotters. A plotter mini driver is only a binary PCD file describing the plotter's characteristics. No plug-in is designed for it because the printer language HPGL/2 is quite fixed. The Windows 2000 DDK provides full source code for the Microsoft Plotter Driver.

The driver frameworks provided by Microsoft use a very interesting data-driven design. They literally support thousands of printers on the market with their differences often captured by a small difference in their data files. A UniDrv GPD file specifies in every detail a printer's orientation, input bin, paper size, rendering resolution, print mode, media type, color mode, print quality setting, halftoning, configuration constraints, printer configuration commands, and printing commands. When a vendor releases a new printer, the only change to the driver may be an updated GPD file with a new higher-resolution print mode. The language in which the GPD file is written is quite an expressive one that supports simple data types like integer, pair, string, and list, and even variables and switch statements. But you may still wonder why Microsoft does not just use standard simple languages like Lisp or Prolog, which are more powerful and easier to parse.

Printer Driver Graphics DLL

A graphics DLL for a printer driver is very similar to the display driver we talked about in [Section 2.6](#). The main difference is that a printer driver needs to support some extra entry points for document and page handling, but not those entry points a display driver has to support for mouse cursor, DirectDraw, and Direct3D.

A printer driver certainly needs to support the basic entry points that are responsible for the initialization for the graphics device drivers, namely DrvEnableDriver, DrvEnablePDEV, DrvCompletePDEV, DrvDisablePDEV, DrvEnableSurface, DrvDisableSurface, and finally DrvDisableDriver. A printer driver also needs to handle the separation of a printing document into individual pages and printer-specific queries. The DDI surface hooking routine for a printer driver is the same as that for a display driver. [Table 2-3](#) lists extra entry points a printer driver needs to or could support.

One interesting thing about Windows 2000 printer drivers is that they could be user mode DLL instead of just kernel mode DLL. Microsoft makes special efforts to encourage printer drivers to get out of the kernel address space to the user address space. But remember, a printer driver could import functions from the graphics engine WIN32K.SYS, which are not available at the GDI level. To solve this problem, Windows 2000 GDI exports a subset of the graphic rendering engine functions so that a user mode printer driver could use them directly. The graphics engine makes a special effect to thunk calls to a printer driver from kernel mode to user mode, and then the GDI thunks engine calls from the driver from user mode back to kernel mode. The benefits of having user mode printer drivers are the lower development cost, more versatile Win32 API to use, and, most importantly, less turmoil in the OS kernel. If a printer driver is a user mode printer driver, DrvEnableDriver, DrvDisableDriver, and DrvQueryDriverInfo need to be exported functions from the driver DLL, and DrvQueryDriverInfo needs to signal as such when queried for DRVQUERY_USERMODE. All standard Microsoft Windows 2000 printer drivers are user mode drivers.

Table 2-3. Printer Driver Specific Entry Points

Entry Points	Function
DrvQueryDriverInfo (optional)	Driver-specific queries. Currently used for user mode driver query.
DrvQueryDeviceSupport (optional)	Device-specific queries. Currently used for JPEG, PNG, support queries.
DrvStartDoc	Informs driver GDI is ready to send document.
DrvEndDoc	Informs driver GDI finished sending document.
DrvStartPage	Informs driver GDI is ready to send drawing commands for a new page.
DrvSendPage	Informs driver GDI finished sending commands for a page; driver can send rendered data to spooler.
DrvStartBanding	Queries driver where the banding should start on a page.
DrvQueryPerBandInfo (optional)	Queries band information through a PERBANDINFO structure, which specifies the size and resolution of a band.
DrvNextBand (optional)	Informs driver GDI finished sending commands for a band; driver can send rendered data to spooler.

The main entry for a printer driver is still DrvEnableDriver. When an application calls CreateDC to create a device context for a printer, the graphics engine checks if the printer driver is loaded. If not, the driver is loaded and its DrvEnableDriver is called.

A printer driver's DrvEnablePDEV is called by the graphics engine when an application calls CreateDC using the Win32 API on a printer device. It may be more complex than a display driver because it needs to react to lots of printing settings passed to it through the DEVMODE structure. For example, the driver's need to adjust the rendering resolution depends on print quality modes, switching the paper dimension for landscape mode, calculating the device dimension based on paper size, or reporting paper margins. The DEVMODE structure may have other printing settings like duplex, collate, copy count, *n*-up, or vendor-specific settings that are implemented by other parts of the whole printing system. For example, the Windows 2000 default system print processor implements duplex, collate, copy count and *n*-up printing so that the core printer driver needs to worry about rendering only a single page.

For raster printers, the printer driver could create a graphics-engine-managed surface during DrvEnableSurface call and ask the graphics engine to do all or most of the rendering. The trouble is that the printer uses a much higher resolution which would require too much memory resource to render the whole at once. For example, a 300 × 300 dpi letter-size page is 300 × 300 × 11.5 × 8 pixels, or close to 8 mega pixels. For a black/white printer at 300 dpi, a full-page bitmap would be close to 1 mega byte; for a 600-dpi color printer using 24-bit rendering, it would be close to 96 megabytes. To reduce this possible huge memory consumption to a manageable level, the graphics engine supports dividing a page into a series of horizontal bands. If you divide a letter-size page into equal half-inch bands, the 96 megabytes figure is now only 4.17 megabytes. If banding is used, the driver needs to call EngMarkBandingSurface to tell the graphics engine that the driver surface needs banding.

A vector-based printer does not need to render the whole page into a bitmap; instead it normally translates the DDI drawing calls into printer commands one by one. Normally, a device-managed surface will be created, and the driver will hook the DDI drawing commands. Banding is normally not needed for vector-based printers.

Once the driver is loaded and a physical device structure and surface are created, GDI with the help of the graphics engine is ready to send the whole document to a printer driver. The process for rendering a document is roughly as follows:

DrvStartDoc(DocName, JobId)

```
for (int i=0; i<nPageNo; i++)
{
    DrvStartPage();
    DrvStartBanding();
    for (BOOL bMoreBands=TRUE; bMoreBands; )
    {
        DrvQueryPerBandInfo();
        for (c=0; c<nCommands; c++)
            if ( touches(Command[c], band) )
                Draw(command[c]);
        bMoreBand = DrvNextBand();
    }
    DrvSendPage();
}
```

DrvEndDoc();

The whole printer job is enclosed by the DrvBeginDoc and DrvEndDoc call, while each page is enclosed by the DrvStartPage and DrvEndPage call. For each page, DrvStartBanding is called first to signal the driver that banding is starting. The graphics engine then calls DrvQueryPerBandInfo to query for geometric information about the band to render, and it scans through its spooled GDI command file to send all the commands touching the command band. After one band is finished, GDI moves to the next band until the whole page is done.

The DDI drawing commands are either rendered directly to the GDI-managed surface by GRE or are sent to driver-provided hook routines. For a vector device, the transformed commands can be sent to the spooler for each drawing call. For a raster device, the rendered bitmap needs to be halftoned to meet the color depth of a printer, divided into several planes (for example, CYMK), compressed, and encoded into printer language. Sending data to the spooler uses the driver handle parameter passed to the driver in the DrvEnablePDEV call. EngWriteSpooler uses this handle to communicate with the spooler.

A printer driver could also support DrvEscape and DrvDrawEscape for official or back-door communication from application to driver. For example, the Postscript driver allows the injection of raw Postscript data into the data stream through Drv Draw Escape. It uses DrvEscape to query for such support.

During the handling of a printer job, the printer driver could call EngCheckAbort to see if the user has cancelled the job. Driver's DrvDisableSurface call should free resources allocated for a surface. When an application or GDI calls DeleteDC, the driver's DrvDisablePDEV is called to let the driver free resources allocated for a physical device. If an application calls ResetDC during printing, GDI calls DrvEnablePDEV to create a new instance. It then calls DrvResetPDEV to let the driver copy information from old to new, DrvDisableSurface on the old device, and then start a new rendering sequence on the new device. DrvDisableDriver will finally be called before unloading a printer graphics DLL.

An HTML Printer Driver

Have you ever thought about how can you convert a page within a document into a bitmap? Capturing the screen

seems to be the only easy way out. But what if you want to convert more than could be displayed on one screen? You don't want to capture multiple times and then put them together. What if you want to convert a 100-page document into 100 bitmaps? Getting a bitmap printer driver is the answer. Or how about writing a bitmap driver yourself?

Printing normally involves multiple-page documents, so having a printer driver generate one bitmap at a time is not really handy. If you can generate one bitmap, you can easily generate an HTML document that links all the rendered bitmaps for a document to have a nice appearance.

An HTML printer driver is one that uses HTML as its output language. Although there may not be a physical printer that accepts an HTML document as direct input, you can easily do a print to file and use your web browser to read your document. HTML is not really a vector-based language, so we can't convert DDI drawing commands one-by-one to HTML commands; instead, we can render a page into a bitmap, which will then be linked by an HTML document.

We have to be modest in our goals to make this project feasible. So we've decided to support only a single paper size: 11.5 by 8-inch letter-size paper; single resolution: 96 dpi; simple color depth: 24-bit DIB. This makes the rendering surface for the whole page 2.46 megabytes in size, manageable without banding. Implementing the DDI commands ourselves is quite troublesome; instead, a GDI-managed surface is created on which the graphics engine can do all the dirty work.

It's not really worth the trouble to implement a printer driver just to get a few bitmaps; the HTML printer driver could be a vital learning tool in understanding how GDI is implemented. So our HTML driver hooks all the common DDI rendering calls just to have an option to check all the parameters. The driver rendering routines still punt back to the graphics engine for the actual rendering.

Generating an HTML page with links to bitmaps means more than the single data stream written to the spooler which is not supported by the DDI interface. Linked-in bitmaps need to be written to separate files. Luckily, WIN32K.SYS provides simple memory-mapped file operations through EngMapFile and EngUnmapFile.

The actual implementation of our HTML printer driver is divided into two pieces: a KDevice that encapsulates the physical device data block created by Drv EnablePDEV and operations on the device, and the DDI interface part.

Here is the header file for the KDevice class:

```
struct PAIR;

class KDevice
{
    int    nNesting; // document, page
    int    nPages;   // no of pages printed

    void   Write(const char * pStr);
    void   WriteW(const WCHAR * pwStr);
    void   WriteHex(unsigned val);
    void   Writeln(const char * pStr = NULL);

    void   Write(DWORD index, const PAIR * pTable);
```

```
char * CopyBlock(char * pDest, void * pData, int size);
void CopySurface(char * pDest, const SURFOBJ * pso);

void LogCall(int index, const void * para, int parano);

public:
    int width; // width in inch * 10
    int height; // height in inch * 10
    HPALETTE hPalette; // GDI need a palette even for 24 bpp
    HSURF hSurface; // Device managed standard surface
    HDEV hDevice; // GDI device handle
    HANDLE hSpooler; // Spooler handle

    int nImage;

    void Create(void)
    {
        nNesting = 0;
        nPages = 0;
        nImage = 0;
    }

    void DumpSurface(const SURFOBJ * psoBM);

    BOOL CallEngine(int index, const void * para, int parano);
    BOOL StartDoc(LPCWSTR pszDocName, const void * firstpara,
                  int parano);
    BOOL EndDoc(const void * firstpara, int parano);
    BOOL StartPage(const void * firstpara, int parano);
    BOOL SendPage(const void * firstpara, int parano);
};
```

The KDevice class has fields to hold paper dimension, palette, surface, device, and spooler handles. Other data fields are for internal bookkeeping. Nesting makes sure that DrvStartDoc, DrvEndDoc, DrvStartPage, and DrvSendPage are called in proper sequence; nPages is the number of pages printed; nImage is the sequence number for HTML links to images. The KDevice class does not have a true constructor, destructor, or virtual functions. We don't want to involve true C++ runtime support in a kernel mode DLL. The Create method does partial initialization. A bunch of Pascal-like writing methods dump HTML data to the data stream sent to the spooler. The DumpSurface method writes the DIB contents of a surface into a separate bitmap file.

The KDevice::LogCall routines may be interesting. It's a simple logger which dumps the driver entry point name and parameter list. The call must supply the entry point index, the address of the first parameter on the stack, and the number of parameters, noting that the Pascal calling convention is used for the parameters. The routine does a simple hex dump of the parameters.

```
void KDevice::LogCall(int index, const void * firstpara,
                      int parano)
{
```

```
Write("<li>");  
Write(index, Pair_DDIFunction); // table lookup for func name  
Write("");  
  
const unsigned * pDWORD = (const unsigned *) firstpara;  
  
for (int i=0; i<parano; i++)  
{  
    WriteHex(pDWORD[i]);  
  
    if ( i==(parano-1) )  
        Write(" , ");  
    }  
    Writeln("</li>");  
}
```

A related routine, KDevice::CallEngine, does simple validation for KDevice's pointer, and calls LogCall if it's OK. CallEngine is used by all drivers' DDI rendering routines before punting back to the graphics engine. So CallEngine's implementation could also selectively disable certain DDI calls to let the graphics engine decompose the simpler calls.

File HTMLDrv.cpp implements the DDI interface for a driver. It starts with a table of the supported entry points:

```
const DRVFN Drv_Funcs[] =  
{  
    INDEX_DrvEnablePDEV,      (PFN) DrvEnablePDEV,  
    INDEX_DrvCompletePDEV,    (PFN) DrvCompletePDEV,  
    INDEX_DrvResetPDEV,       (PFN) DrvResetPDEV,  
    INDEX_DrvDisablePDEV,     (PFN) DrvDisablePDEV,  
    INDEX_DrvEnableSurface,   (PFN) DrvEnableSurface,  
    INDEX_DrvDisableSurface,  (PFN) DrvDisableSurface,  
  
    INDEX_DrvStartDoc,        (PFN) DrvStartDoc,  
    INDEX_DrvEndDoc,          (PFN) DrvEndDoc,  
    INDEX_DrvStartPage,       (PFN) DrvStartPage,  
    INDEX_DrvSendPage,        (PFN) DrvSendPage,  
  
    INDEX_DrvStrokePath,     (PFN) DrvStrokePath,  
    INDEX_DrvFillPath,        (PFN) DrvFillPath,  
    INDEX_DrvStrokeAndFillPath, (PFN) DrvStrokeAndFillPath,  
    INDEX_DrvLineTo,          (PFN) DrvLineTo,  
    INDEX_DrvPaint,           (PFN) DrvPaint,  
    INDEX_DrvBitBlt,          (PFN) DrvBitBlt,  
    INDEX_DrvCopyBits,        (PFN) DrvCopyBits,  
    INDEX_DrvStretchBlt,      (PFN) DrvStretchBlt,  
    INDEX_DrvTextOut,         (PFN) DrvTextOut  
};
```

DrvEnableDriver reports the function table to the graphics engine, after some simple checking:

```
BOOL APIENTRY DrvEnableDriver(ULONG iEngineVersion, U
                                LONG cj, DRVENABLEDATA *pded)
{
    // Validate parameters.
    if (iEngineVersion < DDI_DRIVER_VERSION)
    {
        EngSetLastError(ERROR_BAD_DRIVER_LEVEL);
        return FALSE;
    }

    if (cj < sizeof(DRVENABLEDATA))
    {
        EngSetLastError(ERROR_INVALID_PARAMETER);
        return FALSE;
    }

    pded->iDriverVersion = DDI_DRIVER_VERSION;
    pded->c      = sizeof(Drv_Funcs) /
                    sizeof(Drv_Funcs[0]);
    pded->pdrvfn = (DRVFN *) Drv_Funcs;

    return TRUE;
}
```

Here is part of the DrvEnablePDEV routine that creates a new instance of the KDevice class. DrvEnablePDEV fills in the driver's capabilities in a GDIINFO and a DEVINFO structure, depending on the settings in the incoming DEVMODEW structure. The code here checks for paper orientation.

```
DHPDEV APIENTRY DrvEnablePDEV(DEVMODEW *pdm,
                                LPWSTR pwszLogAddress,
                                ULONG cPat,
                                HSURF *phsurfPatterns,
                                ULONG cjCaps,
                                ULONG *pdevcaps,
                                ULONG cjDevInfo,
                                DEVINFO *pdi,
                                HDEV hdev,
                                PWSTR pwszDeviceName,
                                HANDLE hDriver)
{
    if ( (cjCaps<sizeof(GDIINFO)) || (cjDevInfo<sizeof(DEVINFO)) )
    {
        EngSetLastError(ERROR_INVALID_PARAMETER);
```

```
    return FALSE;
}

KDevice * pDevice;
// Allocate physical device object. Tag = HTMD
pDevice = (KDevice *) EngAllocMem (FL_ZERO_MEMORY,
                                   sizeof(KDevice), 'DMTH');
if (pDevice == NULL)
{
    EngSetLastError(ERROR_OUTOFMEMORY);
    return NULL;
}

pDevice->Create();

pDevice->hSpooler = hDriver;
pDevice->hPalette = EngCreatePalette(PAL_BGR,0,0,0,0,0);

if (pdm == NULL || pdm->dmOrientation == DMORIENT_PORTRAIT)
{
    pDevice->width = PaperWidth;
    pDevice->height = PaperHeight;
}
else
{
    pDevice->width = PaperHeight;
    pDevice->height = PaperWidth;
}

// GDIINFO initialization omitted
// DEVINFO initialization omitted
pdi->hpalDefault = pDevice->hPalette;

return (DHPDEV) pDevice;
}
```

DrvEnableSurface creates a full-page GDI-managed 24-bit DIB surface, and informs the graphics engine that the driver wants to hook out a few DDI calls. Note that the paper dimension is stored in tenths of an inch to avoid floating-point operations in the kernel. The surface is cleared to a white page in DrvStartPage instead of when it's created.

```
HSURF APIENTRY DrvEnableSurface(DHPDEV dhpdev)
{
    KDevice * pDevice = (KDevice *) dhpdev;

    SIZEL sizl = { pDevice->width * Dpi / 10,
                  pDevice->height * Dpi / 10 };
    pDevice->hSurface = (HSURF)
```

```
EngCreateBitmap(sizl, sizl.cy, BMF_24BPP,
    BMF_NOZEROINIT, NULL);

if (pDevice->hSurface == NULL)
    return NULL;

EngAssociateSurface(pDevice->hSurface, pDevice->hDevice,
    HOOK_BITBLT | HOOK_STRETCHBLT | HOOK_TEXTOUT |
    HOOK_STROKEPATH | HOOK_FILLPATH | HOOK_STROKEANDFILLPATH |
    HOOK_COPYBITS | HOOK_LINETO);

return pDevice->hSurface;
}
```

Although the HTML driver hooks out quite a few drawing calls, their implementation does only simple parameter logging before punting back to the graphics engine. Here is one of them, which represents the rest. The first parameter of all DDI drawing calls is a SURFOBJ pointer, which is the graphics engine data structure for a drawing surface. Its dhpdev member is a device-provided handle to a physical device, returned by DrvEnablePDEV. In our case, the handle can be cast into a KDevice pointer.

```
BOOL APIENTRY DrvBitBlt(SURFOBJ *psоТrg,
    SURFOBJ *psоТSrc,
    SURFOBJ *psоТMask,
    CLIPOBJ *pco,
    XLATEOBJ *pxlo,
    RECTL *prclTrg,
    POINTL *pptlSrc,
    POINTL *pptlMask,
    BRUSHOBJ *pbo,
    POINTL *pptlBrush,
    ROP4 rop4)

{
    KDevice * pDevice = (KDevice *) psоТrg->dhpdev;

    if ( pDevice->CallEngine(INDEX_DrvBitBlt, &psоТrg, 11) )
        return EngBitBlt(psоТrg, psоТSrc, psоТMask, pco, pxlo,
            prclTrg, pptlSrc, pptlMask, pbo,
            pptlBrush, rop4);
    else
        return FALSE;
}
```

Those are the interesting parts within the HTML driver. Here is an abridged version of what it generates when printing the standard printer driver test page. Your web browser can display it nicely, as shown in [Figure 2-9](#).

Figure 2-9. Viewing the HTML printer driver output in a browser.



```
<html>
<head>
<title>Test Page </title>
</head>
<body bgcolor=#80B090><font size=1>
<ol>
<li>DrvStartDoc(e2229558, e1e86488, 2)</li>
<li>DrvStartPage(e2229558)</li>
<li>DrvFillPath(e2229558, f1d2fa68, ..., 1)</li>
<li>DrvBitBlt(e2229558, 0, 0, f1d2f7b0, ..., f0f0)</li>
...
<li>DrvTextOut(e2229558, f1d2f824, ..., d0d)</li>
<li>DrvTextOut(e2229558, f1d2f824, ..., d0d)</li>
<li>DrvSendPage(e2229558)</li>
<p></p>
<li>DrvEndDoc(e2229558, 0)</li>
</ol>
</body>
</html>
```

A simple printer driver, or, to be exact, its graphics DLL, is that simple. It can be even simpler if we don't want to handle drawing calls to log parameters. A real printer driver is much more complicated than this. If you want to check for yourself, check the sample plotter driver that comes with Microsoft Windows 2000 DDK, or the Postscript driver on Windows NT 4.0 DDK. A feature-rich, high-quality, high-performance printer driver is even more complicated than what the DDK sample code shows.

2.8 SUMMARY

This chapter covered the whole spectrum of the Windows graphics system architecture. We discussed the overall architecture of the Windows graphics system, Win32 GDI client architecture, DirectX architecture, printing architecture, graphics engines, display drivers, and printer drivers. We developed a tool to show the graphics system service calls both on the client side and on the server side and finished by showing a simple HTML printer driver.

The main takeaways from this chapter may be the few block diagrams showing different levels of the graphics system architecture we are interested in. We should have a general idea which feature of the Windows NT/2000 graphics system is handled by which component of the system, and how a Win32 API call is implemented by different components on the graphics system food chain.

While this chapter focused on architecture, or block diagrams, which programmers may not really like, we will now proceed to [Chapter 3](#) on exploring the unknown data structure behind GDI and DirectDraw, and then to the more exciting [Chapter 4](#) to look at behind-the-scenes runtime behavior.

Further Reading

This chapter covered a lot of ground, so its descriptions are far from complete. Refer to the sources cited below if you're interested in specific areas.

[Section 2.1](#) discusses the overall architecture of Windows NT/2000 Graphics Systems. Microsoft MSDN provides much more detail on the individual components. If you want to get a deep understanding of GDI and DirectDraw, just continue with this book. Otherwise, *OpenGL Programming Guide*, by Mason Woo et al., is a standard reference on OpenGL. If you are interested in Windows 95/98's GDI internals, Matt Pietrek's books can give you some insight.

[Section 2.2](#) gives a general overview of GDI, which will be the topic of almost the entire book. The real new information here is the system service calls. Beside reading the list of system service calls, as generated by the SysCall program, you may want to install the debug symbols for your NT/2000 and step through a few GDI calls to see how they are really mapped to system service calls.

[Section 2.3](#) gives an outline of DirectX architecture, which is actually huge. This book covers only the DirectDraw part of DirectX, both its hidden data structures and API level concepts. For other parts of DirectX, *Tricks of Windows Game Programming Gurus*, by Andre Lamothe, is a good source of information for using DirectX in game programming and for game programming in general.

[Section 2.4](#) touches on the architecture of the Windows printing system. We will devote [Chapter 17](#) to Win32 printing API, after covering EMF in [Chapter 16](#). For the whole printing system, Microsoft DDK is the best source of information.

[Section 2.5](#) goes underneath the API layer to look at the Windows graphics engine. Again, Microsoft DDK is the best source of information. If you want to experiment on your own, the HTML printer driver presented in [Section 2.7](#), [Chapter 3](#) on the graphics system internal data structure, and [Chapter 4](#) on graphics system spying can certainly help a lot.

[Section 2.6](#) describes briefly the graphics device driver in general and the video display driver in particular. For more details, refer to the well-written DDK document and also to source code for several sample video display drivers.

[Section 2.7](#) on printer drivers may be the most programming-oriented part of this whole chapter. It describes how a simple HTML driver can be written, which heavily replies on the graphics engine and yet does something useful. It's definitely worth playing with. [Chapters 3](#) and [4](#) will explore further the internal workings of the graphics system. www.paulyao.com provides printer driver training.

Sample Programs

The complete programs described in this chapter can be found on the CD that comes with the book, as listed in [Table 2-4](#).

Table 2-4. Sample Programs for Chapter 2

Project Location	Description
Sample\Chapt_02\SysCall	Lists system service calls made in Win32 subsystem DLLs (NTDLL.DLL, GDI32.DLL, and USER32.DLL) and system service in the OS kernel (NTOSKRNL.EXE, WIN32K.SYS).
Sample\Chapt_02\Timer	Compares four different methods for timing: GetTickCount, time GetTime, QueryPerformanceCounter, and reading Intel Pentium CPU clock–cycle count.
Sample\Chapt_02\HTMLDrv	HTML printer driver, a printer driver which generates HTML pages, logs DDI commands, and renders pages as embedded bitmaps.

Chapter 3. GDI /DirectDraw Internal Data Structures

The Windows API often claims to be object based, which is close to, but not exactly the same as, object oriented. To use the Win32 API, you often have to create objects of some kind, manipulate them through some functions, and then finally destroy them. When you create an object, the operating system is completely managing the internal representation of the object, and you as a programmer only see a handle.

GDI uses dozens of different kinds of objects, device contexts, logical pens, logical brushes, logical fonts, logical palettes, device-independent bitmaps, DIB sections, etc. For all of them, you only see a handle, a mysterious handle which you can't do anything about except pass it back to GDI.

This chapter will expose every little detail about GDI handles, and more importantly the data structure behind those GDI handles. We will learn the meaning of every bit in a GDI handle, how GDI handles are mapped to an entry in the GDI handle table, the limitations of the handle table, and even the data structure GDI internally keeps for each type of GDI objects. We will also cover DirectDraw. We use logical reasoning, detective-style searching techniques, Microsoft tools, and tools created for this chapter to achieve our goal of understanding key GDI/DirectDraw data structure.

You may not be so interested in the minute details about GDI data structures. But understanding the general design of GDI/DirectDraw internals can help you become a knowledgeable Windows programmer. This chapter also covers some hard-core Windows programming stuff, like walking through virtual memory, writing a kernel mode device driver (not another printer driver), and hosting WinDbg debugger extension to explorer NT/2000 kernel on a single machine.

3.1 HANDLES AND OBJECT-ORIENTED PROGRAMMING

In an object-oriented language or environment, an object is a collection of data and functions that models something in the real world or our imaginary world. Objects are classified into different classes according to their commonalities. Actually, an object-oriented language mainly focuses on class definition; an object is just a runtime instantiation of a class.

The Win32 API also defines different classes of objects. For GDI, the commonly seen objects are device contexts, logical pens, logical brushes, logical fonts, logical palettes, and device-dependent bitmaps. So all device context objects are instances of the device context class; all logical palettes are instances of the logical palette class.

Class and Object

A class in an object-oriented language encloses both data as member variables of the class, and code as member functions of the class. Access to either of them is controlled by the class definition. Some members are private and protected; some are public. To create an instance of a class, memory is first allocated, and then a constructor is called. Use of private and protected members of the class hides information away from code using the class. This encapsulation and information-hiding concept is one of the cornerstones of object-oriented programming.

For the Win32 API, implementation of a certain class is also well encapsulated. As we will later find out, an object is always represented by a number of data fields, usually a structure or even a complicated web of structures linked together. For each class, there is a predefined set of functions that can be applied to objects of that class. For example, a device context is a GDI object, an instance of the device context class. It has functions like GetTextColor and SetTextColor to retrieve/set the text color. We know from programmer's instinct that the text color must be a data member associated with a device context object, but we have no idea where it is and how it is represented internally. In other words, a device context's internal implementation is completely hidden from application programmers.

Encapsulation and Information Hiding

In normal object-oriented programming practice, information hiding is achieved by declaring certain members as private or protected, so the client side code can't access them directly. But the compiler still needs to know perfectly well all members, their types, names, and orders in a class. At least, the compiler needs to know the exact size of an instance of an object for memory allocation. This can cause lots of problems for the modular development of programs. Every time a data member or member function is changed, the whole program needs to be recompiled. Programs compiled with older versions of class definition would not work with newer versions. To solve this problem, there is the abstract base class. The abstract base class, which uses virtual functions to define the interface the client-side program can see while completely hiding away the implementation, improves information hiding and the modularity of programs even further. A COM interface, which is a class with no data members and only pure virtual functions, represents an extreme of information hiding. When a class has a pure virtual function, it cannot be used to create an object directly. You have to derive a class from it, implement all its pure virtual functions, and create an instance of the derived class. To hide the derived class from its client, a separate static function is provided for object creation. For example, a COM DLL always exports a function named DllGetClassObject, which is responsible (with the help of class factory) for the creation of new objects supported by the COM DLL. For hiding the implementation away from the client side of a class, normally a special function is provided to create an instance of a

derived class, including memory allocation; another special function is provided to destroy an instance, including freeing its memory.

Objects in the Win32 API can be seen as being implemented using abstract base class with no data members. The data representation of an object is completely hidden from the user application. The benefit is enormous; a program compiled for Win32s (a subset of the Win32 API implemented on Windows 3.1) can run without problems on Windows 95; a program compiled for Windows 95 can run on Windows NT or Windows 2000 without problems. The binary code of a Win32 program is compatible among different versions of operating systems implementing the Win32 API on the same CPU. Although there are certain limitations to the promise of a unified Win32 API, a substantial subset of the Win32 API is implemented on multiple platforms with the same semantics. For GDI, Windows 95/98 implementation is heavily based on 16-bit GDI implementation used in Windows 3.1, Windows NT 3.51 implements GDI using a separate system process in user mode, and Windows NT 4.0 and Windows 2000 use a 32-bit kernel mode graphics engine. They have significant differences in implementation, yet the perfect information hiding provided by the Win32 API greatly improves the portability of programs. GDI normally provides several functions to create an instance of an object and several functions to destroy them.

To illustrate our comparison between object-oriented programming and the Win32 API, let's try to provide some minimum pseudo-implementation of GDI using C++. [Listing 3-1](#) shows the result.

Listing 3-1 Pseudo-implementation for GDI in C++

```
// gdi.h
class _GdiObj
{
public:
    virtual int GetObjectType(void) = 0;
    virtual int GetObject(int cbBuffer, void * pBuffer) = 0;
    virtual bool DeleteObject(void) = 0;
    virtual bool UnrealizeObject(void) = 0;
};

class _Pen : public _GdiObj
{
public:

    virtual int GetObjectType(void)
    {
        return OBJ_PEN;
    }

    virtual int GetObject(int cbBuffer, void * pBuffer) = 0;
    virtual bool DeleteObject(void) = 0;

    virtual bool UnrealizeObject(void)
    {
        return true;
    }
};

_Pen * _CreatePen(int fnPenStyle, int nWidth, COLORREF crColor);
```

```
// gdi.cpp
#define STRICT
#define WIN32_LEAN_AND_MEAN
#include <windows.h>
#include "gdi.h"

class _RealPen : public _Pen
{
    LOGPEN m_LogPen;

public:
    _RealPen(int fnPenStyle, int nWidth, COLORREF crColor)
    {
        m_LogPen.lopnStyle = fnPenStyle;
        m_LogPen.lopnWidth.x = nWidth;
        m_LogPen.lopnWidth.y = 0;
        m_LogPen.lopnColor = crColor;
    }

    int GetObject(int cbBuffer, void * pBuffer)
    {
        if ( pBuffer==NULL )
            return sizeof(LOGPEN);
        else if ( cbBuffer>=sizeof(m_LogPen) )
        {
            memcpy(pBuffer, &m_LogPen, sizeof(m_LogPen));
            return sizeof(LOGPEN);
        }
        else
        {
            SetLastError(ERROR_INVALID_PARAMETER);
            return 0;
        }
    }

    bool DeleteObject(void)
    {
        if ( this )
        {
            delete this;
            return true;
        }
        else
            return false;
    }
};

_Pen * _CreatePen(int fnPenStyle, int nWidth, COLORREF crColor)
```

```
{  
    return new _RealPen(fnPenStyle, nWidth, crColor);  
}  
  
// test.cpp  
void Test(void)  
{  
    _Pen * pPen = _CreatePen(PS_SOLID, 1, RGB(0, 0, 0xFF));  
    ////  
    pPen->DeleteObject();  
}
```

[Listing 3-1](#) defines an abstract base class for the generic GDI object using class `_GdiObj`. It only has four pure virtual functions with no data members. The generic pen class `_Pen` is derived from `_GdiObj`. It implements two virtual functions and leaves the other two still as pure virtual functions. A `CreatePen` routine is provided to create an instance of the Pen class. In the implementation file (GDI.cpp), the real pen class (`_RealPen`) uses the `LOGPEN` structure to store the information about a pen. It's a full implementation of the abstract class `_Pen` with a constructor and implementations for the two remaining pure virtual functions. The `_CreatePen` routine creates an instance of the `_RealPen` class, and passes its pointer as a pointer to the generic pen class `_Pen`.

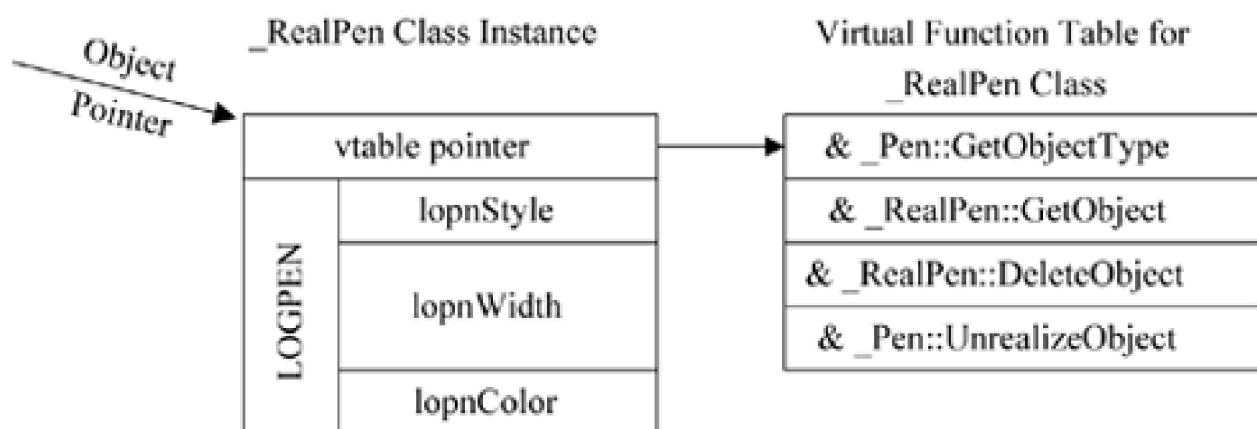
The client side does not need to know how much space a pen occupies, where memory is allocated, and how the virtual functions are implemented. These implementation details are completely hidden. A client only needs to remember the methods in the interface, their meaning, and their semantics.

Pointer vs. Handle

To create an object in an object-oriented language, a block of memory needs to be allocated to hold the member variables of the object. If its class has any virtual function, an extra pointer is allocated together with the member variables, and it's assigned a pointer to a table of all virtual function implementation routines for that class. A pointer to an object is crucial to a language like C++. It's passed to all nonstatic member functions of the class so that data members of the object can be accessed, and the right virtual functions can be called. Such a pointer is referred to as "this" pointer in C++.

[Figure 3-1](#) illustrates what an object pointer points to using the `_RealPen` class shown [Listing 3-1](#). For the `_RealPen` class, 16 bytes are needed to store the only data member, `m_LogPen`; 4 bytes are needed to store the vtable (virtual function table) pointer. So at least 20 bytes need to be allocated for each object. The vtable pointer points to a table of four function pointers. In this case, two of them are implemented in the `_Pen` class, and the other two are implemented by the `_RealPen` class.

Figure 3-1. Sample C++ object representation.



An object pointer in COM is normally known as an interface pointer, which is defined as a pointer to a pointer to a table of functions. Behind the scenes, an interface pointer is just a C++ object pointer or a pointer to the middle of an object. In our example, the `_CreatePen` routine creates an instance of the `_RealPen`, but returns a pointer of `_Pen` class. Client-side code does not know anything about the data representation, just as in COM.

For the Win32 API, although for each object a block of data is allocated somewhere, Microsoft does not want to return a pointer to the user application. A pointer may be considered to contain too much information for "smart" programmers. It gives away the exact location of an object's storage. Pointers normally allow for read/write access to internal representation of objects that an operating system may want to hide. Pointers make it harder to share objects across process address spaces.

To further hide information away from programmers, instead of returning a pointer, normally a Win32 object creation routine returns a handle to an object. A handle is defined as a value that uniquely identifies an object, or an indirect reference to an object. To be more precise, a handle is a value that has a one-to-one correspondence to an object. An object can be mapped to a unique handle, and a handle can be mapped to a unique object. To ensure that the handle mechanism really works, the mappings between objects and handles are not documented, not guaranteed to be fixed, and are known only to Microsoft, or maybe a few makers of system-level tools.

The mapping between object pointers and handles can be visualized as implemented by two functions: `Encode` and `Decode`, whose prototypes are shown below:

```
HANDLE Encode(void * pObject); // map pointer to handle
void * Decode(HANDLE hObject); // map handle to pointer
```

Identity Map

In the extreme case, a handle can be the same as the object pointer when both `Encode` and `Decode` only do type casting. The mapping between object pointers and handles is essentially an identity mapping.

In the Win32 API, an instance handle (HINSTANCE) or a module handle (HMODULE) is simply a pointer to an image of PE (Portable Executable) file mapped into memory. LockResource is suppose to lock a global handle to a resource to a pointer, but actually they have the same value. The resource handle as returned by LoadResource is just a pointer to a memory-mapped resource in disguise.

Table-based Mapping

The most common mechanism of mapping between an object and its handle is table-based mapping. The operating system creates a table to hold all objects under consideration, or at least first level representation. When a new object needs to be created, an empty entry in the table is sought. It's then filled with data to represent the object. When an object is deleted, its data members are freed and its entry in the table is free to be reused.

The indexes in the table are good candidates as handles to objects managed using such a table-based object management scheme. Encoding and decoding are pretty simple too.

In the Win32 API, kernel objects are known to be implemented using a perprocess table. Kernel objects include mutex, semaphore, event, registry key, port, file, symbolic link, object directory, memory-mapped file, thread, process, Window station, desktop, timer, etc. To accommodate the large amount of kernel objects, each process has its own kernel object table. Part of the NT/2000 kernel Executive is the object manager, which only manages kernel objects. One of the functions provided by the object manager is `ObReferenceObjectByHandle`. According to DDK documentation, it provides access validation on the object handle, and when access can be granted, returns the corresponding pointer to the object's body. So it's essentially a decoding routine which translates object handle to object pointer, with some extra security checking. There is a very good tool, `HandleEx`, available from www.sysinternals.com, which can list kernel objects on Windows NT/2000 machines.

A Handle Is Not Enough

While a handle provides nearly perfect abstraction, information hiding, and protection, it is also a major source of frustration for programmers. With a handle-centric API like the Win32 API, Microsoft does not have to document the internal representation of objects and how they are managed. No reference implementation is provided. Programmers have only function prototypes, Microsoft documentation, and books which are more or less based on knowledge from Microsoft documentation.

The first class of problems programmers face involves system resources. When an object is created and its handle is returned, no one has any idea what resources are consumed because their internal representation is unknown. Should a programmer keep and reuse an object, or should he/she delete the object as soon as it's not needed? There are three types of bitmaps supported by GDI—which one should you use to reduce system resource consumption?

CPU time is the main resource the computer has. When internal representation is hidden from programmers, it's hard for programmers to judge the complexity of certain operations in complicated algorithm design. If you use GDI to compose a complicated region, is the algorithm $O(n)$ (of the order of the number of items), $O(n^2)$ (of the order of the square of the number of items), or $O(n^3)$?

With implementation completely hidden, debugging is also a big problem. My program displays garbage after running for 5 minutes, so I guess there is a resource leak, but where is it, and how should I fix it? I'm a system administrator dealing with hundreds of applications on a system; when system resources are low, how can I find the troublemaker? The only available tool for resource-leak detection seems to be Bounds Checker, which relies on API spying techniques to detect mismatches between object creation and deletion.

The most frustrating problem may be compatibility of programs. Why can a program pass GDI objects across from one process to another on Windows 95, and not on Windows NT/2000? Why can't Windows 95 handle large device-independent bitmaps? The "perfect" API abstraction sometimes does not have the same semantics when implemented on different systems.

In the rest of this chapter, we will start from GDI handles and explore the undocumented world behind the handles

for Windows NT/2000.

[< BACK](#) [NEXT >](#)

3.2 DECODING GDI OBJECT HANDLES

When you create a GDI object, you're given back a handle to that object. The type of the handle could be HPEN, HBRUSH, HFONT, or HDC, depending on the type of GDI object you're creating. But the most generic GDI object type is HGDIOBJ. HGDIOBJ is defined as a void pointer. The actual compile type definition of HPEN varies according to a compile-time macro STRICT. If it is defined, HPEN is defined by:

```
struct HPEN__ { int unused; };
typedef struct HPEN__ * HPEN;
```

If STRICT is not defined, HPEN is defined by:

```
typedef void * HANDLE;
typedef HANDLE HPEN;
```

In plain English, when the STRICT macro is defined, a HPEN is a pointer to structure with a single unused field, otherwise a HPEN is a void pointer. The C/C++ compiler allows passing any type of pointer as a void pointer, but not the other way around. Two nonvoid pointers of different types are not compatible with each other. In the STRICT version, compile will be able to give warning on improper mixing of GDI object handles with each other, or with non-GDI handles like HWND, HMENU, etc., while in the non-STRICT version, you can mix different handles without getting a warning at compile time. For example, when STRICT is defined, it's acceptable to pass a HPEN to a function accepting HGDIOBJ—for example, DeleteObject, while it's unacceptable to pass a HGDIOBJ without casting to a function accepting a HBRUSH—for example, FillRgn.

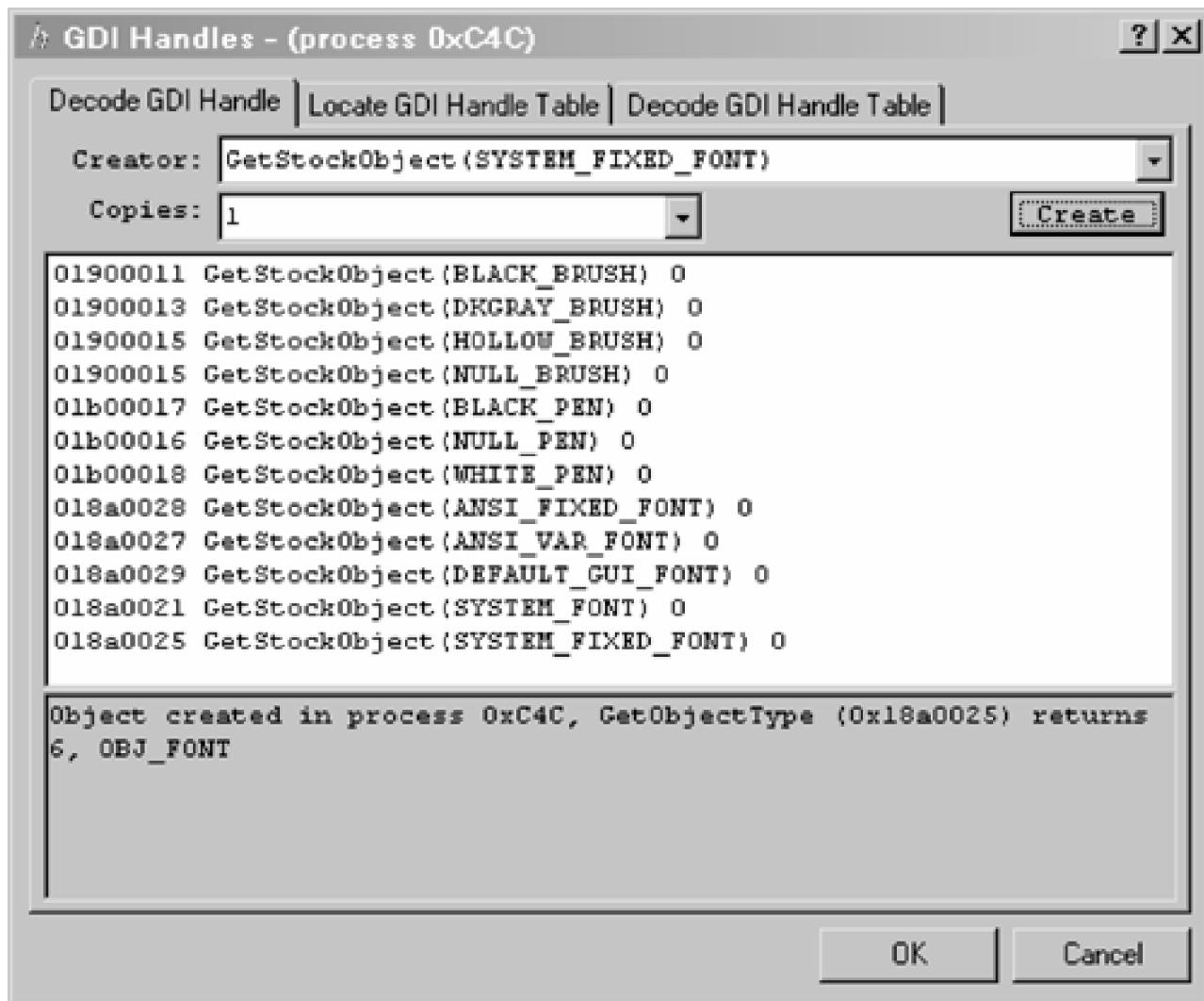
These clear definitions of different GDI handles mimic the right class hierarchy of different classes of GDI objects without using the true class hierarchy.

GDI objects support only a single handle per object, so you can't duplicate a handle to create another handle for the same object. Handles to GDI objects are normally private to a process. That is, only the process that created the GDI object can use it. Handles passed from another process are invalid.

GDI objects normally have multiple creation functions and one destruction function which accepts HGDIOBJ, DeleteObject. Besides creating objects directly, you can use GetStockObject to retrieve the handles of a few precreated GDI objects, or you can use more sophisticated functions to convert a resource attached to a module to a GDI object. The resource loading functions like LoadBitmap or LoadImage create the necessary GDI object on your behalf.

The description given so far is what you normally will get from on-line documents about GDI object handles. But we would like to know much more about those handles. The details about Windows NT/2000 GDI handles are never documented and there are no available tools to help you explore them. So we have developed a quite complicated program, GDIHandles, to help us out. It's a property-sheet based program with functionality divided into three property-sheet pages. We will gradually discuss the design, implementation, and usage of the program as we move along. For the moment, let's look at the first page, Decoding GDI Handles, as shown in [Figure 3-2](#).

Figure 3-2. Decoding GDI handles: handles of stock objects.



The upper part of the page has two combo-boxes to choose the way to create an object, ranging from various GetStockObject calls to CreateEnhMetafile, and the number of objects to create, ranging from 1 to 65,536. After choosing the creator and number of copies, you can hit the Create button to create the specified number of objects. The handles returned from the creator will be displayed in hexadecimal form in the big list box, together with the name of the creator and the sequence number within the batch (starting from 0). The creation loop stops when the creator fails, or when it returns the same value as the last call.

With the "Decode GDI Handles" page, you can design several experiments to observe the making of GDI objects and the pattern of their handles.

Stock Object Handles Are Constants

The handles returned by GetStockObject calls seem to be always constants, no matter the order in which they are called. For example, GetStockObject(BLACK_BRUSH) returns a precreated (stock) black brush object whose handle is always 0x01900011; GetStockObject(BLACK_PEN) returns a precreated black pen object whose handle is always 0x01b00017. Even if you run two instances of the program, GetStockObject returns the same value in both processes. A possible explanation is that stock objects are created once when the system initializes and are reused

by all processes.

HGDIOBJ Is Not a Pointer

Although the Windows header file defines GDI handles to be pointers, when we examine the values of these handles, they do not look like pointers at all. Generating a few GDI object handles and looking at the hexadecimal display of handles returned, you will see that the result varies from 0x01900011 to 0xba040389. If HGDIOBJ is a pointer, as defined in header file WINGDI.H, the former would be an invalid pointer to a free region in user address space, while the latter is addressing kernel address space. This suggests that GDI handles are not pointers. Another observation is that GetStockObject(BLACK_PEN) and GetStockObject(NULL_PEN) are only off by one, which could not possibly be the amount of space needed to store the internal GDI object, if handles are really pointers. So we can safely say, HGDIOBJ is not a pointer.

Process GDI Handle Limit Is Around 12,000

If you choose to call CreatePen 16 times, 16 new logical pen objects will be created. But if you want 65,536 logical pens, not all of them will succeed. Only about 12,000 pens are created successfully during our test; the rest will fail. When this happens, you can notice that the “Creator” and “Copies” combo boxes are not displayed properly.

You cannot even capture the screen using the Print Screen key when the GDIHandles main window is the foreground window. But you can capture the screen when the other program is in the foreground.

Interestingly enough, when CreatePen fails to create more GDI objects in one process, other processes in the system are still running fine.

So it seems that there is a per-process limitation on the number of active GDI object handles, which is around 12,000.

Note

According to our test, Windows NT properly enforces this per-process GDI handle quota to ensure that a single process can't mess up the whole GDI system. But the first release of Windows 2000 fails to enforce the same per-process limitation, which should be considered as a defect.

Systemwide GDI Handle Limit Is 16,384

Now run two instances of GDIHandles on the same system. Try to call CreatePen 8192 times in both processes. The first process creates all objects fine; the second will stop at around 7200.

When the second process fails, the whole screen is a big mess. Other processes can't display properly, even if you try to switch to them. You have to guess when to press an OK button to close one program to free some GDI resources, trying to let the screen refresh to watch what happened to the second process.

Although Microsoft documents give the impression that GDI objects use only local process resources, this

experiment shows that GDI objects are allocating from the same pool of resources. Processes within the system affect each other when GDI resources are used.

Adding 8192 and 7200 gives 15,392. Considering GDI object handles used by the GDIHandles property sheet and pages and by other processes, we can make an educated guess that there is a systemwide limitation on the number of GDI handles, which is 16,384.

Part of HGDIOBJ Is an Index

If we create lots of GDI objects using GDIHandles, pay special attention to the lower words of the double word handles displayed; you will find them to be within the range of 0x0000 to 0x3FFF. The lower words are always unique within the process, even unique across the process boundary except for stock objects.

The lower words of handles sometimes come in increasing order, sometimes in decreasing order. This happens even across processes. For example, in some cases, Create Pen in one process returns 0x03C1 in the lower word; the next CreatePen in another process could return 0x03C2 in its lower word.

One explanation for these behaviors is that the lower word of HGDIOBJ is an index to a systemwide table of 16,384 (0x4000) GDI objects.

Part of HGDIOBJ Is a GDI Object Type

On Windows NT/2000, a GDI object handle is returned as a 32-bit value. The GDIHandles program displays them as 8 hexadecimal digits. Now that we know that the lower 4 hexadecimal digits of the GDI handles form an index, we can focus on the upper 4 hexadecimal digits.

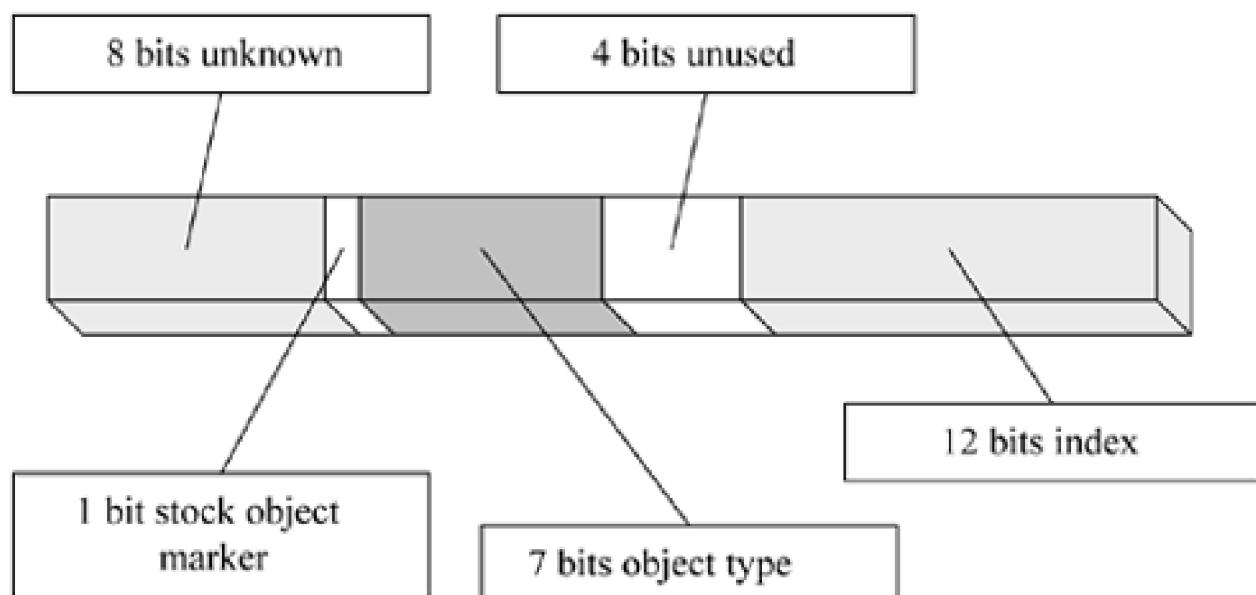
If we create the objects according to their object type—for example, create several brushes, followed by a few pens, a few fonts, DC, etc.—it's not so hard to find out that GDI object handles of the same type have something in common. Objects of the same type have almost the same third and fourth hexadecimal digits in their object handles.

Brushes always have 0x90 or 0x10 in their object handles' third and fourth digits; pens always have 0x30 or 0xb0; fonts always have 0x8a or 0xa0; palettes always have 0x88 or 0x08; bitmaps have 0x05; device contexts have 0x01.

Those object handles having their most significant bit being 1 are all stock objects. So we have enough evidence to say that object handles' third and fourth hexadecimal digits are an encoding of object type, plus a stock object marker.

The remaining two hexadecimal digits in the 32-bit GDI handle value do not show a meaningful pattern for the moment. Let's just summarize what we do know about Windows NT/2000 GDI handles up to now. GDI handles are made up of 8 unknown most significant bits, 1 bit stock object marker, 7 bits object type information, 16 bits index with the upper 4 bits always being 0. With the 7 bits object type information, we could determine device context, region, bitmap, palette, font, brush, enhanced metafile, pen, and extended pen. [Figure 3-3](#) illustrates the composition of a GDI handle.

Figure 3-3. GDI handle encoding for Windows NT/2000.



Here are some C++ type and function definitions to help you in composing or decoding GDI handles.

```
typedef enum
{
    gdi_objtypeeb_dc      = 0x01,
    gdi_objtypeeb_region   = 0x04,
    gdi_objtypeeb_bitmap    = 0x05,
    gdi_objtypeeb_palette   = 0x08,
    gdi_objtypeeb_font      = 0x0a,
    gdi_objtypeeb_brush     = 0x10,
    gdi_objtypeeb_enhmetafile = 0x21,
    gdi_objtypeeb_pen       = 0x30,
    gdi_objtypeeb_extpen    = 0x50
};

inline HGDIOBJ makeHGDIOBJ(unsigned top, bool stock,
                           unsigned objtype, unsigned index)
{
    return ((top & 0xFF) << 24) |
           ((stock & 1) << 23) |
           ((objtype & 0x7F) << 16) |
           (index & 0x3FFF);
}

inline bool IsStockObj(HGDIOBJ hGDIObj)
{
    return ((unsigned) hGDIObj) >= 0x00800000;
}

inline unsigned GetObjType(HGDIOBJ hGDIObj)
{
```

```
return (((unsigned) hGDIObj) >> 16) & 0x7F;  
}  
  
inline unsigned GetObjIndex(HGDIOBJ hGDIObj)  
{  
    return ((unsigned) hGDIObj) & 0x3FFF;  
}
```

With these inline functions, you can check whether a GDI object is a stock object and get its object type and object index.

[< BACK](#) [NEXT >](#)

3.3 LOCATING THE GDI OBJECT HANDLE TABLE

Experiments in [Section 3.2](#) show that the lower word of a GDI object handle (HGDIOBJ) is an index from 0 to 0x3FFF. So we can guess that there is a table of GDI object handles somewhere that is managed by the system, presumably GDI, which the indexes index into. We've seen those tables in Win31 and Win95, and it makes sense for them to reappear in Windows NT and Windows 2000. The problem we want to solve in this section is how to find this undocumented table.

Windows programmers will normally ask if there is a Win32 API function that does that, or, better still, if there is an MFC function which the MSVC wizard can automatically generate. The answers to both questions are negative. There seems to be no other document that confirms the existence of this table, not to mention a documented API to get to this table.

Let's take off the hat of a programmer who only knows APIs and library functions and put on the hat of Sherlock Holmes for a little while. Maybe we only need to be a middle-school student trying to solve a simple geometric problem. Now we can see some light in this treasure-hunting mission.

First let's assume there is indeed a table of GDI objects, for which we are now trying to find a proof by construction—that is, we are trying to find the location of this table.

If the table does exist, most likely it's readable from user address space. The reason is that GDI32.DLL lives in user address space, too, the same as your own EXEs and DLLs. If this table is only readable from kernel address space, then for a little trivial job like GetObjectType(), GDI32 needs to call its kernel mode graphics engine Win32K.Sys for an answer, which could make GDI very slow. So let's make our second assumption that the GDI object table is at least readable from a user mode program—that is to say, its address is within the first 2 GB of Win32 address space.

If there is a GDI object table, creating a new GDI object will add an item to the table, filling in some data, which changes the contents of the table. We stress creating a new GDI object here because we've seen that GetStockObject always returns the same result, which may only return a precreated handle without creating a new object, so it may not change the table contents.

If creating an object changes the GDI table, then we can compare all memory locations before and after creating a new GDI object to locate the table. Based on our assumptions, we only need to compare memory in user address space, not worry about kernel address space. After the comparison, one of the changed memory locations must be within the GDI object handle table area.

Now that we have some ideas, an algorithm is due. We only need to save user addressable memory space before and after creating a simple GDI object, then read those spaces back and compare, byte by byte; any location having a difference could be where the GDI object handle table is. Simple ideas like this will never work in reality. Reading the first byte at 0 in user address space will generate a protection fault, which is meant to catch any dereferencing of the NULL pointer. There are other regions within the 2-GB user space that are not readable. Even writing, reading, and comparing all readable memory regions could consume lots of disk space and be very slow.

To ensure we scan only the readable regions within the user address space, we can use the Win32 API function VirtualQuery, which lets you divide the user virtual address space into regions with the same protection flags, such as read-only, writeable, and executable. Generating a CRC checksum on the readable memory regions reduces significantly the amount of memory we need to save and compare. Here is the working algorithm, a function for taking snapshots of memory regions and comparing them:

```
void shot(vector<CRegion> & Regions)
{
    MEMORY_BASIC_INFORMATION info;

    for (LPBYTE start=NULL;
        VirtualQuery(start, & info, sizeof(info)); )
    {
        if (info.State == MEM_COMMITTED)
        {
            CRegion * pRgn = Regions.Lookup(start,
                info.RegionSize);
            if (pRgn==NULL)
                pRgn = Regions.Add(start, info.RegionSize);

            pRgn->CRC[0] = pRgn->CRC[1];
            pRgn->CRC[1] = GenerateCRC(start,
                info.RegionSize);
            pRgn->usage++;

            if ( (pReg->usage >=2 ) &&
                (pReg->CRC[0]!=pReg->CRC[1]) )
                printf("Possible Table location %08lx",
                    start);
        }
        start += info.RegionSize;
    }
}

void SearchGDIObjectTable(void)
{
    vector<CRegion> UserRAM;

    shot(UserRAM);
    CreateSolidBrush(RGB(0x11, 0x22, 0x33));
    shot(UserRAM);
}
```

Real Windows programs are not allowed to be so user unfriendly these days, so the actual GDI table locator adds a property-sheet page to our GDIHandles program, “Locate GDI Handle Table.” The property-sheet page displays a listview showing the CRC, starting address, region size, region state and type, even the module segment name of the region if we could find one. For modules like GDI32.DLL, it uses GetModuleFileName to find the module name. For sections within a PE module, for example, the “.text” section which normally holds executable codes, the code actually walks through the internal structure of a PE file to find the section name. It also tries to identify regions that are actually process heaps and thread stacks. There is a “Query Virtual Memory” button on the sheet to start taking snapshots of virtual memory whenever you want.

Run the program, take a snapshot of the memory by pressing the “Query Virtual Memory” button; VirtualQuery nicely divides the 2-GB user virtual memory address space into over 100 regions. Most of the space is marked as F for free or R for reserved. The more interesting regions are those marked as C for committed.

Most of those committed regions are segments of program EXE files and system DLLs like KERNEL32.DLL, GDI32.DLL, and even MSIDLE.DLL. You have no idea why MSIDLE.DLL is mapped into this address space, but it's there. Several regions are marked as heaps; one is marked as stack.

For every committed region, there is a CRC checksum displayed on the left.

Now to accomplish our task, go back to the "Decode GDI Handle" page, create a solid brush, go back to "Locate GDI Table" page, and take a second memory snapshot. This time, for almost all the committed regions, we have two CRC checksums, before and after. A few regions may have only one CRC, because the region changes in its starting address or region size. [Figure 3-4](#) shows the screen display after taking the second virtual memory snapshot.

Figure 3-4. Locate GDI object table: focus on changing.

/ GDI Handles - (process 0x88C)						
Decode GDI Handle		Locate GDI Handle Table		Decode GDI Handle Table		
oldCRC	CRC	Base	Size	Type	Module	
≠ ec0e	afcl	003b0000	00002000	C M er		
		003b2000	00006000	R M er		
		003b8000	00048000	F		
≡ 69f2	69f2	00400000	00001000	C I ewc	Handles.exe	
≡ 39b8	39b8	00401000	0003a000	C I ewc	.text	
≡ 9b12	9b12	0043b000	00004000	C I ewc	.rdata	
≡ dd77	dd77	0043f000	00005000	C I ewc	.data	
≡ d044	d044	00444000	00001000	C I ewc		
≡ 9d1a	9d1a	00445000	00005000	C I ewc	.rsrc	
		0044a000	00006000	F		
≠ a16a	f49a	00450000	00043000	C M ro		
		00493000	0000d000	F		
≠ 7714	3132	004a0000	00060000	C M er		
		00500000	002a0000	R M er		
≡ 491b	491b	007a0000	00001000	C P rw		
		007a1000	0000f000	F		

The list view displays a green equal sign in the front of a row for regions having the same CRC checksums, an alarming red not-equal sign for regions with different CRCs. Regions with a single CRC after two snapshots should also be considered as changed regions.

Note

Understanding the usage of virtual memory is a must for hard-core Windows programmers. Virtual memory can explain how Win32 API is implemented, and how your program really runs. For example, the first memory block in virtual memory is a free 64-KB block starting at address zero. If you ever dereference a pointer whose higher 16 bits are zero, you're accessing this "no-fly" zone. Because it's a

free virtual memory block, any read/write access results in an access violation fault. Every thread in your program creates its own stack. A stack appears as three blocks in virtual memory: a large reserved block for the stack to grow, a single page guard block to detect stack growth, and a committed block for the part of the stack that's really being used. Each DLL/EXE in your process may create its own heap, if it's not using the DLL version of C/C++ runtime library. That's why you can find so many heaps in the virtual memory. User modules are normally loaded at the lower end of the virtual memory, and system DLLs are loaded at the higher end. The largest free memory block between them determines the largest amount of data you can process with a Win32 program at the same time. This free virtual memory block is normally around 1.5 GB.

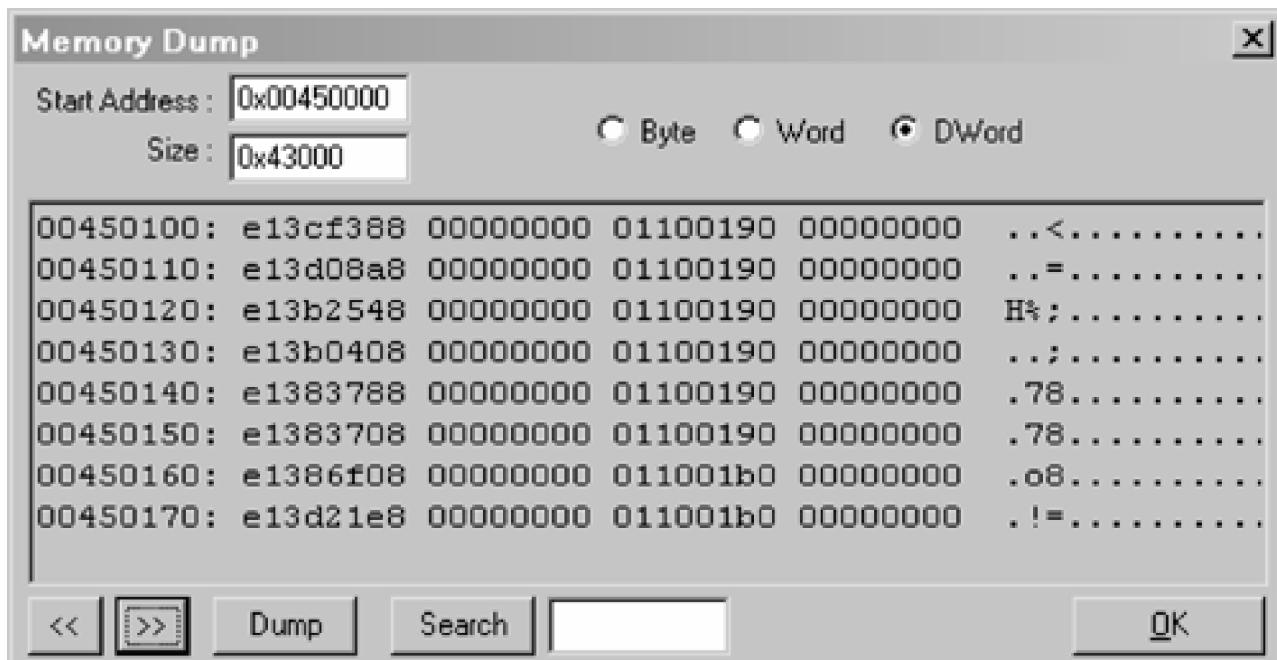
In this test run, the CRC checksums of about ten regions changed between the two memory snapshots, or they changed starting addresses or sizes. Events between snapshots include displaying the result of the first snapshot, switching one page, creating a solid brush, and switching back the property sheet page. At least one new GDI object is created, so the GDI object handle table is in one of those regions.

Dozens of regions are still too many to examine one by one. But we can easily eliminate most of them by requiring them to be above a certain limit on region size. From the previous section, we know that the number of GDI object handles is 12,000 per process, 16,000 systemwide. Assuming the minimum 4 bytes per entry in the GDI object table, the table should be more than 64 KB in size (or in hexadecimal notation 0x10000). Four bytes are the minimum amount of space needed to store a pointer to more complicated data structures.

With the size restriction, there are only two regions left, both shown in [Figure 3-4](#). Both regions are committed, mapped, write protected, and have no meaningful names. They are quite big in size, 268 KB (0x43000) and 384 KB (0x60000). One region is marked as PAGE_READONLY, another as PAGE_EXECUTE_READ. It's much easier now to examine their details to see which one could be the table we are looking for; simply double-click the first column of the row, and the property page pops up a dialog box showing the memory region in binary format.

Now double-click on the region starting from 0x00450000. Choose to decode the memory bits in double word by clicking on the Dword radio button. Paging up and down the memory region, you can clearly see a table formation, a table of 16 bytes for each entry. To verify that the table at 0x00450000 is really the table of GDI object handles we are looking for, you can do some byte-to-byte comparison. You can create a text-file dump of the region by pressing the "Dump" button, creating a bunch of GDI objects, remembering their object handles, coming back to generate another memory dump of the same region, and comparing the two using tools like WinDiff. For example, if you create 256 solid brushes, you will notice that a series of GDI handles are returned, say from 0x01300440 to 0x0130052b, so the indexes are actually 0x440 to 0x52b. Check in the memory dump for addresses from 0x00454400 to 0x004552b0; you will see that their contents before and after the object-creation calls are all different. [Figure 3-5](#) shows the memory dump of the region starting at 0x45000.

[Figure 3-5. Memory dump of a possible GDI object table.](#)



For this program, the GDI object table is found at 0x450000, but there is no reason to believe this address is fixed. If you look closely enough in the virtual address map shown in the list view, 0x450000 is just a free-space region following the main execution program Handles.exe. So for a smaller program, the table could be at 0x420000; for a much larger program, the table could be at 0x630000. We need an easier and sure way to locate this table.

Given that this table is not at a fixed address and GDI needs to use the table frequently, GDI must have a reference to it in its data segment, and we should find it. Open a "Memory Dump" dialog box for the GDI32.data section (.data), change to Dword mode, type in the address 450000, hit the Search button, and the following search report will be generated:

Search for 0x00450000 in region start at 77f78000,
size 1000 bytes
77f78008
77f780bc

The two references mean that there are two internal variable GDI32.DLL keeps, both pointing to this GDI object table.

Continue the search in GDI32's code section (.text), with the addresses of those two variables at 0x77f78008 and 0x77f780bc. You could find four references to the second variable and hundreds to the first. Comparing the addresses with QuickView or Dumpbin output of GDI32.DLL, it's apparent that lots of functions like Select Object and GetObjectType reference them.

Among the functions referencing these GDI object table pointers, one undocumented function looks very interesting—that is, GdiQueryTable. Does the name of this mysterious function sound interesting? The name suggests that there is a table of some kind and this function can query the table for us. It is time to try this mysterious function:

```
// querytab.cpp
#define STRICT
#include <windows.h>
```

```
typedef unsigned (CALLBACK * Proc0) (void);

void TestGdiQueryTable(void) {
    Proc0 p = (Proc0) GetProcAddress(
        GetModuleHandle("GDI32.DLL"), "GdiQueryTable");

    if (p)
    {
        TCHAR temp[32];

        wsprintf(temp, "%8IX", p());
        MessageBox(NULL, temp, "GdiQueryTable() returns",
            MB_OK);
    }

    return 0;
}
```

GdiQueryTable returns the same 0x45000 as does the above program. After all this trouble in digging in the virtual memory, we have accomplished our mission; there is indeed a systemwide table of GDI objects, and there is even an undocumented function GdiQueryTable() which returns its pointer. The table is read-only from user mode programs.

If you have GDI32.DLL debugger symbol files installed, debug the Handles.exe program, change the program display to assembly code, and choose the “Go To...” menu item under the Edit menu, and a dialog box will pop up. Type in 0x77f78008 or 0x77f780bc as the address. The Visual C++ debugger shows the name of the first address as _pGdiSharedHandleTable, and the second as _pGdiSharedMemory. So the first address holds a pointer to the GDI shared handle table, and the second address holds a pointer to GDI shared memory, and the two memory blocks start at the same address.

If you type in _GdiQueryTable@0 as the address, which represents a function having no parameter, the debugger will bring you to the assembly code of the undocumented function GdiQueryTable. GdiQueryTable is very simple; it just returns the contents of the pointer _pGdiSharedHandleTable.

3.4 DECODING THE GDI OBJECT HANDLE TABLE

[Section 3.2](#) tells us the GDI has a limitation of 16,384 handles, which is possibly managed by a table. [Section 3.3](#) we proved that this GDI object table does exist and it's accessible from user mode address space. [Figure 3-5](#) shows some data from the beginning of this GDI object table.

Looking at the binary dump in [Figure 3-5](#) closely, you can see a clear pattern of 16-byte cycles: a big 32-bit value, a zero 32-bit value, another nonzero 32-bit value, followed by a zero 32-bit value. The size of this alleged GDI object table is 268 KB, which gives 16.75 bytes when divided by 16,384. We have confidence to say that each entry of the GDI object table is 16 bytes. The task of this section is to decode the meaning of this 16-byte entry.

Using similar experimenting methods as in the last two sections, which we should be familiar with now, and skipping the details, we can generate the following picture of an individual entry within the table:

```
typedef struct
{
    void      * pKernel;
    unsigned short nProcess;
    unsigned short nCount;
    unsigned short nUpper;
    unsigned short nType;
    void      * pUser;
} GdiTableCell;
```

The first 4 bytes of a GDI table entry is a pointer, which normally holds a value above 0xE1000000. So it's a pointer to something in the upper 2 GB of Windows NT/2000 address space, to which only the kernel mode code has access. What we are saying is that for every GDI object, there is a kernel mode data structure which the GDI object table points to.

Note

For Windows NT/2000, the area from 0xE1000000 to 0xECFFFFFF (192 MB total) is the kernel system paged pool, which holds dynamically allocated data structure for kernel components. It's different from the nonpaged pool in that the former can be paged out of memory when system RAM is low, while the latter is guaranteed to be resident in physical memory at all times. We will later learn that GDI kernel data structures are mostly kept in paged pool, including device-dependent bitmaps (DDBs).

The next two bytes, named nProcess here, is the process identifier of the creating process of the object. Each GDI object created by a user process is marked by the process that creates it. The current process identifier can be obtained by calling Get CurrentProcessId. For certain valid objects like stock objects, this field may be 0.

The two bytes following nProcess are normally set to zero. But it could be non zero under certain conditions. It seems to be a usage count of object handles; that's why it's named nCount here.

After nCount comes nUpper, an exact copy of the upper two bytes of a GDI object handle. We know from previous sections that nUpper has an unknown upper byte and a lower byte containing type information of the object.

Field nUpper is followed by 2 bytes of internal GDI object type information, named nType above.

The last 4 bytes of a GdiTableCell, named pUser here, is clearly another pointer. pUser is NULL most of the time. When it's not NULL, it's a pointer to the lower 2 GB of the address space, which is accessible from the user mode code. For certain types of GDI objects, GDI keeps a data structure that's local to the current process. User mode pointers are accessible from kernel mode address space, but only when the specific process is the current process.

Now that we know both how to get the address of a GDI object table and the structure of each entry in the table, we can wrap the knowledge in a C++ class to make the GDI object table easily accessible to Windows programs. [Listing 3-2](#) shows the KGDItable class.

Listing 3-2 KGDItable Class for Accessing GDI Object Table

```
// GDItable.h
#pragma once

class KGDItable
{
    GDITableCell * pGDItable;

public:
    KGDItable();

    GDITableCell operator[](HGDIOBJ hHandle) const
    {
        return pGDItable[ (unsigned) hHandle & 0xFFFF ];
    }

    GDITableCell operator[](unsigned nIndex) const
    {
        return pGDItable[ nIndex & 0xFFFF ];
    }
};

// GDItable.cpp
#define STRICT
#include <windows.h>
#include <assert.h>
#include "GdiTable.h"
```

```
KGDItable::KGDItable()
{
    typedef unsigned (CALLBACK * Proc0) (void);

    Proc0 pGdiQueryTable = (Proc0) GetProcAddress(
        GetModuleHandle("GDI32.dll"), "GdiQueryTable");
    assert(pGdiQueryTable);

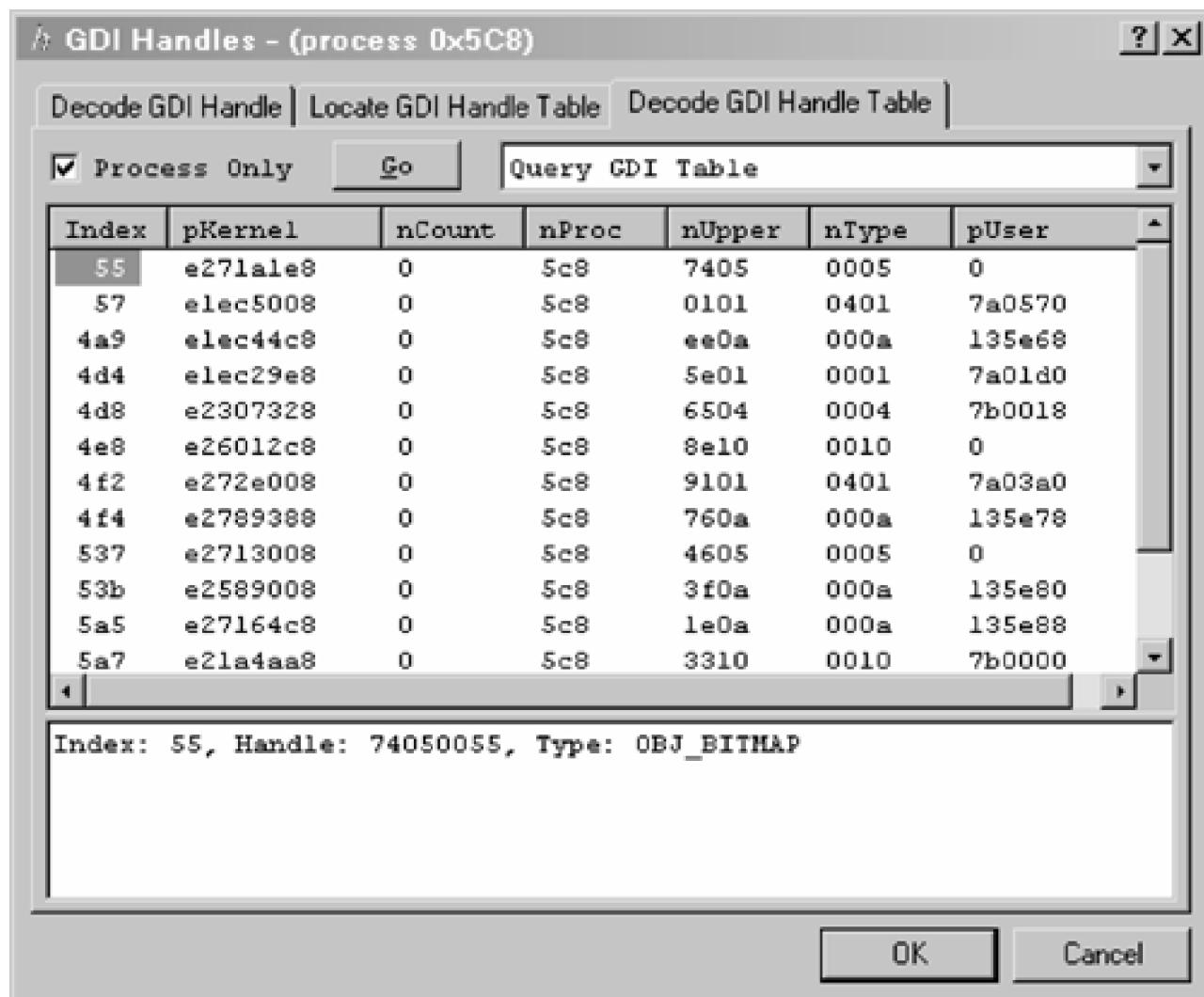
    if (pGdiQueryTable)
        pGDItable = (GDITableCell *) pGdiQueryTable();
    else
        pGDItable = NULL;
}
```

Using the KGDItable class is quite easy. Here is an example of finding the address of the black pen stock object's kernel data structure:

```
const void * BlackPenpKernel(void)
{
    KGDItable gditable;
    return gditable[GetStockObject(BLACK_PEN)].pKernel;
}
```

[Figure 3-6](#) shows a new property sheet page, “Decode GDI Object Table,” of our GDIHandles program. The main feature of this page is listing the contents of GDI object table in a list view format.

Figure 3-6. Contents of GDI object table.



The checkbox on the top left corner gives the user the option of listing all the objects in the table, or only those objects created by the current process. Once GDI objects are listed in the list view, highlight any one of them; the lower part of the property-sheet page shows the details about its index, HGDIOBJ value, and object type.

With this unique tool to display the contents of the GDI object table, we can design more experiments with GDI objects and get more insight into how GDI manages objects.

pKernel Points to Paged Pool

For every valid GDI object, pKernel is never NULL, and it's always unique. So it seems that for every GDI object, there is a corresponding data structure which is only accessible from kernel-mode code, not even from GDI32.DLL directly.

For objects of different processes, there seems to be no clear separation of memory regions as seen from pKernel values. The objects pointed to by pKernel have addresses starting with 0xE1000000. According to "Inside Windows NT," an area starting with 0xE1000000 is a pageable system memory heap normally called the "paged pool."

Visual C++ studio could not dereference these pointers for us, which prevents us from looking behind those pointers for the moment. Development studio is basically a user mode program with no special kernel mode driver supporting it. We will defer the discussion of what's behind pKernel to [Section 3.7](#), with the help of a kernel mode device driver to be developed in [Section 3.6](#).

nCount Is Partly a Selection Count

On Windows 2000, the nCount field is always zero. That is to say, it is not used. But on Windows NT4.0, it is used for certain GDI objects. To understand the meaning of nCount, we could try to select and deselect an object handle into one or multiple device contexts, and watch whether the selection and deselection could succeed now that the nCount value changes. What we will actually do in this experiment is create an object, select it into two device contexts, deselect it from them, and finally delete the object.

What this little experiment tells us is that when an object is created, its nCount starts with zero, which will not be changed for most types of objects.

For a device-dependent bitmap (DDB), nCount changes from zero to one when it's selected into a DC. If you attempt to select it again into another DC, the call will fail. When DDB is deselected from the first DC, nCount returns from one to zero. Without doubt, nCount for DDB bitmap enforces the restriction that it cannot be selected into more than one device context at a given time.

Another GDI object that uses nCount field is font, for which nCount is merely a select count without any enforcing of restriction. Selecting a logical font object into a second device context succeeds, incrementing nCount again.

One obvious question programmers will ask is whether GDI has any protection for objects selected into a device context to prevent them from being deleted. The answer is yes for palette. As you can see from [Table 3-1](#), the first DeleteObject call after the palette is selected into two DCs fails, but the second DeleteObject call after the palette is deselected from the two DCs succeeds. But nCount is not used for this protection.

For other GDI objects like font, bitmap, brush, and pen, programmers are free to delete objects at any time, leaving invalid object handles in device contexts. There is no obvious explanation why Windows does not use nCount consistently, and use it to prevent selected objects from being deleted.

Table 3-1. nCount Used as Selection Count

API Call	Bitmap (DDB)	Font	Palette
Create...()	Succeed, nCount=0	Succeed, nCount=0	Succeed, nCount=0
SelectObject(hDC1)	Succeed, nCount=1	Succeed, nCount=1	Succeed, nCount=0
SelectObject(hDC2)	Fail, nCount=1	Succeed, nCount=2	Succeed, nCount=0
DeleteObject()			Fail
(De)SelectObject(hDC2)	Fail, nCount=1	Succeed, nCount=1	Succeed, nCount=0
(De)SelectObject(hDC1)	Succeed, nCount=0	Succeed, nCount=0	Succeed, nCount=0
DeleteObject()	Succeed	Succeed	Succeed

nProcess Makes GDI Handles Process-Bound

If a program tries to use a GDI object handle of another process, Win32 API calls will normally fail. The nProcess field in GdiTableCell is the magic behind that. For stock objects like GetStockObject (BLACK_PEN), nProcess is set to zero. For other GDI objects created by a user process, nProcess is the process identifier of the creating process. To get the current process identifier, call GetCurrentProcessId().

With nProcess field, the GDI can easily check if the current process identifier matches the nProcess field of a GDI object, to enforce the rule that object handles can't be accessed in another process.

The “Decode GDI Object Table” page provides the option to list all GDI objects or only those created by the current process. When you click on a row in the table, the program displays detailed information about that object in the lower part of the page, including object information returned by GetObject call. But if you choose to list all GDI objects, click on an object created by another process and GetObject will fail; the program will show “Invalid Object.”

According to Microsoft documentation, when a process terminates, all GDI objects it has created will be freed. If you ever wonder how this is implemented, now we may have a clue. The GDI just needs to scan through the GDI object table and delete all objects with the specific process identifier.

nUpper: More Checking for Handles

The nUpper field in a GDI object table entry is found to be an exact copy of the upper two bytes of a four-byte GDI object handle—a low-cost redundancy paid to do more error checking of GDI object handles.

Suppose you created a font; CreateFont returns 0x9d0a047f. This object will now occupy entry 0x047f in the GDI object handle table, which will have nUpper== 0x9d0a. Now some other part of the program deletes the font without knowing it's still in use, which frees the entry at 0x047f; it then creates another font. Assume the GDI somehow decides it should use entry 0x047f for the new GDI object, giving it a handle of 0x9e0a047f. If the first part of program still tries to use handle 0x9d0a047f, the Win32 GDI calls will fail, because the GDI will check to find 0x9d0a does not match the new nUpper value at entry 0x047f, which is now 0x9e0a.

To experiment with this field, try to change the highest byte of a GDI object handle, keeping the remaining three bytes the same, which have object type and index information; the call to GetObject or GetObjectType will fail.

Keeping the upper word of a handle in the table may have other uses. If you only have the GDI table index part of the handle, you can reconstruct the whole handle by getting nUpper from the GDI table and combining the two together. This is useful in implementing 16-bit GDI support on Windows NT. Remember, 16-bit GDI only has a 16-bit HGDIOBJ handle, which is basically an index. For 16-bit GDI to work on Windows NT, calls need to be forwarded to 32-bit GDI, which needs full 32-bit handles.

When we discuss the meaning of GDI handles in [Section 3.3](#), the only field we do not understand is the most significant 8 bits. More experiments will show that it's a very clear recycle count, another simple mechanism for validation. For each entry in the GDI object table, its recycle count starts at 0. When a GDI object needs to be created at an entry, its recycle count is incremented (rounded back to 0 if it's 255). So when an entry is used for the first time, its recycle count starts at 0x01; that's true for all stock GDI objects that are created once and never deleted. When a GDI object is deleted and later another GDI entry is created at the same entry, even if it has the same object type, its GDI handle is different, because the recycle count has been incremented. Any code using the original handle will be rejected. [Table 3-2](#) in the next section illustrates the working of this recycle count. Now we understand every bit of a 32-bit GDI handle.

nType: Internal Object Type

In [Section 3.2](#), through the examination of GDI object handle values, we found out that every GDI object handle has a seven-bit type information. This object type information shows up in the GDI object handle table in modified form, expanded to two bytes.

The lower byte of nType is normally the same as the 7-bit type information in HGDIOBJ, while the upper byte is normally zero. The nType field treats an enhanced metafile handle as a device context handle, and extended pen handles as brush handles. For certain object types like device context, the upper byte of nType defines object subtypes like memory device context. Here is what we know about this internal object type word:

```
typedef enum
{
    gdi_int_objtypew_dc      = 0x0001,
    gdi_int_objtypew_memdc   = 0x0401, // not all memdc
    gdi_int_objtypew_region   = 0x0004,
    gdi_int_objtypew_bitmap   = 0x0005,
    gdi_int_objtypew_palette  = 0x0008,
    gdi_int_objtypew_font     = 0x000a,
    gdi_int_objtypew_brush    = 0x0010,
    gdi_int_objtypew_enhmetafile = 0x0001, // same as DC
    gdi_int_objtypew_pen      = 0x0030,
    gdi_int_objtypew_extpen   = 0x0010 // same as brush
};
```

pUser Points to User Mode Data Structure

You may have noticed that GdiTableCell is designed to be symmetric in field arrangement, starts with one pointer, followed by four 16-bit words, and then ends with another pointer, which is pUser.

pUser is normally null, except for certain types of GDI objects. When it's not null, it assumes values like 0x001420c8 or 0x790320. You can verify that they are valid addresses pointing to readable, writeable memory regions. We will look at user mode data structure for GDI objects in more detail in the next section.

3.5 USER MODE DATA STRUCTURE OF GDI OBJECTS

In the last section, we learned that for every GDI object, there is an entry in a global GDI object handle table, which has a pointer called pUser. For most GDI objects, pUser is a NULL (empty) pointer. But for the brush, region, font, and device context GDI objects, the corresponding pUser fields in the GDI object table are not empty. They do point to some interesting data structures in user mode address space, which form the topic of this section.

User Mode Brush Data: Solid Brush Optimization

For solid brush, pUser points to a block of 24 bytes, of which the first 12 bytes is a copy of LOGBRUSH structure. When a brush is a solid brush, the only essential information is the color of the brush. For other types of brush, pUser is a null pointer.

```
typedef struct
{
    LOGBRUSH logbrush;
    DWORD   dwUnused[3];
} UserData_SolidBrush;
```

If you wonder why the GDI implementation treats a solid brush in a special way, try to answer another question: What makes a solid brush different from other brushes? Solid brushes are special because they are short-lived GDI creatures needed in large quantities. In achieving a gradient fill, shadowing or lighting effects, hundreds or thousands of solid brushes are created, used once or a few times to draw slices of a picture, and then deleted immediately. Because they are needed in large quantity, the application can't keep them around until the next time they are needed; otherwise, you may easily hit the limit of the GDI object table. So most solid brushes are destroyed shortly after they are used once and are recreated when needed.

By keeping a copy of LOGBRUSH in user mode, the GDI may be able to optimize the sequential pattern of creating a solid brush, using it, and deleting it by a big amount. When the first solid brush is deleted, the GDI can just keep it internally, instead of really freeing the allocated data structure. When a new solid brush is needed, the GDI can reuse it to create the new brush by just changing the color, without going through the trouble of going to kernel mode to allocate a chunk of memory and fill it with new brush information.

We have designed a simple experiment to check the reality. Eight solid brushes are created, examined, and then destroyed in a loop. [Table 3-2](#) shows that the GDI is keeping several solid brush entries in the table to reuse them. Notice that brushes 1, 3, and 6 have the same index, 0x0e11. Their pKernel and pUser fields are the same, but their lbColor fields in the LOGBRUSH

structure pointed to by pUser are different.

Table 3-2. Solid Brush GDI Handle Reuse When Creating and Deleting Objects in a Loop

No	Handle	IbColor	pKernel	pUser
1	0xa6103e11	0x000000	0xe125d710	0x870048
2	0x3c1031f5	0x202020	0xe125d878	0x870060
3	0xa7103e11	0x404040	0xe125d710	0x870048
4	0x941031be	0x606060	0xe125da70	0x870078
5	0x3d1031f5	0x808080	0xe125d878	0x870060
6	0xa8103e11	0xa0a0a0	0xe125d710	0x870048
7	0x5f103f49	0xc0c0c0	0xe125d908	0x870090
8	0x951031be	0xe0e0e0	0xe125da70	0x870078

[Table 3-2](#) also illustrates our previous point about the recycle count (the most significant 8 bits of a GDI handle). Handles 1, 3, and 6 in [Table 3-2](#) are created at the same GDI table entry 0x3e11; they are all brushes (0x10), but their recycle counts all differ by one from the previous one.

User Mode Region Data: Rectangular Region Optimization

For region objects, which are created by functions like CreateRectRgn and ExtCreate Rgn, the pUser field is used only for the simplest case, rectangle regions, similar to solid brush. The data block pointed to by pUser for a rectangle region is 24 bytes in size. Inside the data block, the first two DWORDs have unknown meaning; the remaining 16 bytes form a RECT structure.

```
typedef struct
{
    DWORD dwUnknown1; // = 17
    DWORD dwUnknown2; // = 1, 2
    RECT rcBound;
} UserData_RectRgn;
```

As you may have guessed, handles for rectangle regions are reused in the same way as solid brushes. We did a simple experiment of creating, dumping pKernel and pUser pointers, and then deleting rectangular regions 8 times. You can see that the GDI re uses the index three times, without changing its pKernel and pUser pointers, although the rectangle coordinates do change.

Normally, the GDI only provides functions to create and then use GDI objects. Once a GDI object is created, it's cast in stone and can't be changed. In object oriented programming, such objects are called immutable objects. For example, after you create a brush, you can't directly change the color

of that brush. For region objects, the GDI breaks this rule. You can use SetRectRgn to change an existing region to a rectangular region with specific coordinates. Knowing the data structure pointed to by the pUser field, it's apparent how this could be easily implemented in the GDI. The GDI just needs to make sure the pUser field is not empty—that is, a UserData_RectRgn structure is allocated—and then sets the coordinates inside properly. So regions as GDI objects are mutable objects.

User Mode Font Data: Width Table

The Windows GDI has more data structures defined for fonts than any other GDI object. We have LOGFONT, TEXTMETRIC, PANOSE, ABC, GLYPHSET, etc. But we found no traces of those data structures in the GDI object table. The user mode data structure for font handles is very simple:

```
typedef struct
{
    DWORD dwUnknown; // = 0
    void *pCharWidthData;
} UserData_Font;
```

The first field in UserData_Font is always found to be zero. The second field is usually zero, except after calls to functions like GetCharWidth. GetCharWidth can be used to fill an integer array with the character width of characters within a certain range. After GetCharWidth is called, pCharWidthData points to a data structure, allocated from the system heap, which is mostly a cached character-width table. Once again, we see the GDI taking extra trouble to optimize performance.

Once you have a value of pCharWidthData like 0x1456b0, it's not hard to figure out where that address is. The “Locate GDI Object Table” property page displays all the regions within the user address space. From that, we know that address 0x1456b0 points to the first heap, which is the default process heap. If you double-click on the row showing the first heap, GdiHandle program will show a memory-dump window, as in [Figure 3-5](#). If you click the “Dump” button, the region will be dumped to a text file together with a list of all the heap allocation blocks within the region, if it is a heap.

User Mode Device Context Data: Keeping the Settings

Before you can draw anything using the Windows GDI, you need a handle for the device context, either by creating your own or getting it from the OS. There are two dozen attributes you can set to and retrieve from a device context. For example, mapping modes, text foreground color, background color, brush, pen, and font are commonly used device context settings. So, naturally, the GDI needs a structure to store the various settings associated with a device context. On Windows NT/2000, the pUser field in the GDI object table points to such a structure for a device context handle.

After the tedious process of changing device context settings and watching the changes in binary data, we can get a rough picture of the structure pointed to by pUser for a device context. But still the information is incomplete and unofficial. With the help of the kernel-level GDI debugger extension Microsoft provides, together with Win Dbg (Microsoft's system level source code debugger), the picture is much more complete and formal. We will go into detail about how to use the GDI debugger extension in [Section 3.7](#).

Here is what we know about this 456-byte data structure for Windows 2000 (400 bytes for Windows NT 4.0), which we have named DC_ATTR for device context attribute:

```
// dcattr.h

typedef struct
{
    ULONG ul1;
    ULONG ul2;
} FLOATOBJ;

typedef struct
{
    FLOATOBJ efM11;
    FLOATOBJ efM12;
    FLOATOBJ efM21;
    FLOATOBJ efM22;
    FLOATOBJ efDx;
    FLOATOBJ efDy;
    int fxDx;
    int fxDy;
    long flAccel;
} MATRIX;

// Windows NT 4.0: 0x190 bytes
// Windows 2000 : 0x1C8 bytes
typedef struct
{
    void *    pvLDC;           // 000
    ULONG     ulDirty;
    HBRUSH    hbrush;
    HPEN      hpen;

    COLORREF  crBackgroundClr; // 010
    ULONG     ulBackgroundClr;
    COLORREF  crForegroundClr;
```

```
ULONG    ulForegroundClr;

#if (_WIN32_WINNT >= 0x0500)
    unsigned unk020_00000000[4]; // 020
#endif
    int     iCS_CP;           // 030
    int     iGraphicsMode;
    BYTE   jROP2;            // 038
    BYTE   jBkMode;
    BYTE   jFillMode;
    BYTE   jStretchBltMode;

    POINT   ptlCurrent;      // 03C
    POINTFX ptxCurrent;     // 044
    long    IBkMode;         // 04C

    long    IFillMode;        // 050
    long    IStretchBltMode;

#endif (_WIN32_WINNT >= 0x0500)
    long    flFontMapper;     // 058
    long    llcmMode;
    unsigned hcmXform;       // 060
    HCOLORSPACE hColorSpace;
    unsigned unk068_00000000;
    unsigned lcmBrushColor;
    unsigned lcmPenColor;    // 070
    unsigned unk074_00000000;
#endif

    long    flTextAlign;      // 078
    long    ITextAlign;
    long    ITextExtra;       // 080
    long    IRelAbs;
    long    IBreakExtra;
    long    cBreak;

    HFONT   hlfntNew;         // 090
    MATRIX  mxWorldToDevice; // 094
    MATRIX  mxDeviceToWorld; // 0D0
    MATRIX  mxWorldToPage;   // 10C

    unsigned unk048_00000000[8]; // 148

    int     iMapMode;         // 168
```

```
#if (_WIN32_WINNT >= 0x0500)
    DWORD    dwLayout;          // 16c
    long     lWindowOrgx;       // 170
#endif
    POINT    ptlWindowOrg;      // 174
    SIZE     szlWindowExt;      // 17c
    POINT    ptlViewportOrg;    // 184
    SIZE     szlViewportExt;    // 18c

    long     flXform;          // 194
    SIZE     szlVirtualDevicePixel; // 198
    SIZE     szlVirtualDeviceMm; // 1a0
    POINT    ptlBrushOrigin;    // 1a8

    unsigned unk1b0_0000000[2]; // 1b0
    unsigned RectRegionFlag;   // 1b4
    RECT    VisRectRegion;     // 1b8
} DC_ATTR;
```

Most of the fields in a DC_ATTR structure are quite intuitive. For GDI objects like brushes, pens, and fonts, their handles will be stored in it when they are selected into a DC. There is no trace of device-dependent bitmaps, palette, and regions. Scalar attributes like text color, background color, graphics mode, binary raster operation, stretch bitbltng mode, text justification, and map mode get stored in this structure, too. Some attributes like text color and text alignment get two copies of their value stored for unknown reasons.

Windows NT/2000 GDI supports world coordinate transformation between logical coordinate space and device coordinate space. World transformation allows for translation, scaling, rotation, and shearing. It's described by a 2-by-3 floating-point matrix XFORM, which is set by SetWorldTransform. XFORM is not stored in floating point in this DC_ATTR data structure. XFORM has six single-precision floating-point numbers which define a two-dimensional linear transformation. We know the standard IEEE single-precision floating-point number is 4 bytes long, while the double-precision floating-point number is 8 bytes long. But XFORM's representation in DC_ATTR uses neither of these two floating-point formats. XFORM is represented by a MATRIX structure which has six pairs of DWORD and three 32-bit values. The closest data type you can find for these pairs of DWORDs is the FLOATOBJ structure defined in WinNT DDK.

An IEEE single-precision floating-point number occupies 32 bits. It has a 1-bit sign, an 8-bit exponent biased at 127, plus a 23-bit significand with one hidden bit, which is always 1.

The real number represented would be: sign * 2^(exponent-127) * (1<<24 + significand)/2^24.

For example, number 1.0 is stored as 0x3F800000, which has a positive sign, 127 as exponent, and significands being all zero. According to the definition, we have:

$$1 * 2^{(127-127)} * (1<<24+0)/2^{24} = 1$$

What Microsoft wants here is to simulate floating-point calculation using integer arithmetic with “high performance” and precision. Having high precision means a long significand, while performance can be achieved by designing a format which is easy to take apart in doing computation. Microsoft uses a FLOATOBJ structure to represent a floating-point number for GDI. A FLOATOBJ is divided into two 32-bit numbers; the higher DWORD (ul2) is the exponent, the lower DWORD (ul1) is the sign plus significand. There is no hidden bit or bias, as in IEEE format.

To convert a FLOATOBJ structure back to a floating-point number is very simple:

```
double FLOATOBJ2Double(const FLOATOBJ & f)
{
    return (double) f.ul1 * pow(2, (double)f.ul2-32);
}
```

For example, number 1.0's FLOATOBJ representation will have ul2 being 2, and ul1 being 0x40000000, which can be converted back to $2^{30} \cdot 2^{(2-32)} = 1$.

Besides this FLOATOBJ representation of XFORM, the GDI also stores a rounded version of translation values (eDx and eDy) in their integer forms XFORM_eDxi and XFORM_eDyi.

There are still lots of fields in the DC_ATTR to which we do not have a clue. At the same time, quite a few functions on device context do not change DC_ATTR in any noticeable way; for example, SelectPalette, SetMitterLimit, and SetArtDirection seem to have no effect on DC_ATTR structure. We will see later that the DC_ATTR is just part of the complicated data structure that the GDI keeps for a device context. The kernel portion data structure of a device context has a mirror copy of the DC_ATTR structure plus much more information, especially information regarding the device driver supporting the device context.

To summarize, in this section we've looked at the user mode accessible data structures for solid brush, rectangle region, font, and device context GDI objects. Those data structures are quite simple, mostly designed to optimize GDI performance by reducing the cases where the GDI needs to switch back and forth between user mode and kernel mode execution privileges. To understand more about internal GDI data structure, we have to switch our attention to the remaining kernel mode data structure.

Note

For the Intel CPU, real floating-point operations and MMX instructions are not recommended for kernel mode components, because the CPU's floating-point and MMX operation state is not properly saved during task switching. This is one reason

why the GDI represents floating-point numbers using two 32-bit integers. Another reason is certainly performance on earlier machines with slow or no hardware floating-point support. The Windows NT/2000 graphics engine supports dozens of floating-point enumeration routines—for example, FLOATOBJ_Add, FLOATOBJ_GreaterThan, etc. With newer Intel CPUs, floating-point operations could be as fast as integer operations, except that converting a floating point to an integer could be slow. Windows 2000 provides a new pair of functions to allow the driver to save the floating-point state and use hardware floating-point/MMX instructions in kernel mode.

[< BACK](#) [NEXT >](#)

3.6 ACCESSING KERNEL MODE ADDRESS SPACE

The first step in our quest to understand the GDI kernel mode data structure is to be able to read data in kernel mode address space from a user mode program like GdiHandle. On Windows NT/2000, each process has 4 GB of address space. But only the lower 2 GB is accessible from a user mode program; the higher 2 GB is not readable, writeable, or executable from the user mode program. Any attempt to access this higher 2 GB of address space directly from a user mode program will generate a hardware protection fault.

Even Microsoft Visual C++ debugger is a user mode program. That's why you can't use it to display data at address 0xE1245680, for example, or step into kernel mode DLLs like the Win32k.sys. More powerful debuggers, such as Numega's SoftIce/W, are supported by kernel mode drivers. If you're using SoftIce/W, you may have noticed that when you start SoftIce/W manually, a DOS window shows briefly with a single command: **net start ntice**. The command loads a kernel mode DLL ntice.sys into kernel address space and starts a new device, which the user mode portion of SoftIce/W can talk to.

Kernel mode drivers are special DLLs conforming to a set of rules. For example, the Win32 API cannot be called from kernel mode drivers, because their entry points are in user mode address space. Kernel mode drivers are loaded into kernel address space, where code can access the full 4 GB of address space.

Most of Windows NT's device driver handles input/output operations which can be modeled after file operations. To access these drivers using the Win32 API, you just need to call CreateFile, ReadFile, WriteFile, and the lesser-known DeviceIoControl. For example, you can use these file operations to access serial communication drivers, parallel communication drivers, and file system drivers. Win32 also provides a set of functions to load, start, stop, or close device drivers through the service API.

The beauty of this file-operation-based design is that a device driver does not have to correspond to a real hardware device like the parallel port or USB (universal serial bus). You can create an imaginary device, write a device driver for it, use the Win32 service API to install it, and then use Win32 file operation to talk to it. The Windows NT/2000 device driver architecture really opens the door for lots of clever applications which can't be achieved solely in the Win32 level of programming.

For this stage of our GDI exploration, what we want is simply to be able to read from kernel address space. We can treat the 2-GB kernel address space as a RAM disk and design a driver to allow us to randomly read a block of memory from it and send it to user mode applications.

A Windows NT/2000 kernel I/O device driver normally has a single entry point, DriverEntry, which will be called when the driver is loaded.

```
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver,
                     IN PUNICODE_STRING RegistryPath)
```

The DriverEntry routine has the same role as _DlMainStartCRTStartup, the user mode DLL entry point. But unlike user mode DLLs, kernel mode drivers do not normally export functions. Instead, DriverEntry is responsible to report to the system the addresses of functions which need to be exposed to the system, using the DRIVER_OBJECT structure. A simple I/O driver can choose to implement just a minimum subset of functions. For example, the following four lines of code in DriverEntry expose two functions. DrvUnload is to be called when the driver unloads. DrvDispatch is to be called for creating, closing, and calling DeviceIOControl.

```
Driver->DriverUnload      = DrvUnload;
Driver->MajorFunction[IRP_MJ_CREATE] = DrvDispatch;
Driver->MajorFunction[IRP_MJ_CLOSE]  = DrvDispatch;
Driver->MajorFunction[IRP_MJ_DEVICE_CONTROL] = DrvDispatch;
```

To achieve our goal of accessing the kernel mode address space from the user mode code, we need a simple device driver here, namely Periscope. The main function of Periscope is to handle a single DeviceIoControl request. The input parameter to DeviceIoControl tells Periscope the starting address to read from and the amount of data to read. Periscope reads the data in the kernel mode and stores them in an output buffer which can then be accessed in the user mode when DeviceIoControl returns. Here is the full source code for Periscope, our spying eye in the kernel.

```
// periscope.cpp
#include "kernelopt.h"
#include "periscope.h"

const WCHAR DeviceName[] = L"\Device\Periscope";
const WCHAR DeviceLink[] = L"\DosDevices\PERISCOPE";

// Process CreateFile, CloseHandle
NTSTATUS DrvCreateClose(IN PDEVICE_OBJECT DeviceObject,
                       IN PIRP      Irp)
{
    Irp->IoStatus.Information = 0;
    Irp->IoStatus.Status     = STATUS_SUCCESS;

    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return STATUS_SUCCESS;
}

// Process DeviceIoControl
```

```
NTSTATUS DrvDeviceControl(IN PDEVICE_OBJECT DeviceObject,
                           IN PIRP      Irp)

{
    NTSTATUS nStatus = STATUS_INVALID_PARAMETER;

    Irp->IoStatus.Information = 0;

    // Get a pointer to the current location in the Irp,
    // where the function codes and parameters are located.
    PIO_STACK_LOCATION irpStack =
        IoGetCurrentIrpStackLocation (Irp);

    unsigned * ioBuffer = (unsigned *)
        Irp->AssociatedIrp.SystemBuffer;
    if ( (irpStack->Parameters.DeviceloControl.IoControlCode
          == IOCTL_PERISCOPE) && (ioBuffer!=NULL) &&
        (irpStack->Parameters.DeviceloControl.
         InputBufferLength >= 8) )
    {
        unsigned leng = ioBuffer[1];

        if ( irpStack->Parameters.DeviceloControl.
            OutputBufferLength >= leng )
        {
            Irp->IoStatus.Information = leng;
            nStatus = STATUS_SUCCESS;

            __try
            {
                memcpy(ioBuffer, (void *) ioBuffer[0], leng);
            }
            __except ( EXCEPTION_EXECUTE_HANDLER )
            {
                Irp->IoStatus.Information = 0;
                nStatus = STATUS_INVALID_PARAMETER;
            }
        }
    }

    Irp->IoStatus.Status = nStatus;
    IoCompleteRequest(Irp, IO_NO_INCREMENT);

    return nStatus;
}
```

```
// Process driver unloading
void DrvUnload(IN PDRIVER_OBJECT DriverObject)
{
    UNICODE_STRING deviceLinkUnicodeString;

    RtlInitUnicodeString(&deviceLinkUnicodeString,
        DeviceLink);

    IoDeleteSymbolicLink(&deviceLinkUnicodeString);
    IoDeleteDevice(DriverObject->DeviceObject);
}

// Installable driver initialization entry point.
NTSTATUS DriverEntry(IN PDRIVER_OBJECT Driver,
    IN PUNICODE_STRING RegistryPath)
{
    UNICODE_STRING deviceNameUnicodeString;

    RtlInitUnicodeString( &deviceNameUnicodeString,
        DeviceName );

    // Create a device
    PDEVICE_OBJECT deviceObject = NULL;

    NTSTATUS ntStatus = IoCreateDevice (Driver,
        sizeof(CDeviceExtension), & deviceNameUnicodeString,
        FILE_DEVICE_PERISCOPE, 0, TRUE, & deviceObject);

    if ( NT_SUCCESS(ntStatus) )
    {
        // Create a symbolic link that Win32 apps can specify
        // to gain access to this driver/device
        UNICODE_STRING deviceLinkUnicodeString;

        RtlInitUnicodeString (&deviceLinkUnicodeString,
            DeviceLink);

        ntStatus = IoCreateSymbolicLink(
            &deviceLinkUnicodeString,
            &deviceNameUnicodeString);

        // Create driver dispatch table
        if ( NT_SUCCESS(ntStatus) )
        {
            Driver->DriverUnload      = DrvUnload;
            Driver->MajorFunction[IRP_MJ_CREATE] =

```

```
DrvCreateClose;
Driver->MajorFunction[IRP_MJ_CLOSE] =
    DrvCreateClose;
Driver->MajorFunction[IRP_MJ_DEVICE_CONTROL] =
    DrvDeviceControl;
}

}

if ( !NT_SUCCESS(ntStatus) && deviceObject!=NULL )
    IoDeleteDevice(deviceObject);

return ntStatus;
}
```

Most of the code belongs to the basic skeleton kernel mode driver. Every device a device driver is supposed to support must have a name; ours is “Periscope.” It will be put into the “Device” directory of Windows object name space. DDK provides a little tool, OBJDIR, to let you list, among other things, device drivers installed on your system. The DrvCreateClose routine handles creation and closing of instantiations of the device, which starts from the Win32 API call CreateFile and CloseHandle. The Drv DeviceControl routine does the main job of Periscope, reading a block of memory when a user application calls DeviceIoControl. The only interesting part of our task is a few lines in the DrvDeviceControl routine. After making sure the IO control code is valid, the IO buffer is not empty, and the input parameter is at least 8 bytes long (starting address and length), the code retrieves the starting address to read from and the size of the data to read, then simply copies the data requested to the output buffer. Note that the actual reading is protected by an exception handling, just in case some addresses are invalid. The Drv Unload routine handles driver unload, and finally Driver Entry is the main entrance to the driver.

To compile the code into a valid kernel mode driver, quite a few compiler and linker options generated by the development studio need to be changed. For example, you need to tell the compiler to use the `__stdcall` calling convention, instead of the default `__cdecl` calling convention. The Windows subsystem flag in the driver needs to be set to “native,4.00”, instead of the Windows GUI. The changes are enforced by adding settings in the project file and the header file `kernelopt.h`. With the right settings, we can manage to let the development studio generate a kernel mode driver.

Periscope will be compiled into a tiny kernel mode DLL, `Periscope.sys`. Subsequent programs using it assume it's copied into the root directory of your C: drive. It's designed and tested for both Windows NT 4.0 and Windows 2000.

Dynamic loading, starting, stopping, and unloading of a kernel mode device driver are well supported by the Win32 service API. We defined a C++ class `KDevice` to handle the job. The constructor of `KDevice` uses `OpenSCManager` call to establish a connection to the service control manager. The public function `KDevice::Load` loads a driver using `CreateService`, starts the driver using `StartService`, and then uses `CreateFile` to get a handle of device object. It uses “`\.\Periscope`” as the file name to call `CreateFile`, a Windows convention to open a device. After that, you can use `Device IoControl` on the handle to talk to the Periscope driver living in kernel mode.

KDevice is a generic class handling the Windows NT/2000 device drivers. It could be used to handle other device drivers. The class is actually quite simple, so instead of showing the source code of KDevice here, we jump directly to a small test program using it to access the Periscope driver.

```
// TestPeriscope.cpp
#define STRICT
#include <windows.h>
#include <winioc.h>
#include <assert.h>

#include "device.h"
#include "..\Periscope\Periscope.h"

class KPeriscopeClient : public KDevice
{
public:
    KPeriscopeClient(const TCHAR * DeviceName)
        : KDevice(DeviceName)
    {
    }

    bool Read(void * dst, const void * src, unsigned len);
};

bool KPeriscopeClient::Read(void * dst, const void * src,
                           unsigned len)
{
    unsigned cmd[2] = { (unsigned) src, len };
    unsigned long dwRead;

    return IoControl(IOCTL_PERISCOPE, cmd, sizeof(cmd),
                    dst, len, &dwRead) && (dwRead==len);
}

int WINAPI WinMain(HINSTANCE, HINSTANCE, LPSTR, int)
{
    KPeriscopeClient scope("PeriScope");
    if ( scope.Load("c:\\periscope.sys") == ERROR_SUCCESS )
    {
        unsigned char buf[256];

        scope.Read(buf, (void *) 0xa000004e, sizeof(buf));
    }
}
```

```
    scope.Close();

    MessageBox(NULL, buf, "Mem[0xa000004e]", MB_OK);
}
else
    MessageBox(NULL, "Unable to load c:\\periscope.sys",
               NULL, MB_OK);

return 0;
}
```

The code derives a class KPeriscopeClient from the KDevice class and adds an extra Read method which wraps around DeviceIoControl to instruct Periscope to read a memory block using a special IO control code, IOCTL_PERISCOPE. The main program allocates an instance of KPeriscopeClient on the stack, loads the kernel mode driver, and then reads 256 bytes starting from address 0xa000004e. This is where Win 32k.sys is loaded, which provides the underlying implementation for GDI32.DLL and USER32.DLL. The base address of Win32k.sys is 0xa0000000. Reading at offset 0x4e from a Win32 module's beginning normally gets the warning message which will appear in a DOS box when you execute a Windows program, “This program cannot be run in DOS mode \$”.

If this is the first time you get hold of a simple Win NT/2000 kernel mode driver source code and a trivial user mode code to access the driver, you may be tempted to step through the code to see for yourself how things really work together. Arm yourself with a kernel level debugger like SoftIce/W, load the necessary symbols or export table for system DLLs, step from WinMain in TestPeriscope.cpp all the way to Drv DeviceControl in Periscope.cpp, and you will see the calling stack shown in [Table 3-3](#).

Please note that after entering Kernel32.dll's DeviceIoControl and before reaching DrvDeviceControl of Periscope, the Windows system code is running, so you have to step in assembly code. Also note that ntdll.dll is a user mode DLL. It invokes in terruption 2Eh to switch the CPU to kernel address mode. In other words, interruption 2Eh is served by kernel mode code.

Table 3-3. Calling Stack from User Mode Program to Kernel Mode Driver

Level	Function	Module/File
1	WinMain	TestPeriscope.cpp
2	KPeriscopeClient::Read	TestPeriscope.cpp
3	KDevice::IoControl	Device.h
4	DeviceIoControl	Kernel32.dll
5	NTDeviceIoControlFile	Ntdll.dll
6	Int 2Eh	
7	NTDeviceIoControlFile	Ntoskrnl.exe
8	IofCallDriver	Ntoskrnl.exe
9	DrvDeviceControl	Periscope.cpp

[< BACK](#) [NEXT >](#)

3.7 WINDBG AND THE GDI DEBUGGER EXTENSION

Being able to access Windows kernel mode data provides the basic tool to start understanding Windows kernel data structures, but we still need the Windows kernel knowledge and expertise to guide us to where to look for information and how to decode information. The most knowledgeable entity in the world on the Windows kernel is certainly Microsoft itself. That's why we are turning to official Microsoft tools to help us understand the Windows GDI kernel data structure.

As part of the Windows Platform SDK and Windows NT/2000 DDK, Microsoft provides a powerful graphical tool that allows you to debug Win32 applications and Windows NT/2000 kernel mode drivers or examine crash dumps and blue-screen dump data. This is the Microsoft Windows System Debugger (WinDbg). Best of all, WinDbg is free.

There are several ways you can use WinDbg.

- WinDbg can be used to debug Win32 applications on a single computer, just like a normal user mode debugger—for example, Microsoft Visual C++ debugger. In this mode, you can't step into kernel mode code and can't access kernel mode data.
- WinDbg can be used to debug Win32 applications remotely, similar to Visual C++ debugger's remote debugging features. In this mode, a host computer and a target computer are linked using a null modem cable, modem or network. You can use the WinDbg user interface on the host computer to debug the program running on the target computer. You only have user mode access.
- WinDbg can also be used to debug Windows NT/2000 kernel mode code remotely, similar to SoftICE/W's kernel debugging features. In this mode, a host computer and a target computer are linked using a null modem cable. The target computer is started from a special configuration with the kernel mode debugging feature enabled. You run WinDbg on the host computer with the same serial-cable setting as the target computer to control the running of programs on the target computer. In this remote kernel debugging mode, the host computer has access to the full 4-GB address space of the target computer.

In the area of Windows kernel debugging, SoftICE/W may be much easier to use because it needs only a single machine while WinDbg needs an extra host machine. Besides, using SoftICE/W, you can step from user mode code into kernel mode code and back to user mode code easily. But WinDbg offers lots of advantages over SoftICE/W in certain areas, simply because it's an official living Microsoft tool. WinDbg is free, small, and supports different versions of Windows NT/2000, while SoftICE/W costs money and needs frequent upgrades to catch up with new releases of Windows NT/2000.

The most powerful feature of WinDbg is its modular, extensible design. A normal debugger provides a limited set of commands, which you use to access data, code, set breakpoint, control program execution, etc. WinDbg allows you to add commands to the debugger by writing debugger extension DLLs. Each extension DLL normally focuses on one particular area of the Windows operating system. WinDbg comes with extension DLLs provided by Microsoft as listed in [Table 3-4](#).

The interface between WinDbg and debugger extensions is very simple. It's fully defined in the DDK header file WDBGEXTS.h. Debugger extensions are required to export at least three functions: CheckVersion, ExtensionApiVersion, and WinDbg ExtensionDlInit, for version check and initialization. CheckVersion makes sure the OS version running on the target machine is the same as the version of the extension.

Table 3-4. Windbg Debugger Extensions Provided by Microsoft

Extension	OS Functionality Covered
Gdikdx.dll	GDI kernel mode debugger extension
Kdextx86.dll	Executive/HAL kernel mode debugger extension
Ntsdexts.dll	Default user mode debugging extension
Rpcexts.dll	Remote procedure call debugger extension
Userexts.dll	USER user mode debugger extension
Userkdx.dll	USER kernel mode debugger extension
Vdmexts.dll	NT DOS/WOW (Window in Window) debugger extension

You should not expect to get the right results when loading a FREE (release) build extension DLL to debug a CHECKED (debug) build OS. ExtensionApiVersion checks to see if the extension DLL and WinDbg host are speaking the same version of the API. WinDbg ExtensionDlInit, the most critical among the three, passes the WINDBG_EXTENSION_APIS structure from WinDbg to extension DLL. The WINDBG_EXTENSION_APIS structure currently defines 11 callback routines that extension DLLs can call. Callback routines are implemented by WinDbg with the help of image hlp DLL, debug symbol files, and the target system linked by the cable.

```
typedef struct _WINDBG_EXTENSION_APIS {
    ULONG nSize;
    PWINDBG_OUTPUT_ROUTINE IpOutputRoutine;
    PWINDBG_GET_EXPRESSION IpGetExpressionRoutine;
    PWINDBG_GET_SYMBOL IpGetSymbolRoutine;
    PWINDBG_DISASM IpDisasmRoutine;
    PWINDBG_CHECK_CONTROL_C IpCheckControlCRoutine;
    PWINDBG_READ_PROCESS_MEMORY_ROUTINE IpReadProcessMemoryRoutine;
    PWINDBG_WRITE_PROCESS_MEMORY_ROUTINE IpWriteProcessMemoryRoutine;
    PWINDBG_GET_THREAD_CONTEXT_ROUTINE IpGetThreadContextRoutine;
    PWINDBG_SET_THREAD_CONTEXT_ROUTINE IpSetThreadContextRoutine;
    PWINDBG_IOCTL_ROUTINE IpIoctlRoutine;
    PWINDBG_STACKTRACE_ROUTINE IpStackTraceRoutine;
} WINDBG_EXTENSION_APIS, *PWINDBG_EXTENSION_APIS;
```

As you can see, the extension DLLs can call the host program WinDbg to display a string, evaluate an expression, look for a symbol, disassemble code, check for abortion, read/write memory, get/set thread context, call IO control routine, and even produce stack trace. In other words, extension DLLs provide the domain knowledge on how to present certain areas of OS internal data structure to developers, while WinDbg plays the role of interfacing with the user and the system under debug.

Besides the three required functions, extra exported functions from the extension DLLs are all usable commands from the WinDbg command line. The name of an exported function is the same as the corresponding command's name. All these exported functions have the same prototype, as defined by the following macro:

```
#define DECLARE_API(s) \
```

```

CPPMOD VOID          \
s(                  \
HANDLE      hCurrentProcess, \
HANDLE      hCurrentThread, \
ULONG       dwCurrentPc, \
ULONG       dwProcessor, \
PCSTR       args      \
)

```

Since this is a book on Windows NT/2000 graphics programming, we are very interested in Gdikdx.dll, the GDI kernel debugger extension DLL. After WinDbg is set up, gdikdx.dll can be loaded by issuing the load command at the WinDbg command prompt:

> **load gdikdx.dll**

Debugger extension library [...]system32\gdikdx] loaded

All commands provided by debugger extensions start with the character “!” to differentiate them from standard WinDbg commands. Issuing the help command now shows a brief summary of the GDI debugger extension's dozens of commands. As may be expected of a mainly internal debugging aid, gdikdx's help command displays outdated information, with missing commands, missing options, and obsolete commands. For example, commands brush, cliserv, gdicall, and proxymsg are listed by help but not supported; command difi is replaced by command ifi; new commands like dbli and ddib are not mentioned. But luckily each command has a help option “-?”, from which you can get more recent command information. You can also check exported functions from gdikdx.dll to find new commands not listed by help. [Table 3-5](#) lists the GDI kernel debugger extension commands.

By now, you may be eager to order your second development machine and a null modem cable, ready to try those new cool-looking commands you never thought could have existed in the first place. I've been there and done that. Even if you can set up the host and target systems properly, link them together, and start using WinDbg on the host computer to control the target computer, using the GDI extension commands is not easy. Most of those commands need either a GDI object handle or a pointer to a certain data structure. It's lots of work to break the target system at the right place and get the object handle or pointer to feed to those commands to really make use of them.

Table 3-5. GDI Kernel Debugger Extension Command

Command	Parameters	Usage
dumphmgr	[-?]	Summary of GDI objects according to object types
dumpobj	[-?] [-p pid] [-I] [-s] object_type	All GDI objects of specific type
dumpdd		DirectDraw handle manager objects
dumpddobj	[-p pid] [type]	DirectDraw objects of specific type
dh	[-?] object handle	HMGR entry of a GDI object
dht	[-?] object handle	GDI handle type/uniqueness/index
ddib	[-?] [-I LPBITMAPINFO] [-w Width] [-h Height] [-f filename] [-b Bits] [-y Byte_Width] [-p palbits palsize] pbits	Bitmap dump
dbli	[-?] BLTINFO *	
ddc	[-?adeghrstuvx] hdc	Device context

dpdev	[-?abdfghmnprRw] ppdev	Physical device object
dldev	[-?] [-f] [-F #] ldev	Logical device object
dgdev	[-?m] dgdevptr	GRAPHICS_DEVICE
dco	[-?] clipobj	CLIPOBJ
dpo	[-?] pathobj	PATHOBJ
dppal	[-?] pal	EPALOBJ
dpw32	[-?] [process]	
dpbrush	[-?] pbrush hbrush	HBRUSH or PBRUSH
dfloat	[-?] [-l num] Value	Dump an IEEE float or float array
ebrush	[-?] pbrush hbrush	HBRUSH or PBRUSH
dps0	[-?] [-f filename] surfobj	SURFACE struct from SURFOBJ
dblt	[-?] BLTRECORD_PTR	BLTRECORD
dr	[-?] hrgn prgn	REGION
cr	[-?] hrgn prgn	Check REGION
dddsurface	[-?haruln] ddsurface	EDD_SURFACE
dddlocal	[-?ha]	EDD_DIRECTDRAW_LOCAL
dddglobal	[-?ha]	EDD_DIRECTDRAW_GLOBAL
dsprite	[-?ha]	SPRITE
dspritestate	[-?ha]	SPRITE_STATE
rgnlog	[-?] nnn [s1] [s2] [s3] [s4]	Last nnn rgnlog entries
stats	[-?]	Accumulated statistics
verifier	[-?hds]	Dump verifier information
hdc	[-?gltf] handle	
dcl	[-?] DCLEVEL *	
dca	[-?] DC_ATTR *	Dump HDC user mode data structure
ca	[-?] COLORADJUSTMENT*	
mx	[-?] MATRIX *	Dump MATRIX in DC_ADDR
la	[-?] LINEATTRS *	
ef	[-?] address [count]	
dteb	[-?] TEB	Display TEB batched commands
dpeb	[-?] [-w]	Display PEB cached objects
e	[-?] address [count]	
xo	[-?] EXFORMOBJ *	

Font Extensions

tstats	[-?] [1..50]	
gs	[-?] FD_GLYPHSET *	
gdata	[-?] GLYPHDATA *elf	[-?] LOGFONTW *
tm	[-?] TEXTMETRICW *	
tmwi	[-?] TMW_INTERNAL *	
fo	[-?acfhwxy] FONTOBJ *	
pfe	[-?] PFE *	

pff	[?-] PFF *	
pft	[?-] PFT *	
stro	[?-phe] STROBJ *	
gb	[?-hmg] GLYPHBITS *	
gdf	[?-] GLYPHDEF *	
gp	[?-] GLYPHPOS *	
cache	[?-] CACHE *	
fh	[?-] FONTHASH *	
hb	[?-] HASHBUCKET *	
fv	[?-] FILEVIEW *	
ffv	[?-] FONTFILEVIEW *	
helf	[?-] font handle	
ifi	[?-] IFIMETRICS *	
pubft	[?-]	Dump all public fonts
pvtft	[?-]	Dump all private or embedded fonts
devft	[?-]	Dump all device fonts
dispcache	[?-]	Dump glyph cache for display PDEV

Once again, we need to be more creative and adventurous in our exploration of GDI. Could we write a simple replacement for WinDbg, not for general-purpose debugging, but for the sole purpose of understanding Windows NT/2000 GDI? What we need is a simple host application for the GDI extension DLL which requires only a single machine to run and is easy to use.

Just imagine how the GDI extension's command dumphmgr is able to summarize the target machine's GDI object table on the host machine. The process should roughly be:

- WinDbg loads gdikdx.dll on the host machine at the user's request.
- When the user issues the **!dumphmgr** command, WinDbg passes the command to gdikdx.dll's exported function **dumphmgr**.
- Gdikdx's dumphmgr routine asks WinDbg to find a win32k.sys global variable which holds the address of the GDI object table in kernel address space. This is done through callback functions passed from WinDbg to Gdikdx.dll. WinDbg calls imagehelp API to get the symbol address. Remember, debugger symbol files for the target machine need to be installed on the host machine, so WinDbg has full access to the target machine's debugging symbols.
- Gdikdx asks WinDbg to read the variable on the target machine to get the pointer to the GDI object table, given the address of that variable in the target machine's address space. WinDbg sends a request to the target machine through a null modem cable, which is served by the target machine running in debug mode.
- Gdikdx asks WinDbg to read the whole GDI object table, given the GDI object table's starting address. Again, WinDbg passes the request to the target machine.
- Gdikdx summarizes the data and asks WinDbg to display the information in its window.

As a host for the GDI debugger extension gdikdx.dll, WinDbg does two things: It passes the command to gdikdx.dll and services the callback routines. Passing the command to gdikdx.dll is very easy; WinDbg just needs to pass the current process, thread handle, the CPU program counter, number of processors running on the target CPU, and the full command line to an exported function. Servicing the callback routines may look complicated, because there are 11 callback functions. But actually, gdikdx.dll uses only a few of them. The hardest within them is the routine to read process memory, which could be within kernel address space. Luckily we have Periscope, the kernel mode driver developed in the last section.

Now let's try to develop our little host program for gdikdx.dll. The program is called Fosterer. The idea behind Fosterer is quite simple. It's a program with a user interface through which the developer can issue commands. The commands are simply passed to the GDI debugger extension to be executed. When the debugger extension needs help to decode a symbol or read a memory block, it will call back to Fosterer just as it would call back to WinDbg when it's loaded there.

The program listing below shows the KHost class declaration, which provides the basic functionality used by the callback routines:

The KHost class has five member variables. Pointer pWin32k points to an instance of KImageModule class, which uses imagehlp.dll to look for symbols in Windows graphics engine win32k.sys's debug information files. The second pointer, pScope, points to an instance of KPeriscopeClient to read data from kernel mode address space. The first window handle is the handle of the main text window simulating WinDbg's output window. The second window handle is for logging extra information to monitor how the callback routines are used by gdikdx.dll. The last member variable, hProcess, is the handle of the process under investigation. The first two member functions are supporting functions; after them come five functions corresponding to five callback functions we are actually going to implement. The code list that follows shows the implementation for ExtGetExpression and ExtReadProcessMemory.

```
unsigned KHost::ExtGetExpression(const char * expr)
{
    if ( (expr==NULL) || strlen(expr)==0 )
    {
        assert(false);
        return 0;
    }

    if ( (expr[0]>='0') && (expr[0]<='9') ) // hex number
    {
        DWORD number;

        sscanf(expr, "%x", & number);
        return number;
    }

    if ( pWin32k )
    {
        const IMAGEHLP_SYMBOL * pis;

        if ( expr[0]=='&' ) // skip the first &
            pis = pWin32k->ImageGetSymbol(expr+1);
        else
            pis = pWin32k->ImageGetSymbol(expr);
        if ( pis )
        {
            Log("GetExpression(%s)=%08lx\n", expr, pis->Address);

            return pis->Address;
        }
    }

    ExtOutput("Unknown GetExpression(\"%s\")\n", expr);
    throw "Unknown Expression";

    return 0;
}
```

```
bool KHost::ExtReadProcessMemory(const void * address,
                                  unsigned * buffer, unsigned count,
                                  unsigned long * bytesread)
{
    if ( pScope )
    {
        ULONG dwRead = 0;

        if ( (unsigned) address >= 0x80000000 )
            dwRead = pScope->Read(buffer, address, count);
        else
            ReadProcessMemory(hProcess, address, buffer,
                               count, & dwRead);

        if ( bytesread )
            * bytesread = dwRead;

        if ( (unsigned) address >= 0x80000000 )
            Log("ReadKRam(%08x, %d)=", address, count);
        else
            Log("ReadURam(%08x, %08x, %d)=", hProcess,
                address, count);

        int len = min(4, count/4);

        for (int i=0; i<len; i++)
            Log("%08x ", buffer[i]);

        Log("\n");
        return dwRead == count;
    }
    else
    {
        assert(false);
        return false;
    }
}
```

The KHost::ExtGetExpression routine accepts a symbolic expression represented as a character string and is supposed to evaluate it to a numeric value. It first tries to decode the expression as a hexadecimal number, if possible. Hexadecimal numbers are commonly used by WinDbg to represent handles and addresses. Failing the first attempt, it assumes it's a symbol, so it calls pWin32k->ImageGetSymbol to look for the address of the symbol, which is normally something like win32k!gcMaxHmgr. pWin 32k points to a KImageModule object which is preloaded with debug symbols for win32k.sys. Routine KImageModule::ImageGetSymbol, which is not shown here, calls imagehlp routine SymGetSymFromName to translate a symbol name to its address.

One interesting thing is that SymGetSymFromName accepts a nonconstant char pointer as a parameter, while ExtGetExpression only accepts a constant char pointer as a parameter. You would guess just casting constant pointer to nonconstant pointer would trick the compiler and make things work. Not in this case;

SymGetSymFromName will fail and report an access violation if you do that. Both sides really mean what they tell you. ExtGetExpression is called from gdikdx.dll, which may be compiled with the Visual C++ compiler string pooling compiler option, moving strings into the read-only section. So the strings passed to ExtGetExpression are read-only. SymGetSym From Name searches for the “!” character, for which it is supposed to separate the module name from the function name. If the “!” character is found, it is replaced with a zero, to zero-terminate the module name. This generates a CPU access violation error. To solve the problem, ImageGetSymbol copies the parameter to a local variable before calling SymGetSymFromName.

The KHost::ReadProcessMemory routine is responsible for reading a memory block. It first checks if the address lies within kernel space; if so, it uses the KPeri ScopeClient class shown in the last section, which uses our little kernel mode driver Periscope.sys; if not, it simply calls Win32 API ReadProcessMemory with the process handle. Note that with the right process handle, ReadProcessMemory is able to read from the other process's user mode address space.

But KHost is a C++ class, while the WinDbg debugger extension API is defined using C features only. We need some extra glue to link them together. Here is part of the remaining code:

```
KHost theHost;
```

```
VOID WDBGAPI ExtOutputRoutine(PCSTR format, ...)
```

```
{  
    va_list ap;  
    va_start(ap, format);
```

```
    theHost.WndOutput(theHost.hwndOutput, format, ap);
```

```
    va_end(ap);  
}
```

```
ULONG WDBGAPI ExtGetExpression(PCSTR expr)
```

```
{  
    return theHost.ExtGetExpression(expr);  
}
```

```
VOID WDBGAPI ExtGetSymbol(PVOID offset, PUCHAR pchBuffer,  
    PULONG pDisplacement)
```

```
{  
    throw "GetSymbol not implemented";  
}
```

```
ULONG WDBGAPI ExtReadProcessMemory(ULONG address,  
    PVOID buffer, ULONG count, PULONG bytesread)
```

```
{  
    return theHost.ExtReadProcessMemory(  
        (const void *)address, (unsigned *)buffer, count,  
        bytesread);  
}
```

...

```
WINDBG_EXTENSION_APIS ExtensionAPI =
{
    sizeof(WINDBG_EXTENSION_APIS),
    ExtOutputRoutine,
    ExtGetExpression,
    ExtGetSymbol,
    ExtDisAsm,
    ExtCheckControl_C,
    ExtReadProcessMemory,
    ExtWriteProcessMemory,
    ExtGetThreadContext,
    ExtSetThreadContext,
    ExtIOCTL,
    ExtStackTrace
};
```

We need to fill the WINDBG_EXTENSION_APIS structure as required to interface with a debugger extension, which needs 11 callback functions. Five of these calls map to functions of KHost through a global instance of it, theHost. The rest of them just throw exceptions, which will be caught by the main program, if not by gdikdx.dll.

The whole Fosterer program is much more than what's shown here. But it's a quite standard, simple Windows program. The main program creates several child windows, one for entering the process handle, one for entering the command; still another is the main output window. An extra popup window is created for extra logging information. The main program is responsible for loading the Periscope kernel mode driver, debug symbols for win32k.sys, and most importantly WinDbg debugger extension gdikdx.dll. It initializes gdikdx.dll with the callback function table and checks to see if the current OS version is compatible with that of gdikdx.dll. There is a very interesting GDI debugger extension command, "dumphmgr," we would like to try first. It's supposed to display a summary of GDI handle manager—that is, the GDI object table we have been chasing so far in this chapter. If everything is set up right, enter the "dumphmgr" command in the command window, click the "Do" button, close your eyes, and make a wild guess on what you will see.

It is real! You are using gdikdx.dll without WinDbg, without two machines, without running OS in debug mode, without a null modem cable, and we just got a summary report on the contents of the GDI object table from the kernel address space. Best of all, we don't need to know anything about the GDI object table to make Fosterer work, because the GDI debugger extension is providing all the domain expertise. [Figure 3-7](#) shows Fosterer program's screen display.

Figure 3-7. Using Fosterer to host the WinDbg debugger extension.

The screenshot shows a Windows application window titled "Fosterer". The menu bar includes "File", "Help", and "Command". Below the menu is a toolbar with buttons for "20c", "Do", and "dumphmgr". The main window displays the output of the "dumphmgr" command. The output starts with status messages about Periscope loading and symbol files. It then checks if the Extension DLL matches the target system. Following this, it provides handle statistics: "Max handles out so far 1130", "Total Hmgr: Reserved memory 2097152 Committed 36864", and "ullLoop=1130 gcMaxHmgr=1130 handles. (objects)". A detailed table follows, listing various GDI object types with their current, maximum, allocated, and lock-aside counts, along with LAB (Logical Address Block) counts. The table ends with totals for unused and unknown objects. The window has standard Windows-style borders and a title bar.

TYPE	current	maximum	allocated	LockAside	LAB	Cur	LAB	Max
DEF_TYPE	132,	0 -	0,	0 -	0 -	0	0	0
DC_TYPE	102,	0 -	0,	0 -	0 -	0	0	0
RGN_TYPE	37,	0 -	0,	0 -	0 -	0	0	0
SURF_TYPE	492,	0 -	0,	0 -	0 -	0	0	0
CLIOBJ_TYPE	3,	0 -	0,	0 -	0 -	0	0	0
PAL_TYPE	34,	0 -	0,	0 -	0 -	0	0	0
ICMLCS_TYPE	1,	0 -	0,	0 -	0 -	0	0	0
LFONT_TYPE	95,	0 -	0,	0 -	0 -	0	0	0
PFE_TYPE	102,	0 -	0,	0 -	0 -	0	0	0
BRUSH_TYPE	132,	0 -	0,	0 -	0 -	0	0	0
TOTALS	998,	0 -	0,	0 -	0 -	0	0	0

cUnused objects 132
cUnknown objects 0 0

In [Figure 3-7](#), the small window on the left shows the process identifier; the window on the top right is the command-line window. A command is sent to the GDI debugger extension when the “Do” button is pressed. The main window shows the output from the program itself and the GDI debugger extension. The first few lines show the status of loading the Periscope kernel mode driver, the debug symbol file for the graphics engine, and the GDI debugger extension. Each debugger extension is built together with the whole Windows OS, so it has a build number. A check will be made to see that the build number of the debugger extension matches that of the OS, and a warning message will be displayed if not. It's hard to find the exact match between them. But it's essential to find a match as close as possible.

After the status information, we see the output from the “dumphmgr” command. It shows that the GDI handle manager (the piece of code responsible for the GDI object table) has reserved 2 MB of memory (in kernel address space), but only 36 KB is committed. Of the maximum 16,384 GDI handles allowed, only 1130 have been used so far during peek time since the last reboot of the machine. When the “dump hmgr” command was issued, only 998 GDI objects were actually being used, the remaining 132 were unused (objects created and then deleted). The breakdown of the GDI objects shows the number of device contexts, bitmaps, palette, logical fonts, brushes, etc., that are actually present in the system.

This is only a glimpse of what the GDI debugger extension can do. It will prove to be a vital tool to understand GDI internal data structure.

3.8 GDI KERNEL MODE DATA STRUCTURE

With the help of the kernel mode device driver Periscope, the WinDbg debugger extension DLL gdikdx.dll, and Fosterer, our simple host for the debugger extension, we are finally ready to explore the undocumented Windows NT/2000 GDI kernel mode data structure.

GDI Object Handle Table in the GDI Engine

We know from previous sections that each Win32 process has access to a table of GDI objects. The table is read-only from user processes. The table may be mapped to different addresses in different processes. GDI provides an undocumented function, the GdiQueryTable, to return the address of the GDI object table in the current user process.

The GDI object table is actually managed by the GDI graphics engine win32k.sys. It's readable and writeable from kernel address space from a fixed address. The table is mapped to user address space for each process requiring GDI. The code managing the GDI handle table is called handle manager. That's why you see so many "hmgr" when dealing with GDI internals.

According to log information generated by Fosterer, Win32k.sys keeps a global variable "gpentHmgr" which points to the start of the GDI object table in kernel address space. The GDI object table is designed to support a maximum of 16,384 GDI objects. Normally, the table is not fully used, so lots of entries in the table would be empty. Win32k.sys keeps another variable "gcMaxHmgr" to be the highest index in the table used so far.

```
GdiTableCell * gpentHmgr;  
unsigned long gcMaxHmgr;
```

Each entry in the GDI object table is a 16-bit structure we named GdiTableCell in [Section 3.4](#).

GDI Object Types in the GDI Engine

We are quite familiar with GDI objects like pen, brush, font, region, palette, etc. But the GDI handle table holds many more kinds of objects than those exposed in Win32 API. [Table 3-6](#) is a list of GDI object types shown by the dumphmgr command.

Of around 30 GDI object types listed in [Table 3-6](#), only a few are familiar to Win32 programmers—for example, DC_TYPE, BRUSH_TYPE, and LFONT_TYPE, which correspond to device context, brush/pen, and logical font. It's interesting to note that brushes and pens are classified as a single type, BRUSH_TYPE, although their type identifiers are a little different. The

Win32 API does not provide any functions to create PATH_TYPE objects directly, although you may have always suspected that some object must be created. We call BeginPath to start forming a path.

With the help of the GDI kernel extension, we will examine the GDI objects' kernel data structures.

Device Context in the GDI Engine

The device context is a major GDI object which holds lots of settings for the Win32 API to interact with a graphics device, whether a display card, a printer, a plotter, or an image setter. The GDI stores data for a device context in two places. There is a user mode structure, named DC_ATTR, which stores settings like current pen, current brush, background and foreground color. DC_ATTR structure is shown in [Section 4.3](#). The GDI engine also keeps a structure, DCOBJ, in kernel mode address space which holds complete information for a device context object, including a duplicate of DC_ATTR. For a device context handle, the pKernel field in its GDI object-table entry points to an instance of DCOBJ, while the pUser field points to an instance of DC_ATTR.

Table 3-6. Master List of GDI Object Types

TYPE	Type ID	
DEF_TYPE	0x00	Deleted GDI objects
DC_TYPE	0x01, 0x21	Device context, meta file
DD_DRAW_TYPE	0x02	DirectDraw object, managed separately now
DD_SURF_TYPE	0x03	DirectDraw surface, managed separately now
RGN_TYPE	0x04	Region
SURF_TYPE	0x05	Device-dependent bitmap
CLIOBJ_TYPE	0x06	Client object
PATH_TYPE	0x07	Path
PAL_TYPE	0x08	Palette
ICMCS_TYPE	0x09	
LFONT_TYPE	0x0a	Logical font
RFONT_TYPE	0x0b	
PFE_TYPE	0x0c	
PFT_TYPE	0x0d	
ICMCXF_TYPE	0x0e	
ICMDLL_TYPE	0x0f	
BRUSH_TYPE	0x10, 0x30	Brush, pen
D3D_HANDLE_TYPE	0x11	
DD_VPORT_TYPE	0x12	
SPACE_TYPE	0x13	
DD_MOTION_TYPE	0x14	
META_TYPE	0x15	
EFSTATE_TYPE	0x16	
BMFD_TYPE	0x17	
VTFD_TYPE	0x18	
TTFD_TYPE	0x19	
RC_TYPE	0x1a	
TEMP_TYPE	0x1b	
DRVOBJ_TYPE	0x1c	
DCIOBJ_TYPE	0x1d	
SPOOL_TYPE	0x1e	

The GDI kernel extension provides several commands to decode the data structure behind a device context handle. Command “ddc” decodes a HDC, mainly the DCOBJ structure; command “dcl”

shows DCLEVEL structure within a DCOBJ structure; command “dca” shows DC_ATTR structure, which is in both kernel and user address space. Here is what we know about them.

```
// dcobj.h

// Windows 2000, 440(0x1B8) bytes
typedef struct
{
    HPALETTE      hpal;
    void *        ppal;
    void *        pColorSpace;
    unsigned      llcmMode;
    unsigned      lSaveDepth;
    unsigned      unk1_00000000;
    HGDIOBJ      hdcSave;
    unsigned      unk2_00000000[2];
    void *        pbrFill;
    void *        pbrLine;
    void *        unk3_e1a28d88;
    HGDIOBJ      hpath; // HPATH
    unsigned      flPath; // PathFlags
    LINEATTRS     lapath; // 0x20 bytes
    void *        prgnClip;
    void *        prgnMeta;
    COLORADJUSTMENT ca; // 0x18 bytes
    unsigned      flFontState;
    unsigned      ufi;
    unsigned      unk4_00000000[12];
    unsigned      fl;
    unsigned      flbrush;
    MATRIX        mxWorldToDevice;
    MATRIX        mxDeviceToWorld;
    MATRIX        mxWorldToPage;
    FLOATOBJ      efM11PtoD;
    FLOATOBJ      efM22PtoD;
    FLOATOBJ      efDxPtoD;
    FLOATOBJ      efDyPtoD;
    FLOATOBJ      efM11_TWIPS;
    FLOATOBJ      efM22_TWIPS;
    FLOATOBJ      efPr11;
    FLOATOBJ      efPr22;
    void *        pSurface;
    SIZE          sizl;
} DCLEVEL;
```

```
// Windows 2000, 1548(0x60C) bytes
typedef struct
{
    HGDIOBJ    hHmgr;      // 000
    void *     pEntry;     // 004
    ULONG      cExcLock;   // 008
    ULONG      Tid;        // 00c

    DHPDEV     dhpdev;     // 0x010
    unsigned    dtype;
    unsigned    fs;         // Flags
    void *     ppdev;
    void *     hsem;       // 0x020
    unsigned    flGraphics;
    unsigned    flGraphics2;
    void *     pdcattr;    // point to user mode DCATTR
    DCLEVEL    dcLevel;    // 0x030 0x1B8(440) bytes
    DC_ATTR    dcAttr;     // 0x1C8(456) bytes
    unsigned    hdcNext;    // 0x3B0
    unsigned    hdcPrev;
    RECTL     erclClip;
    unsigned    unk4_00000000[2];
    RECTL     erclWindow;
    RECTL     erclBounds;
    unsigned    unk5_00000000[4];
    void *     prgnAPI;
    void *     prgnVis;
    void *     prgnRao;
    POINT     FillOrigin;
    unsigned    unk6_00000000[10];
    void *     pca1;       // point to DCLEVEL.ca
    unsigned    unk7_00000000[20];
    void *     pca2;
    unsigned    unk8_00000000[20];
    void *     pca3;
    unsigned    unk9_00000000[20];
    void *     pca4;
    unsigned    unka_00000000[10];
    HFONT     hlfntCur;
    unsigned    unkfb_00000000[2];
    void *     prfnt;
    unsigned    unkc_00000000[33];
    unsigned    unkd_0000ffff;
    unsigned    unke_ffffffff;
```

```
unsigned unkf_00000000[3];  
} DCOBJ;
```

The last time the Windows programming community saw a detailed description of something like DCOBJ was in *Undocumented Windows*, by Schulman, Maxey, and Pietrek, published in 1992. This book continues to help us understand some fields inherited from the Windows 3.0/3.1 period, both decoded and not decoded by the GDI debugger extension's commands.

For every GDI object, its kernel data starts with a 16-byte structure. The first field stores the GDI handle for that object; the second field is an unknown pointer; the third field is a lock count; and the last pointer is the creating thread's thread identifier. With the GDI handle stored in kernel mode, the GDI engine can access the GDI object table to find out which process it belongs to and related user mode data structures like DC_ATTR.

The first field after the header is dhpdev, a handle to the PDEV structure managed by a graphics device driver. A graphics device driver is supposed to manage several physical devices, which could be different printers, printers linked to different ports, or two printing jobs. To do that, the device driver is free to define a data structure needed to manage such physical devices. The Windows DDI document calls those data structures PDEV (physical device), which is actually only defined and used by a device driver. The GDI engine calls the driver's DrvEnablePDEV entry point to let the device driver allocate and initialize a PDEV structure. Because the PDEV structure is solely managed by the device driver, the GDI engine is not interested in any details about it. So the DDI (device driver interface) allows DrvEnablePDEV to return a handle to PDEV, instead of a pointer to PDEV. In this way, the GDI engine is playing a fair game to allow the driver vendor to hide its implementation using a handle, just as the GDI is hiding its implementation using the GDI handles. The handle returned by DrvEnablePDEV will be used by the GDI engine in subsequent calls to the physical device to create a drawing surface. The GDI engine finally calls DrvDisablePDEV to let the device driver free any memory and resource used by the physical device.

The Windows NT/2000 DDK source-code directory contains several sample display drivers' source code, each having a different PDEV structure. DrvEnable PDEV normally just returns the pointer to PDEV as its handle.

There are different types of device contexts, as we know from the Win32 API. Within DCOBJ, their differences can be indicated by dtype field. There seem to be only three types of device contexts:

```
typedef enum  
{  
    DCTYPE_DIRECT = 0, // normal device context  
    DCTYPE_MEMORY = 1, // memory device context  
    DCTYPE_INFO   = 2 // information context  
};
```

DCOBJ stores a few flags regarding the device context in the fs field. Here are some of the flags

shown by the GDI extension:

```
typedef enum
{
    DC_DISPLAY      = 0x0001,
    DC_DIRECT       = 0x0002,
    DC_CANCELLED    = 0x0004,
    DC_PERMANENT     = 0x0008,
    DC_DIRTY_RAO     = 0x0010,
    DC_ACCUM_WMGR    = 0x0020,
    DC_ACCUM_APP      = 0x0040,
    DC_RESET         = 0x0080,
    DC_SYNCHRONIZEACCESS = 0x0100,
    DC_EPSPRINTINGESCAPE = 0x0200,
    DC_TEMPINFODC    = 0x0400,
    DC_FULLSCREEN      = 0x0800,
    DC_IN_CLONEPDEV    = 0x1000,
    DC_REDIRECTION     = 0x2000
} DCFLAGS;
```

The next field in DCOBJ is named ppdev by the GDI extension. So naturally we would guess it means pointer to a physical device. The GDI extension even provides a command **dpdev** to decode a PDEV pointer. But, according to the DDK, the physical device data structure is managed by an individual device driver, which the GDI engine is not supposed to know about. The DrvEnablePDEV entry point from the driver returns a handle to the physical device instead of a pointer to it. One explanation is that the GDI engine creates its own data structure for a physical device, which we may call PDEV_WIN32K to avoid confusion with the device driver's PDEV. The PDEV_WIN32K structure is a very complicated structure. We will discuss it in the next subsection with more details.

The hsem field points to a semaphore structure, apparently used to synchronize certain access to fields.

The flGraphics and flGraphics2 fields store flags about the capabilities of the device. Those flags are documented in the DDK; among them are GCAPS_ALTERNATEFILL, GCAPS_WINDINGFILL, GCAPS_COLOR_DITHER, etc. Flags fl Graphics and flGraphics2 are from the DEVINFO structure, which is filled by the device driver's DrvEnablePDEV function.

Field pdcattr points to the DC_ATTR structure for that device context in user mode address space, which contains most device context settings. The DCOBJ structure has a duplication of it in its dcAttr field. It's possible the GDI designers want setting DC attributes to be efficient, with as little kernel mode code involvement as possible. This requires a DC_ATTR structure in user mode. But they also want to be able to access it easily in kernel mode, which requires a copy of DC_ATTR in kernel mode. Certain flags may be used to synchronize the two copies of DC_ATTR.

When trying to understand the DC_ATTR structure, we found that certain functions on the device context handle have no impact on it—for example, selecting an HBIT MAP into a memory DC, or selecting a palette into a DC. If you ever wondered where those settings are stored, they are in the DCLEVEL structure, which is contained in DCOBJ. DCLEVEL stores information about palette, color space, color adjustment, line attribute, clipping region, transformation, path, etc.

GDI Engine PDEV Structure

The initial entry point exposed by a graphical driver is DrvEnableDriver. When a driver is loaded, the GDI engine calls its DrvEnableDriver entry point, which fills a DRV_ENABLEDATA structure. DrvEnableDriver basically gives the GDI engine a table of functions it implements, telling the GDI engine what functions are supported by the driver. DirectDraw also defines several callback function table structures. The GDI engine certainly needs some structure to keep this per-graphic-driver information. Here is the GDI engine PDEV structure:

```
// Windows 2000 3304 (0xCE8) bytes
typedef struct
{
    unsigned    header[4];

    void *      ppdevNext;          // 0010
    int         cPdevRefs;          // 0014
    int         cPdevOpenRefs;       // 0018
    void *      ppdevParent;        // 001c
    unsigned    flags;              // 0020
    unsigned    fIAccelerated;      // 0024
    void *      hsemDevLock;        // 0028
    void *      hsemPointer;        // 002c
    POINT      ptlPointer;         // 0030
    unsigned    unk_0038[2];         // 0038
    SPRITESTATE SpriteState;       // 0040, 476(1dc) bytes

    HFONT      hlfntDefault;        // 021c
    HFONT      hlfntAnsiVariable;   // 0220
    HFONT      hlfntAnsiFixed;      // 0224
    HGDIOBJ    ahsurf[6];          // 0228
    unsigned    unk_0240[2];         // 0240
    void *      prfntActive;        // 0248
    void *      prfntInactive;      // 024c
    unsigned    clnactive;          // 0250
    unsigned    unk_0254[27];        // 0254

    void *      pfnDrvSetPointerShape; // 02c0
```

```
void *      pfnDrvMovePointer;      // 02c4
void *      pfnMovePointer;        // 02c8
void *      pfnSync;              // 02cc
unsigned    unk_02d0;              // 02d0
void *      pfnDrvSetPalette;     // 02d4
unsigned    unk_02d8[2];           // 02d8

void *      pldev;                // 02e0
DHPDEV     dhpdev;               // 02e4
void *      ppalSurf;             // 02e8

DEVINFO    devinfo;              // 02ec—417
GDIINFO    gdiinfo;              // 0418—547

void *      pSurface;             // 0548
void *      hSpooler;              // 054c
unsigned    pDesktopId;            // 0550
unsigned    unk_0554;              // 0554
EDD_DIRECTDRAW_GLOBAL eDirectDrawGlobal; // 1552(0x610) bytes

void *      pGraphicsDevice;      // 0b68
POINT       ptlOrigin;             // 0b6c
DEVMODEW *   pdevmode;             // 0b74
unsigned    unk_0b78[3];            // 0b78
void *      apfn[89];              // 0b84
} PDEV_WIN32K;
```

The PDEV_WIN32K is quite a massive data structure of 3304 bytes, with lots of information regarding a device driver. The main function of PDEV_WIN32K is for GDI engine to call a graphic device driver to execute various user requests. The structure starts with 16 unknown bytes.

Various PDEV_WIN32K structures in the system can be linked in a tree structure, with the ppdevNext field being the link to the next structure; ppdevParent field points to the parent structure. The GDI debugger extension provides a **dpdev** command to decode a PDEV_WIN32K structure. It has a recursive option **-R** to dump all such structures linked from a parent structure. If you use the **-R** option on a PDEV_WIN32K corresponding to a display DC, you will see its ppdevNext field is linked to quite a few font drivers' PDEV_WIN32K structures.

The Windows GDI allows different kinds of graphic drivers, each with different features. The flag field helps to classify different drivers. [Table 3-7](#) lists some flags shown by the GDI extension.

Table 3-7. Flags for PDEV_WIN32K Structure

FLAG	MEANING
PDEV_DISPLAY	Display device
PDEV_HARDWARE_POINTER	Supports hardware cursor
PDEV_GOTFONTS	Has font driver
PDEV_DRIVER_PUNTED_CALL	Driver calls back to GDI engine
PDEV_FONTPRINTER	Font device

Several fields are used to manage the mouse pointer for the display driver. hsemPointer is a semaphore to synchronize access to the mouse pointer. The current location of the mouse is stored in ptlPointer. The device driver provides several callback functions to display the mouse pointer; their addresses are stored in pfnDrvSetPointerShape and pfnDrv Move Pointer fields. On my system, they point to mga64!DrvSetPointerShape and mga64!DrvMovePointer.

The SPRITESTATE and DIRECTDRAWGLOBAL structures embedded in PDEV_WIN32K are for DirectDraw implementation. We cover them in the next section.

Three GDI font handles are stored in PDEV_WIN32K. On my system hlfntDefault is of typeface "System," hlfntAnsiVariable is of typeface "MS Sans Serif," and hlfnt AnsiFixed is of typeface "Courier."

While the Windows GDI wants to be what-you-see-is-what-you-get, hatch brushes create a problem. In the GDI, hatch brushes are defined by 8 × 8 monochrome bitmaps. They normally show quite nice horizontal, vertical, diagonal or cross patterns on 72-dpi to 120-dpi screen display. But on printers, where the resolution goes from 180 dpi to 2400 dpi or even higher, patterns defined by 8 × 8 pixel bitmaps turn out to be levels of grayscale instead of patterns. To make the hatch brush more visible on high-resolution devices, the GDI engine allows a device driver to supply bitmaps to implement the standard six Windows GDI hatch brushes. The device driver's EnablePDEV routine can fill an array of six surface (bitmap) pointers. Their corresponding GDI object handle gets stored in the ahSurf array field. Although the display driver is suggested to provide those handles, display DC's PDEV_WIN32K still holds six valid 8 × 8 bitmap handles provided by the GDI.

The GDI structures normally come in pairs, one for logical description, another for physical implementation. So for a physical device structure like PDEV_WIN32K, there should be a logical description. The pldev field points to such a structure, which can be decoded by the GDI extension's **dldev** command. The LDEV_WIN32K structure forms a double-linked list. Starting from a display DC, the list starts with the display driver, for example "\SystemRoot\System32\mga64.dll," followed by a few "linked-in" font drivers, and ends with ATM font driver "\SystemRoot\System32\atmfd.dll".

```
// Windows 2000, 384 (0x180) bytes
typedef struct
{
```

```
LDEV_WIN32K * nextldev;
LDEV_WIN32K * prevldev;
ULONG      levtype;
ULONG      cRefs;
ULONG      unk_010;
void *     pGdiDriverInfo;
ULONG      ulDriverVersion;
PFN       apfn[89];
} LDEV_WIN32K;
```

According to the log generated by Fosterer, the GDI debugger extension reads a global variable, win32k!gpldevDrivers, to find the first logical graphic device.

The PDEV_WIN32K has a copy of the DEVINFO structure, which is filled by the graphic driver's DrvEnablePDEV entry point. The DEVINFO structure is documented in the DDK. Its main information is the device driver's capabilities in handling curve, color, font, image formats, etc.

PDEV_WIN32K also has a copy of GDIINFO structure, again filled by Drv EnablePDEV entry point and documented in DDK. GDIINFO structure's main information is the size, format, and resolution of the drawing surface. Most of GDIINFO structure's fields can be queried using Win32 API GetDeviceCaps. For example, GetDeviceCaps (hDC, TECHNOLOGY) is related to GDIINFO's ulTechnology field; GetDeviceCaps (hDC, RASTERCAPS) is taken from GDIINFO's flRaster field.

The pSurface field points to a SURFACE structure, on which drawing operations really occur. We will talk about surface structure later in this section. The pdevmode fields point to a DEVMODEW structure, the UNICODE version of DEVMODE. The DEVMODE structure is normally initialized by the graphic driver and modified by user application, both to get information from the device driver and to set options allowed by the device driver. The DEVMODE structure may not be very useful to screen display, but it's critical to the printer driver to set print quality, paper size, media type, resolution, etc.

The final and most important field in a PDEV_WIN32K structure is apfn, a table of 89 function pointers. The Windows 2000 DDI defines 89 functions which the graphical device driver could provide; each has a predefined index. For example, INDEX_DrvEnablePDEV is 0, while INDEX_DrvSynchronizeSurface is 88. Some of those 89 indexes are not used, some are reserved, some are for display drivers only, and some are for printer drivers only. Only a few of them are required to be provided by device drivers; the rest are optional. When a device driver is loaded, its initial entry point Drv EnableDriver is called to fill in a DRVENABLEDATA structure. DRV ENABLEDATA is basically a condensed form of a table of 89 function pointers. Filling 89 function pointers is tedious work, error prone and hard to expand. So the GDI engine allows the driver to provide a list of functions supported by the driver together with their indexes, from which the GDI engine builds the expanded function table. The function table is kept in two places. The logical device structure LDEV_WIN32K stores the original function table, constructed from DRVENABLEDATA, which has only callback functions to the device driver. The physical device structure PDEV_WIN32K stores the function table that the GDI engine will actually use, constructed

from the LDEV_WIN32K table plus entry points from the GDI engine to fill in the function not supported by the device driver. For example, if a device driver does not support DrvBitBlt, it's actually asking the GDI engine to implement it. So PDEV_WIN32K's function table points to a function in win32k.sys, win32k!SpBitBlt, instead of being a NULL pointer.

[Table 3-8](#) shows the contents of an actual DDI function table on my machine.

Table 3-8. Sample PDEV_WIN32K Function Table

Index	Address	Function Name
00	ffd691f0	mga64!DrvEnablePDEV
01	ffd69390	mga64!DrvCompletePDEV
02	ffd69350	mga64!DrvDisablePDEV
03	ffd693b0	mga64!DrvEnableSurface
04	ffd69640	mga64!DrvDisableSurface
05	ffd69760	mga64!DrvAssertMode
06	ffd69710	mga64!DrvOffset
07	ffd68fa0	mga64!DrvResetPDEV
10	ffd4fc00	mga64!DrvCreateDeviceBitmap
11	ffd4fd00	mga64!DrvDeleteDeviceBitmap
12	ffd50fb0	mga64!DrvRealizeBrush
13	ffd50cc0	mga64!DrvDitherColor
14	a00a8fe3	win32k!SpStrokePath
15	a00aaafc	win32k!SpFillPath
17	ffd68890	mga64!DrvPaint
18	a001834b	win32k!SpBitBlt
19	a001d26d	win32k!SpCopyBits
20	a0064500	win32k!SpStretchBlt
22	ffd68c10	mga64!DrvSetPalette
23	a001d72e	win32k!SpTextOut
24	ffd70fcc	mga64!DrvEscape
29	ffd5de10	mga64!DrvSetPointerShape
30	ffd5df90	mga64!DrvMovePointer
31	a00aa63b	win32k!SpLineTo
40	a003c327	win32k!SpSaveScreenBlt
41	ffd69950	mga64!DrvGetModes
43	ffd5bf60	mga64!DrvDestroyFont
59	ffd6ef10	mga64!DrvGetDirectDrawInfo
60	ffd6f190	mga64!DrvEnableDirectDraw
61	fdda0410	mga64!DrvDisableDirectDraw
67	ffd68dc0	mga64!DrvIcmSetDeviceGammRamp
68	a0036184	win32k!SpGradientFill
69	a0073180	win32k!SpStretchBltROP
70	a010d193	win32k!SpPlgBlt

71	a010d0f1	win32k!SpAlphaBlend
74	a010d049	win32k!SpTransparentBlt

As you can see from the sample table, for this display driver, the GDI engine is doing most of the work of drawing curves, filled areas, texts, and bitmaps, while the display driver is handling initialization, mouse pointer, object realization, etc.

GDI Engine Surfaces

GDI drawing calls finally require a surface at the GDI engine level, associated with a device driver, on which all the drawing occurs. Such a device surface has a coordinate system similar to the MM_TEXT mapping mode in the GDI API. Pixels on a surface are addressed by a pair of 28-bit signed integers, with the upper left corner given the coordinates (0, 0). The device surface lies in the lower right quadrant of this coordinate space, where both coordinates are nonnegative. Although coordinates in the Win32 API are stored and passed as 32-bit signed integers, for certain drawing operations the GDI engine uses the lower 4 bits of the 32-bit integer to represent subpixel coordinates, in order to make coordinate calculation more accurate.

The GDI engine defines two main types of surfaces. The first is the GDI engine-managed surface, commonly referred to as a DIB (device-independent bitmap) in the DDK document. GDI engine-managed surfaces are single-plane, packed-pixel, scan-line DWORD aligned bitmaps. If a device driver uses an engine-managed surface, the GDI engine is able to perform all the rendering operations. Simple display drivers or raster-based printer drivers could take advantage of this GDI engine support to make their drivers really simple and easy to maintain. The frame buffer display driver sample which comes with the Windows 2000 DDK creates an engine-managed surface as its primary surface and lets the GDI handle rendering operations. The Windows 2000 printer UniDrv also uses an engine-managed surface, after dividing the physical page into a sequence of rectangle bands.

The second main type of surfaces is device-managed surfaces, through which device drivers are allowed to manage their own surfaces. Internally, a device-managed surface could use either an engine-managed or a non-engine-managed surface format. If it's in engine-managed surface format, the device driver can still call the GDI engine to perform rendering operations.

A device-format bitmap is a special kind of device-managed surface in a non-engine-managed surface format. It is supported to allow certain display drivers to implement faster bitmap-to-screen block transfers. It also allows drivers to draw to banked or device-specific bitmaps in off-screen display memory.

SURFOBJ is the main data structure to represent different kinds of GDI engine surfaces. SURFOBJ is documented in Windows NT/2000 DDK. It is one of the central data structures in the DDI interface, representing both the bitmaps and drawing surfaces. The SURFOBJ structure is duplicated from the DDK document here because of its importance in the GDI engine.

```
typedef struct _SURFOBJ {
    DHSURF dhsurf;
    HSURF hsurf;
    DHPDEV dhpdev;
    HDEV hdev;
    SIZEL sizlBitmap;
    ULONG cjBits;
    PVOID pvBits;
    PVOID pvScan0;
    LONG lDelta;
    ULONG iUniq;
    ULONG iBitmapFormat;
    USHORT iType;
    USHORT fjBitmap;
} SURFOBJ;
```

The first field, dhsurf, stores a device-managed handle used to identify a device-managed surface, which could be a pointer, an index, or any value the device driver knows how to use. hsurf is the GDI handle for the surface, normally a device-dependent bitmap handle or a DIB section handle. dhpdev is a device-managed handle to the device driver's PDEV structure, returned by DrvEnablePDEV entry point. hdev is the GDI engine's logical handle to the physical device.

The pixel size of a surface is specified in SURFOBJ's sizlBitmap field. For the GDI engine-managed surface, pvBits points to the surface bitmap data; cjBits is its size; pvScan0 points to the first scan line of bitmap data. Remember, a DIB could either be top-down or bottom-up. In the latter case, pvBits is not the same as pvScan0. The lDelta field is the number of bytes from one scan line to the next; this enables the GDI engine code to move quickly between scan lines. It would be positive in top-down bitmaps and negative in bottom-up bitmaps. The iUniq field is a nice feature for performance. It specifies the current state of an engine-managed surface, updated each time the surface changes. This allows the device driver to cache surfaces. For example, if a Postscript printer driver receives two bitmap drawing calls with the same source bitmap with unchanged iUniq value, the driver only needs to store the source bitmap the first time and reuse it for the second call.

SURFOBJ's iBitmapFormat field specifies the standard engine-managed surface format most closely matching the surface, which could be 1, 4, 8, 16, 24, or 32 bits per pixel, uncompressed or RLE (run-length-encoding) compressed. The Windows 2000 GDI also makes it possible for the device driver to support JPEG or PNG compressed image, by allowing iBitmapFormat to be either BMF_JPEG or BMF_PNG. But the Windows GDI or the graphics engine itself does not support JPEG or PNG compressed images; it just passes them to the device drivers if they claim to support them.

Table 3-9. Surface Types

SUROBJ.iType	MEANING
STYPE_BITMAP	GDI engine-managed bitmap
STYPE_DEVICE	Device driver-managed surface
STYPE_DEVBITMAP	Device driver-managed device-format bitmap

The iType field tells the types of surface, as shown in [Table 3-9](#).

The final field fjBitmap stores some flags for engine-managed surfaces. Those flags tell whether the bitmap is top-down or bottom-up, initialized to zero, transient, or not in system memory.

If SURFOBJ is supposed to represent all drawing surfaces in the GDI engine, where is the color management information, such as the palette? Color is managed separately from a SURFOBJ in the GDI engine. For every engine drawing call requiring a SURFOBJ, a pointer to XLATEOBJ is passed to translate color between the source and target surface when needed. For example, DrvStretchBlt and DrvPlgBlt have a pxlo parameter which points to a XLATEOBJ object.

Device-Dependent Bitmaps in the GDI Engine

Device-dependent bitmaps (DDBs) are bitmaps managed by graphic device drivers, with the help of the Windows GDI engine. Before you can use a DDB, you have to create a GDI object for it; its handle of type HBITMAP will be returned. Although device-dependent bitmaps are supposed to be supported by the device driver in its unique format, for Windows NT/2000 more and more device drivers are relying on the GDI engine to perform most of the rendering operations on them. This requires them to be in the format supported by the GDI engine.

HBITMAP handles are managed by the GDI handle manager, too. So each handle has 16 bytes of information associated with it in the GDI's object table, among them a pointer to a structure in kernel address space. This structure is referred to as SURFACE in the GDI kernel debugger extension. The main part of a SURFACE structure is the SURFOBJ structure. Here is what a SURFACE structure looks like:

```
// Windows 2000, 128 (0x80) bytes
typedef struct
{
    HGDIOBJ hHmgr;    // 000
    void * pEntry;   // 004
    ULONG cExcLock; // 008
    ULONG Tid;      // 00c

    SURFOBJ surfobj; // 010, documented in DDK
```

```
XDCOBJ * pdcoAA; // 044, shown by gdikdx
FLONG flags; // 048
PPALETTE ppal; // 04c
unsigned unk_050[2]; // 050
SIZEL sizIDim[2]; // 058
HDC hdc; // 060
ULONG cRef; // 064
HPALETTE hpalHint; // 068
unsigned unk_06c[5]; // 06c
} SURFACE;
```

The SURFACE structure starts with a 16-byte header, the same as kernel structures for other types of GDI objects.

The header is followed by a SURFOBJ structure, which holds information about its format, size, reference to bitmap data, etc. The SURFACE structure needs to be a complete description of a GDI bitmap, either a DDB or a DIB section. So within the SURFACE structure, after SURFOBJ, there is a palette handle and pointer to a PALETTE structure. PALETTE is the kernel mode data structure for a GDI logical palette object. We will touch PALETTE later in this section.

The flags field in a SURFACE structure is seen as a combination of API_BITMAP, meaning a bitmap created in Win32 API, and DDB_SURFACE, meaning a Win32 API device-dependent bitmap.

A Win32 API device-dependent bitmap and DIB section can be selected into a device context. In this case, hdc holds the device context handle, and cRef counts the number of times it's selected into a DC. The sizIDim field supports the Win32 SetBitmapDimensionEx and GetBitmapDimensionEx API by providing the storage for the bitmap's physical dimension.

There may be some confusion regarding the terminology used in the WIN32 GDI API and the Windows NT/2000 DDK; both use DIB and DDB. In the Win32 API there are three types of bitmaps, device-dependent bitmaps (DDB), DIB sections, and device-independent bitmaps (DIB). DDBs and DIB sections are managed by the GDI, which means you have to use the GDI API to create them, select them, copy data from them, write data to them, and finally destroy them. DIBs are not managed by the GDI. You can create a DIB on your own, without the help of the GDI. You can read from it and write to it directly through a pointer, instead of going through a handle using the GDI. The GDI does provide several functions to draw DIB into GDI device contexts.

On the GDI engine level, Win32 DDBs, DIBs, and DIB sections are all surfaces. DIBs and DIB sections are apparently in GDI engine-managed surfaces (also referred to as DIB in the DDK document). But DDBs could be in either DIB format or device format (also referred to as DDB in the DDK document), as determined by graphic device drivers.

Each device-dependent bitmap (DDB) has a corresponding GDI handle (HBITMAP). Its full

information is stored in a SURFACE structure in kernel address space. For normal display drivers, for the SURFOBJ structure within its SURFACE structure, its iType field is normally STYPE_BITMAP; fjBitmap is BMF_TOPDOWN; flags is normally API_BITMAP | DDB_SURFACE; pvBits points to kernel address space. So storage space for bits in a DDB is allocated from the shared kernel address space, from the paged-pool area.

DIB Sections in the GDI Engine

In the Win32 API, a DIB section is a bitmap managed by the GDI but directly accessible through a pointer to a user program. You call CreateDIBSection to create a DIB section, by telling the GDI its specification in a BITMAPINFO structure. The GDI returns back an HBITMAP handle, which can be treated the same as a DDB handle, and a pointer to the actual bits, which you can read and write as a pointer to a plain memory block.

In the GDI handle table, a DIB section is almost the same as a DDB. It has a handle and a corresponding SURFACE structure in kernel address space. The main difference is that a DIB section's bits storage is allocated from user address space, not from kernel address space. This makes the bits accessible in the user program; also, the GDI engine drawing can only be executed when the owner process is the current process. SURFOBJ's fjBitmap for a DIB section has a BMF_DONTCACHE, which hints the graphic driver should not trust the iUniq field to cache the bits, because the bitmap bits can be changed by the user program without the GDI's knowledge through the pointer returned by CreateDIBSection. Another small difference is that a DIB section is normally bottom-up, unless its height is supplied as a negative value, the same as for a device-independent bitmap.

We know that device-independent bitmaps (DIBs) are not managed by the GDI. You can't create GDI handles for them. But the DDI interface uses the same SURFOBJ structure when passing a DIB to a device driver, instead of BITMAPINFO structure, which is used to represent DIBs in Win32 API. Apparently, the GDI engine creates a temporary SURFOBJ structure to represent a DIB before calling the GDI engine or device driver entry points.

Brushes in the GDI Engine

Brushes specify the color and pattern an area should be filled with. In the Win32 API, you can create solid brushes, hatched brushes, DDB pattern brushes, and DIB pattern brushes. We know from [Section 3.5](#) that, for the solid brush, there is a small user mode structure which basically stores brush color, to allow efficient reuse of solid brushes. For all other brushes, the GDI keeps all the information in a kernel structure BRUSH.

```
typedef struct
{
    unsigned AttrFlags;
    COLORREF lbColor;
```

```
} BRUSHATTR;

// Windows 2000, 112 (0x70) bytes (?)
typedef struct
{
    HGDIOBJ    hHmgr;      // 000, GDI kernel object header
    void *     pEntry;     // 004
    ULONG      cExcLock;   // 008
    ULONG      Tid;        // 00c

    ULONG      ulStyle;    // 010
    HBITMAP    hbmPattern; // 014
    HANDLE     hbmClient;  // 018
    ULONG      flAttrs;    // 01c

    ULONG      ulBrushUnique; // 020
    BRUSHATTR * pbrushtrr;  // 024
    BRUSHATTR brushattr;   // 028
    unsigned   unk_030;     // 030
    unsigned   bCacheGrabbed; // 034
    COLORREF   crBack;     // 038
    COLORREF   crFore;     // 03c
    ULONG      ulPalTime;   // 040
    ULONG      ulSurfTime;  // 044
    ULONG      ulRealization; // 048
    unsigned   unk_04c[3];   // 04c
    ULONG      ulPenWidth;  // 058
    unsigned   unk_05c;     // 05c
    ULONG      ulPenStyle;  // 060
    DWORD *    pStyle;     // 064
    ULONG      dwStyleCount; // 068
    unsigned   unk_06c;     // 06c
} BRUSH;
```

The BRUSH structure starts with the standard 16-byte header for the GDI kernel object. It's followed by a brush style field ulStyle, which does not have the same value as in LOGBRUSH structure. The GDI kernel debugger extension decodes it as HS_CROSS, HS_PAT, HS_DITHEREDCLR, etc. The crBack and crFore fields keep the device context's foreground and background colors, while brushattr.lbColor keeps the real color of the brush. The flAttrs field stores some extra flags, as summarized in [Table 3-10](#).

For a DIB pattern brush, the GDI creates a bitmap object for pattern bitmap, whose handle is stored in the hbmPattern field, while hbmClient is still the HGLOBAL passed to CreateDIBPatternBrush call.

For a pattern brush (DDB pattern brush), the GDI copies the original DDB and stores its handle in hbmPattern, while hbmClient keeps the original DDB handle. Making a copy of the original bitmap allows the programmer to delete the original DDB after a pattern brush is created.

A pattern brush object is never alone; it's always paired with its pattern bitmap object. By now, we should have a good idea how various types of brushes are represented in the GDI engine.

Table 3-10. BRUSH Attributes

BRUSH.flAttrs flags	MEANING
BR_NEED_BK_CLR (0x0002)	Background color is needed
BR_DITHER_OK (0x0004)	Allow color dithering
BR_IS_SOLID (0x0010)	Solid brush
BR_IS_HATCH (0x0020)	Hatch brush
BR_IS_BITMAP (0x0040)	DDB pattern brush
BR_IS_DIB (0x0080)	DIB pattern brush
BR_IS_NULL (0x0100)	Null/hollow brush
BR_IS_GLOBAL (0x0200)	Stock objects
BR_IS_PEN (0x0400)	Pen
BR_IS_OLDSTYLEPEN (0x0800)	Geometric pen
BR_IS_MASKING (0x8000)	Pattern bitmap is used as transparent mask (?)
BR_CACHED_IS_SOLID (0x80000000)	

Pens in the GDI Engine

Pens specify the color and styles of lines, arcs, and curves. The Win32 API allows you to create cosmetic or geometric pens of different styles, widths, and brush attributes. Surprisingly, there is no special data structure defined in the GDI engine for pens. Pens use the same BRUSH structure as brushes. This arrangement makes sense, if you notice that extended pens created with ExtCreatePen are defined using a LOGBRUSH structure.

To tell pens from brushes, the GDI engine uses the BR_IS_PEN flag in flAttrs field. Another flag, BR_IS_OLDSTYLEPEN, tells if the pen is created using the old-style CreatePen or CreatePenIndirect function instead of using ExtCreatePen. The fields ulPenWidth, ulPenStyle, pStyle, and dwStyleCount have the same meaning as similar fields in the EXTLOGOPEN structure defined in WIN32 API.

The GDI kernel debugger extension provides a single command, **dpbrush**, to decode the BRUSH structure, which only decodes true brush-related fields. This information may not be complete for

pens created with ExtCreatePen.

Palletes in the GDI Engine

A palette is a color-lookup table, which can be used to translate color index into RGB values, or translate RGB values back to color index. To use palette with a device context, you need to create a logical palette by calling CreatePalette or CreateHalftone Palette, which returns a logical palette handle of the type HPALETTE.

Besides the palette, which is normally specified using a LOGPALETTE structure, the Win32 uses another form of color-lookup table, the BITMAPINFO structure that comes as part of a DIB or DIB section. The number of indexes in the color-lookup table can be calculated using the bmiHeader part of the BITMAPINFO structure, while the bmiColors array is the lookup table. The BITMAPINFO structure allows lookup by index for bitmaps with no more than 256 colors. For 16-, 24-, and 32-bit DIBs, it also allows defining the masks to get the red, green, and blue components out of 16-, 24-, or 32-bit raw data.

The GDI engine needs to support both forms of the color-lookup table in a uniform way. This goal is achieved using the EPALOBJ structure (name given by gdikdx).

```
typedef unsigned long HDEVPPAL;
typedef void * PTRANSLATE;
typedef void * PRGB555XL;
typedef unsigned PAL ULONG;

// Windows 2000, 84+4n bytes
typedef struct _EPALOBJ
{
    HGDIOBJ hHmgr;      // 000, GDI kernel object header
    void * pEntry;      // 004
    ULONG cExcLock;    // 008
    ULONG Tid;          // 00c

    FLONG fIPal;        // 010
    ULONG cEntries;     // 014
    ULONG ulTime;       // 018
    HDC hdcHead;        // 01c
    HDEVPPAL hSelected; // 020
    ULONG cRefhpal;     // 024
    ULONG cRefRegular;  // 028
    PTRANSLATE ptransFore; // 02c

    PTRANSLATE ptransCurrent; // 030
    PTRANSLATE ptransOld;   // 034
```

```
unsigned unk_038;      // 038
PFN    pGetNearest;   // 03c
PFN    pGetMatch;     // 040
ULONG  ulRGBTime;    // 044
PRGB555XL pRGBClate; // 048
_EPALOBJ * pPalette; // 04c, this
PAL ULONG * papalColor; // 050, this->apalColor
PAL ULONG apalColor[1]; // 054
} EPALOBJ;
```

The EPALOBJ is the kernel mode structure for a logical palette object, so it starts with the standard header, as do all GDI handles. The fIPal tells the difference between types of color-lookup tables. [Table 3-11](#) shows what is seen from the output from the GDI kernel debugger extension, also partly from winddi.h.

The cEntries field counts the number of entries in the color-lookup table apalColor. Those two fields are similar to what's in LOGPALETTE structure. GDI engine stores the addresses of two routines in EPALOBJ structure, pGetNearest and pGetMatch. Although they may well point to somewhere else, pGetNearest points to win32k!ul IndexedGet-NearestFromPalentry, and pGetMatch points to ullIndexedGet MatchFromPalentry on my machine.

The EPALOBJ structure is not exposed directly to device drivers. Winddi.h defines a PALOBJ structure which has only a single ulReserved field. Device drivers are supposed to call the Windows graphics engine function PALOBJ_cGetColors to read the color-lookup table. A related data structure is XLATEOBJ, which also provides a XLATEOBJ_cGetPalette routine to allow accessing the color-lookup table.

Regions in the GDI Engine

A region is a set of points on a graphic device's drawing surface. A region can be defined by a rectangle, polygon, ellipse, or any combination of them. Regions can be painted, inverted, framed, and used to achieve clipping or hit testing. The most common use of regions may be clipping.

Table 3-11. EPALOBJ Flags

EPALOBJ.fIPal FLAGS	VALUE	MEANING
PAL_INDEXED	0x0001	Indexed palette
PAL_BITFIELDS	0x0002	Bit fields used for DIB, DIB section
PAL_RGB	0x0004	Red, green, blue
PAL_BGR	0x0008	Blue, green, red
PAL_CMYK	0x0010	Cyan, magenta, yellow, black
PAL_DC	0x0100	
PAL_FIXED	0x0200	Can't be changed
PAL_FREE	0x0400	
PAL_MONOCHROME	0x2000	Two colors only
PAL_DIBSECTION	0x8000	Used for a DIB section
PAL_HT	0x100000	Halftone palette
PAL_RGB16_555	0x200000	16-bit RGB in 555 format
PAL_RGB16_565	0x400000	16-bit RGB in 565 format

Regions are one of the several types of objects managed by the GDI. To use a region, you call routines like CreateRectRgn, CreateRoundRectRgn, or CreateEllipticRgn to create a new region, or combine existing regions using logical operations. Those routines returns a GDI object handle, HRGN, which you can pass back to GDI routines.

We found in [Section 3.5](#) that for rectangle regions, GDI stores the rectangle's coordinates in a user mode data structure. But for other regions the information is apparently kept in kernel address space.

The GDI kernel debugger extension provides a **dr** command to decode a HRGN, or a pointer to its kernel data structure, REGION. The command even displays all rectangles the region is composed of. With the help of this command, here is the REGION structure as we know it.

```
// Windows 2000, variable size
// don't reference scnPntCntToo directly
typedef struct
{
    LONG scnPntCnt;    // count of x coordinates
    LONG scnPntTop;   // top line (inclusive)
    LONG scnPntBottom; // bottom line (exclusive)
    LONG scnPntX[2];  // v-length array of x pairs
    LONG scnPntCntToo; // same as scnPntCnt;
} SCAN;
```

```
// Windows 2000, variable size
struct REGION
{
    HGDIOBJ hHmgr;      // 000, GDI kernel object header
    void * pEntry;       // 004
    ULONG cExcLock;     // 008
    ULONG Tid;          // 00c

    unsigned sizeObj;    // 010
    unsigned unk_014[2]; // 014
    SCAN * pscnTail;    // 01c
    unsigned sizeRgn;    // 020
    unsigned cScans;     // 024
    RECTL rcl;          // 028
    SCAN scnHead[1];    // 038
};
```

The REGION structure starts with a standard GDI kernel structure header. We mentioned in [Section 3.5](#) that for a region defined by a single rectangle, the GDI optimizes the creation by attaching a user mode RECT structure to the GDI handle. In doing so, the GDI binds the region object to that process. Any changes to the REGION will affect the creating process. To keep the link, the GDI engine stores the thread identifier of the creator in the header.

The REGION structure is a variable-size structure. The structure itself stores all the information about a region, which could grow or shrink in size when region operations are applied. For example, if a region is combined with another region with RGN_OR, the size tends to grow; if combined with RGN_AND, the size tends to shrink. To reduce the number of memory allocation/deallocation calls, the GDI engine does not allocate the exact amount of memory needed to represent a region; instead, it allocates more space to allow for some growth without reallocation. The sizeObj field of a REGION seems to be the allocation size of a REGION structure, while the actual size used is kept in the sizeRgn field.

The rcl field stores the bounding rectangle for the region. The main data within a REGION is an array of SCAN structure. The cScans field counts the number of such structures in the array, while the pscnTail field points past the end of the last structure in the array.

Programmers normally do not keep a pointer to the last item of an array, because this can easily be calculated from the starting address, number of elements, and size of elements. But the interesting thing here is that the SCAN structure is a variable-size structure. SCAN is not documented in Windows NT/2000 DDK, but a 16-bit version of it is documented in Windows 95 DDK.

The SCAN structure keeps information about a single “scan line” in a region, which may be or may not be a single pixel high in the coordinate space. More precisely, SCAN keeps information about the intersection of the region with the area between two horizontal lines, insofar as corresponding intersections of the defining curve of the region with the two lines form vertical lines. The GDI engine

divides a region into a series of SCANS in top-down order. Because the intersections of the region with the top and bottom line of the SCAN are required to be the same, the GDI engine needs to store only one of them. So the SCAN structure stores the *y* value for the top and bottom line, pairs of *x* values for the intersections, and two copies of the number of intersections. So, for complicated regions, such as regions with holes, the SCAN structure saves the memory space needed to represent a region.

The two copies of intersection number are stored in the first and last fields of the SCAN structure. Because SCAN is a variable-size structure, its last field does not have a fixed offset from the structure. If you ask why the REGION and SCAN structures are designed so strangely, the GDI engine does have a good reason to do so. Regions are normally passed as CLIPOBJ to routines in graphic drivers. The DDI interface does not expose all the internal data structure of CLIPOBJ. Instead, it allows graphic drivers to enumerate all the rectangles making up the region through the CLIPOBJ_bEnum routine. The driver can specify the order in which the rectangles can be enumerated in CLIPOBJ_cEnumStart call. The GDI engine allows left to right, top to bottom; right to left, top to bottom; left to right; bottom to top; right to left, bottom to top; or any order that is convenient for the GDI. The pscanTail field in REGION allows the GDI engine to quickly jump to the last SCAN structure. The scnPntCount field in SCAN allows a jump from left to right, or move to the next SCAN structure in top-down order. The scnPntCountToo field allows a jump from right to left, or move to the next SCAN structure in bottom-up order.

Table 3-12. Array of SCAN Structures in a REGION Structure, for a Circle

Cnt	Top	Bottom	X[]	CntToo
0	-maxint-1	0		0
2	0	1	47, 52	2
2	1	2	39, 60	2
...				
2	39	47	1, 98	2
2	47	52	0, 99	2
2	52	60	1, 98	2
...				
2	97	98	39, 60	2
2	98	99	47, 52	2
0	99	maxint		0

Let's look at an example, to demonstrate how the REGION structure is linked with regions we know in the Win32 API. If you call CreateEllipticRgn(0, 0, 100, 100), you get back a region handle. Pass it to the GDI extension's **dr** command; it will show the address of a REGION structure and list all the rectangles making up the region. The REGION has 63 SCAN structures, with a bounding rectangle of [0, 0, 99, 99], occupying 52 bytes. [Table 3-12](#) shows a condensed list of its SCAN structure array.

From this example, we can see that the REGION structure is an approximation of the original shape, using a combination of rectangles in integer coordinates. So if you create a GDI handle to a region and later try to scale it using a transformation (using GetRegionData and then ExtCreateRegion with a XFORM parameter), the result would not be the same as when you scale it mathematically first and then create a GDI handle to it.

The REGION structure represents a region in left-right, top-bottom order. The coordinates are top, left inclusive, and bottom, right exclusive. The GDI tries to be as accurate as possible when creating REGION structures, so lots of SCANS are a single pixel high. For example, the first few and last few SCANS for the circle are a single pixel high. But when possible, the GDI allows SCAN to be as high as possible to save space. For example, the center part of a circle is quite close to two vertical lines. So you can see the middle SCAN structure in the array shown above is from 47 to 52.

To accurately represent a region that is not so much a rectangle, the size of the REGION structure tends to be proportional to the height of a region, not so much to the width of a region. For example, if you double the height of an ellipse, the size of REGION may be doubled; while if you double the width of an ellipse, the size of REGION may not change at all. The number of SCAN structures in a REGION also tends to be proportional to the height of a region. The number of SCAN structures and size of REGION have a direct impact on the GDI engine and device driver memory usage and performance, especially for high-resolution printing modes for high-quality printers.

Another interesting thing about the circle sample shown in [Table 3-12](#) is the first and last SCAN structures. They are actually empty SCANS, or SCANS without x coordinates. They basically state that between $y=-\text{maxint}-1$ and $y=0$, also between $y=99$ and $y=\text{maxint}$, there is nothing in the coordinate space that belongs to the region. If they do not represent any visible portions of the region, why are they stored in the precious kernel address space? They exist to make region operations uniform and easier to implement. For example, to invert the region, you can use the same number of SCANS; just add $-\text{maxint}-1$ and maxint as the first and last x coordinates for every SCAN.

So how is an empty region represented? It's a REGION with a $\{0, 0, 0, 0\}$ as the bounding box, and a single SCAN $\{0, -\text{maxint}-1, \text{maxint}, 0\}$.

Have you noticed that when you call the GDI to create a circle region with $0, 0, 100, 100$ as coordinates, the GDI returns back a region with $0, 0, 99, 99$ as the bounding box, which is still bottom-right exclusive? In other words, CreateEllipticRgn creates a smaller shape than Ellipse would do. Yes, that's the Windows reality. This is a known defect with Windows, from Windows 3.0 up to Windows 2000. It's documented in MSDN Win32 SDK knowledge-base article Q83807.

The REGION structure is not exposed in the Win32 API to programmers like us, nor in DDI to device-driver programmers. In the Win32 API, the only low-level region structure you can get is the RGNDATA structure, used by GetRegionData and ExtCreateRegion. RGNDATA uses an array of rectangles, instead of an array of SCANS. In DDI interface, there is the abstract CLIPROBJ structure. You need to call CLIPROBJ_bEnum to get the rectangles making up the region.

Paths in the GDI Engine

A path is a collection of shapes (or figures) that can be filled, outlined, or both filled and outlined. You can use the Win32 API to create a path, but you won't even get a handle back.

As a programmer, you know there must be a data structure that the GDI uses internally to represent a path during its formation and when using it. The Windows graphics engine (Win32k.sys) even exports quite a large set of functions to manipulate path object to device drivers. According to the GDI kernel debugger extension, paths do exist in the GDI object table. You can use **dumpobj PATH** to list all the path objects in the system. Unlike other GDI objects, the GDI debugger extension does not provide a command to decode a GDI path object handle or its kernel object stored in the GDI handle table. The **dpo** command only decodes a PATHOBJ structure passed to the GDI engine or device driver routines—for example, EngStrokePath or Dev StrokeAndFillPath.

With several sample binary data of path objects and the help of the Win32 API and DDK document, here is what we know about the structure representing a path in the GDI engine, named PATH here.

```
// Windows 2000, variable size
typedef struct _PATHDT
{
    _PATHDT * pNext;          // 000
    _PATHDT * pLast;         // 004
    unsigned nFlags;          // 008
    unsigned pointno;        // 00c
    POINTFIX point[1];       // 010
} PATHDT;

// Windows 2000, variable size
typedef struct
{
    unsigned unk_000;          // 000
    void *  pTail;            // 004
    unsigned nAllocSize;       // 008
    PATHDT  pathdt[1];        // 010
} PATHDEF;

// Windows 2000, ? bytes
typedef struct
{
    HGDIOBJ hHmgr;           // 000, GDI kernel object header
    void *  pEntry;           // 004
    ULONG   cExcLock;         // 008
    ULONG   Tid;              // 00c
```

```
PATHDEF * ppachain;      // 010
SEGMENT * pFirst;        // 014
PATHDT * pprfirst;       // 014
PATHDT * pprlast;        // 018
RECTFX rcfxBoundBox;    // 01c
POINTFX ptxSubPathStart; // 02c
ULONG nCurves;          // 034
unsigned unk_38[10];     // 038
} PATH;
```

Unlike the REGION structure, the PATH structure is a fixed-size structure. It starts with the standard 16-byte GDI kernel object header, followed by a pointer (ppachain field) to a PATHDEF structure, which stores the real definition of a path. Recall that a path is a collection of figures; its internal representation PATHDEF is a list of PATHDT structures, each of which may be one, part of a figure making up the path. The PATH structure keeps pointers to its first and last PATHDT structures in the pprfirst, pprlast fields. Also appearing in PATH are the bounding box and starting point for the path.

We mentioned that the device coordinate is stored in 32-bit fixed-point values, unlike the Win32 GDI API, which allows 32-bit signed values. The PATH structure is an example. Both the bounding box and starting point are represented in 32-bit fixed-point format. The upper 28 bits of the 32 are the integer part, while the lower 4 bits are the fraction part. For example, 1 would be 0x10; 1.125 could be 0x12.

Microsoft calls this format FIX coordinates or fractional coordinates. Fractional coordinates allow specifying coordinates on the device surface within 1/16 of a pixel. FIX notation is used for lines and Bezier curves, which are the basic components of paths. It improves accuracy without the burden of floating-point numbers.

The PATHDEF structure is a variable-length structure, containing all PATHDT structures in a path. Its nAllocSize field remembers the size of the current memory block, while the pTail field points to the first free byte. It's quite easy to determine if the path is running out of memory using both of them. After them comes a sequence of PATHDT structures forming a double-linked list.

A PATHDT structure represents a group of points on a curve with certain common attributes. For every structure, its pNext field points to the next PATHDT structure or NULL for the end of a list; its pLast field points to the previous PATHDT structure or NULL for the start of a list. The flags field keeps the common attributes of the points. Those flags are documented in Windows NT/2000 DDI when the PATHDATA structure is described, as listed in [Table 3-13](#).

Table 3-13. PATHDT Flags

PATHDT.flags FLAGS	VALUE	MEANING
PD_BEGINSUBPATH	0x0001	First point begins a new subpath (figure)
PD_ENDSUBPATH	0x0002	Last point ends the subpath (figure)
PD_RESETSTYLE	0x0004	Reset style to zero at the beginning of the subpath
PD_CLOSEFIGURE	0x0008	Add an implicit line connecting last point with first point of the subpath (figure)
PD_BEZIERS	0x0010	Each set of three points describes Bezier curve, not line segments

So a path is a GDI object, just like a region or a DDB. GDI objects need to be selected into a device context to be used in GDI drawing calls. A path, unlike other GDI objects, does not have an explicit selection function. It's selected implicitly when it's created. When a new path is created, the old path in that device context is destroyed. But the GDI engine still needs to remember path handles for different device contexts. This assures that the GDI engine keeps a device context's path object handle within its DEVLEVEL structure, in the hpath field. Following that, there are also the flag field, flPath, and an LINEATTRS structure for the path's line attributes.

Now let's look at an example, a few lines of Win32 code which create a path.

```
const POINT Points[3] = { {200,50}, {250,150}, {300, 50} };

BeginPath(hDC);
MoveToEx(hDC, 100, 100, NULL);
LineTo(hDC, 150, 150);
PolyBezierTo(hDC, & Points[0], 3);
EndPath(hDC);
```

With the help of the GDI kernel debugger extension, you can search for all GDI path objects in the system. Use the **dumpobj PATH** command, then use **dt <gdihandle>** to display a GDI handle's entry in the GDI object table. From there we get a pointer to the PATH structure, which has a pointer to a PATHDEF structure. The PATHDEF structure looks like this:

```
// sample PATHDEF structure
0000: unk_000 0x00000000
0004: pTail    & pathdt[2]
0008: nAllocSize 0xfc0
000c: pathdt[0] & pathdt[1], NULL, 5, 2,
          100.0, 100.0, 150.0, 150.0
0014: pathdt[1] NULL, & pathdt[0], 0x12, 3,
          200.0, 50.0, 250.0, 150.0, 300.0, 50.0
0054: pathdt[2]
```

The GDI engine allocates 4032 (0xfc0) bytes for this path, of which only 84 (0x54) bytes are currently occupied. There are lots of spaces for the path to grow. There are two PATHDT structures in the PATHDEF, forming a double-linked list. The first PATHDT structure has two points, with PD_BEGINSUBPATH | PD_RESETSTYLE flags on. So it's two points forming a line segment. The second PATHDT structure has three points, with PD_ENDSUBPATH | PD_BEZIERS. So it's one Bezier curve continuing from the last point, which also ends the subpath. The PATHDEF structure records the exact information specified by the Win32 code.

Now we know that straight lines and Bezier curves, or any combination of them, can be represented by a PATH structure. For example, CloseFigure, LineTo, MoveToEx, PolyBezier, PolyBezierTo, Polygon, Polyline, PolylineTo, PolyPolygon, and PolyPoly line can all be converted to a series of lines and Bezier curves. But, Windows 95/98 allows TextOut and ExtTextOut in path creation. How does a PATH structure represent text? It turns out that only TrueType fonts are allowed in path creation, and only outlines of text strings, which are basically Bezier curves, are actually recorded as part of the paths.

Besides the functions mentioned, Windows NT/2000 allows elliptic curves in a path. For example, you can use AngleArc, Arc, ArcTo, Ellipse, Pie, etc., in a path definition. How does the GDI engine handle them? It breaks elliptic curves into Bezier curves, similar to the way it breaks nonrectangle regions into a collection of scan lines. Let's add two more lines before the EndPath(hDC) call in the code shown above:

```
...
CloseFigure(hDC);
Ellipse(hDC, -100, -100, 100, 100);
...
```

The CloseFigure call closes the second PATHDT structure shown above. The Ellipse adds one more PATHDT structure to the list, a group of Bezier curves with 13 points. The first point is the starting point of a new figure; the remaining 12 points form 4 Bezier curves. The GDI engine is approximating an ellipse using 4 Bezier curves. Here are the 13 control points:

```
{ 99, -0.5 },
{ 99, -55.4375 }, { 54.5, -100 }, { -0.5, -100 },
{ -55.5, -100 }, { -100, -55.4375 }, { -100, -0.5 },
{ -100, 54.4375 }, { -55.5, 99 }, { -0.5, 99 },
{ 54.5, 99 }, { 99, 54.4375 }, { 99, -0.5 }
```

Now we see why the PATH structure uses FIX coordinates. An ellipse will not look like a perfect ellipse with its coordinates rounded to integers.

The PATH structure is not used only to store Win32 GDI paths, it's also very important in DDI

interface, which is the interface between the GDI engine and the graphic device drivers. Line-drawing calls, for example, LineTo and PolyBezier, are converted to DrvStrokePath, which accepts a pointer to the PATHOBJ structure. Area-filling calls, for example, Ellipse and Polygon, are converted to DrvStrokeAndFillPath, which also accepts a PATHOBJ pointer. Compared with Windows NT 4.0, Windows 2000 adds a new entry point DrvLineTo in order to improve the performance of integer -end-points-only LineTo calls.

The PATHOBJ structure is another DDI semihidden structure with only two public fields. You can use several functions provided by the GDI engine to query the components of a path, construct a new path, or add a new curve to extend a path. For example, EngCreatePath creates a new PATHOBJ; PATHOBJ_bPolyBezierTo adds Bezier curves to a path; PATHOBJ_bEnum enumerates records in a path in a PATHDATA structure, which is very similar to the PATHDT structure shown above.

Fonts in the GDI Engine

What we normally refer to as fonts in Win32 API are actually logical fonts. Logical fonts are created by CreateFont, CreateFontIndirect, or CreateFontDirectEx calls, given a description of the features you want about the font. The GDI, to be more specific the font mapper, will find an available physical font which best meets your requirement.

Logical font objects created by the GDI are referred by their handles, which are of HFONT type. The GDI kernel debugger extension lists them as LFONT type. For example, you can use **dumpobj** **LFONT** to list all logical font handles in the system. Given a logical font handle, you can use the **helf** command to decode what's behind the handle in kernel address space. The command simply gives a dump of the corresponding LOGFONTW structure:

```
// Windows 2000, ? bytes
typedef struct
{
    HGDIOBJ     hHmgr;      // 000, kernel object header
    void *      pEntry;     // 004
    ULONG       cExcLock;   // 008
    ULONG       Tid;        // 00c

    unsigned     unk_010[3];  // 010
    PDEV_WIN32K * ppdev;    // 01c
    unsigned     unk_020[8];  // 020
    HGDIOBJ     hPFE;       // 040
    unsigned     unk_020[39]; // 044
    WCHAR       Face[32];   // 0d0
    unsigned     nSize;      // 110
    ENUMLOGFONTEXW enumlogfontex; // 114
} LFONT;
```

The actual data stored in the kernel for a logical font are much more than the LOGFONTW structure shown by the GDI extension. Even the GetObject function on a logical font handle returns a structure of 356 bytes, which looks more like an ENUMLOGFONTEXW structure. The first field of it is indeed a LOGFONTW structure. Another noticeable field is a pointer to the GDI engine physical device structure.

So in the GDI engine, the LFONT structure kept for a logical font is basically a LOGFONTW structure with some extra fields forming an ENUMLOGFONTEXW structure, which is quite fair. But where is the link between logical fonts and physical fonts? How is font information passed to graphic device driver entry points like Drv ExtTextOut? The GDI kernel debugger extension shows one more undocumented GDI handle, type PFE. Each LFONT structure has a handle to a PFE in the hPFE field.

You can use the GDI extension command **dumpobj PFE** to list all the PFE handles, and then use the **hpfe** command to look behind individual handles. If you use the command on a PFE handle, the GDI extension talks back: "Why are you using a handle? Nobody uses a handle to a PPE." So you have to switch to another command **pfe** to dump the kernel mode PFE structure for the handle. Here is what a PFE structure looks like:

```
// Windows 2000, 108 (0x6c) bytes
struct PFF;

typedef struct
{
    HGDIOBJ    hHmgr;      // 000, GDI kernel object header
    void *      pEntry;     // 004
    ULONG       cExcLock;   // 008
    ULONG       Tid;        // 00c

    PFF *      pPFF;       // 010, pff
    ULONG       iFont;      // 014
    ULONG       flPFE;      // 018
    FD_GLYPHSET * pfdg;    // 01c, gs
    void *      unk_020;    // 020 f8ddef60
    IFIMETRICS * pifi;     // 024, ifi
    unsigned    idifi;     // 028
    void *      pkp;        // 02c
    unsigned    idkp;      // 030
    unsigned    ckp;        // 034
    unsigned    iOrientation; // 038
    unsigned    cjEfdwPFE;  // 03c
    void *      pgiset;    // 040
    unsigned    ulTimeStamp; // 044
```

```
unsigned    ufi;      // 048
unsigned    unk_04c;   // 04c
unsigned    pid;      // 050
unsigned    ql;       // 054
unsigned    unk_058;   // 058
void *     pFIEntry; // 05c
unsigned    cAlt;     // 060
unsigned    cPfdgRef; // 064
unsigned    aiFamilyName; // 068
} PFE;
```

A PFE structure starts with the standard GDI kernel object header, which is 16 bytes long. The pPFF fields points to a PFF structure which stores information about a physical font file. We will cover the PFF structure later in this section. The pfdg file points to a FD_GLYPHSET structure, which is documented in DDK.

A FD_GLYPHSET structure defines the mapping from Unicode characters to glyph handles provided by a font internally. Unicode characters are represented using 16-bit values. There are thousands of characters in a Unicode character set. A particular font may only support a small subset of the Unicode character set. The font is a collection of glyphs, each with a unique handle as its identifier. The FD_GLYPHSET structure enables the GDI engine to map Unicode characters to glyph handles, through which the GDI engine can access the actual glyph. The GDI kernel debugger extension provides the **gs** command to decode a FD_GLYPHSET structure. For example, on “Small Fonts” (SMALLF.FON), the command shows the font has only 224 glyphs. The glyph for space character is 0, for “A” is 0x21, for 0x2122 is ox79. The FD_GLYPHSET structure is created by the DrvQueryFontTree entry point of a font driver, when the iMode parameter is QFT_GLYPHSET.

Another interesting field is the pifi field, which holds a pointer to an IFI METRICS structure, again documented in the DDK. The IFIMETRICS structure defines information for a given typeface that the GDI can understand. It contains information about family name, style name, face name, unique name, simulation capabilities, embedding ID, and, last but not least, the 10-byte panose array which describes the visual characteristics of a typeface. The DrvQueryFont entry of a font driver fills the IFIMETRICS structure. The GDI debugger extension has an **ifi** command to decode the IFIMETRICS structure. For example, on “Small Fonts,” the command returns 1-bpp bitmaps, does integral scaling and 90-degree rotating, and supports bold, italic, and bold italic simulation.

A logical font is created by an individual Win32 process. When a process is destroyed, all its GDI handles get destroyed and reused. But PFE handles are system wide, not attached to any user process. Multiple logical fonts may be mapped to the same PFE.

A PFF structure describes a physical font file. The GDI extension provides the **pff** command to decode it, as you may have guessed. A PFF structure looks like this:

```
struct RFONT;
```

```
typedef struct _PFF
{
    ULONG    sizeofThis; // 000
    _PFF *   pPFFNext; // 004, pff
    _PFF *   pPFFPrev; // 008, pff
    WCHAR *  pwszPathName; // 00c
    ULONG    cwc;       // 010
    ULONG    cFiles;    // 014
    unsigned unk_018[2]; // 018
    ULONG    flState;   // 020
    ULONG    cLoaded;   // 024
    ULONG    cNotEnum; // 028
    ULONG    cRFONT;    // 02c
    RFONT *  prfntList; // 030, fo
    void *   hff;      // 034
    void *   hdev;     // 038
    unsigned dhpdev;   // 03c

    void *   pfhFace;  // 040
    void *   pfhFamily; // 044
    void *   pfhUFI;   // 048
    void *   pPFT;     // 04c, pft

    ULONG    ulCheckSum; // 050
    unsigned unk_054;   // 054
    ULONG    cFonts;    // 058
    void *   ppfv;     // 05c
    void *   pPvtDataHead; // 060
    unsigned unk_064;   // 064
    PFE *   pPFE;     // 068, pfe
    WCHAR    wszStrings[1]; // 06c
} PFF;
```

PFF structures in the GDI engine are chained together into double-linked lists. The pPFFNext, pPFFPrev fields store the links. The next field points to the name of the font file on disk, for example, “\??\C:\WIN2000\FONTS\SMALLF.FON,” for which the flState has PFF_STATE_PERMANET_FONT. The cLoaded field indicates whether the file is loaded into RAM; the cRFONT is the number of realized fonts created based on this font, while prfntList points to the first realized font in a list form.

The pPFT field points to a PFT structure, which is a table of PFF structures. The PFT structure can be decoded by the **pft** command. A PFT structure looks like this:

```
typedef struct
{
    void * pfhFaimly; // 000
    void * pfhFace; // 004
    void * pfhUFI; // 008
    ULONG cBuckets; // 00c
    ULONG cFiles; // 010
    PFF * apPFF[1]; // 014
} PFT;
```

The first three fields of a PFT structure point to three font hash tables, which can be decoded by the **fh** command. The hash tables are designed to match logical fonts quickly with physical fonts. Fonts are put into buckets in a PFT structure, which has been seen to have 100 buckets. The PFT structure stores the pointer to the first PFF structure in all the buckets, each forming a double-linked list through two link fields in the PFF structure. The cFiles field counts the number of total font files linked by the PFT structure.

The GDI engine keeps three tables of PFF structures, one for public fonts, another one for private fonts, and the last one for device fonts. Three global variables hold pointers to these tables—`win32k!gpPFTPublic` for public fonts, `win32k!gpPFT Private`, and `win32k!gpPFTDevice` for device fonts. The GDI debugger extension uses those variables to provide three commands to display the three tables. They are **pubft**, **pvtft**, and **devft**.

If you try the **pubft** command, here may be some of the fonts that get listed:

```
apPFF[2] "\??\C:\WIN2000\FONTS\TREUCBD.TTF"
        "\??\C:\WIN2000\FONTS\CGA80W0A.FON"
apPFF[3] "\??\C:\WIN2000\FONTS\MICROSS.TTF"
apPFF[5] "\??\C:\WIN2000\FONTS\PALA.TTF"
...
apPFF[98] "\??\C:\WIN2000\FONTS\TIMES1.TTF"
```

Now it's the time to describe the most important font structure to a graphic driver, the **FONTOBJ** structure, and its expanded version, the **RFONT** structure. We know the **LFONT** structure describes a logical font, a requirement for a particular instance of a font with a fixed size and rotation angle, special features like weight, etc. On the other hand, a font file, as described by the **PFF** structure, is a generic font template which can be scaled to different sizes, rotated to different angles, and enhanced with different features. A simple way to use the font is that, for every character in a text string, the GDI engine queries the font driver for the generic character outline, scales it to the right size, rotates it to the right angle, converts it to a bitmap, uses it, and forgets about it. But this will be very inefficient, especially considering that, for single-byte fonts, the limited number of characters in a font can easily be cached, saving lots of time to regenerate the bitmaps for each character. To manage this, the GDI engine uses an **RFONT** structure.

An RFONT structure describes a distinct realization of a font, or a particular instance of a font. It's not a logical font, not a physical font. It's a collection of character glyphs, made to the order of a logical font, based on a generic description from a font file. The first part of an RFONT structure is documented in DDK as an FONTOBJ structure, to allow quick access from graphic drivers. The remaining RFONT fields can be accessed only through limited FONTOBJ methods—for example, FONT OBJ_cGetGlyphHandles, FONTOBJ_cGetGlyphs.

The GDI debugger extension provides the **fo** command to decode an RFONT structure. Here comes the massive RFONT structure:

```
typedef struct
{
    void *      pgdNext;
    void *      pgdThreshold;
    void *      pjFirstBlockEnd;
    void *      pdblBase;
    ULONG       cMetrics;
    ULONG       cjbbI;
    ULONG       cBlocksMax;
    ULONG       cBlocks;
    ULONG       cGlyphs;
    ULONG       cjTotal;
    void *      pbblBase;
    void *      pbblCur;
    void *      pgbNext;
    void *      pgbThreshold;
    void *      pjAuxCacheMem;
    void *      cjAuxCacheMem;
    ULONG       cjGlyphMax;
    ULONG       bSmallMetrics;
    ULONG       iMax;
    ULONG       iFirst;
    ULONG       cBits;
} CACHE;

struct RFONT
{
    FONTOBJ     fobj;          // 000
    ULONG       iUnique;        // 02c
    ULONG       flType;         // 030
    ULONG       ulContent;      // 034
    PVOID       hdevProducer;   // 038
    ULONG       bDeviceFont;    // 03c
```

```
PVOID    hdevConsumer;      // 040
DHPDEV   dhpdev;          // 044
PFE *    ppfe;            // 048
PFF *    ppff;            // 04c
FD_XFORM fdx;             // 050
ULONG    cBitsPerPel;     // 060
MATRIX   mxWorldToDevice; // 064
ULONG    iGraphicsMode;   // 0a0
FLOATOBJ eptflNtoWScale_x_i; // 0a4
FLOATOBJ eptflNtoWScale_y_i; // 0ac
ULONG    bNtoWIdent;      // 0b4
MATRIX * xoForDDI_pmx;    // 0b8
ULONG    xoForDDI_ulMode; // 0bc
unsigned unk_000;          // 0c0
MATRIX   mxForDDI;        // 0c4
ULONG    flRealizedType;  // 100
POINT    ptlUnderline1;   // 104
POINT    ptlStrikeOut;    // 10c
POINT    ptlULThickness; // 104
POINT    ptlSOThickness; // 10c
ULONG    ICharInc;
FIX      fxMaxAscent;
FIX      fxMaxDescent;
FIX      fxMaxExtent;
POINTFX  ptxfMaxAscent;
POINTFX  ptxfMaxDescent;
ULONG    cxMax;
LONG    lMaxAscent;
LONG    lMaxHeight;
ULONG    cyMax;
ULONG    cjGlyphMax;
FD_XFORM fdxQuantized;
LONG    lNonLinearExtLeading;
LONG    lNonLinearIntLeading;
LONG    lNonLinearMaxCharWidth;
LONG    lNonLinearAvgCharWidth;
ULONG    ulOrientation;
FLOATOBJ pteUnitBase_x;
FLOATOBJ pteUnitBase_y;
FLOATOBJ efWtoDBase;
FLOATOBJ efDtoWBase;
LONG    lAscent;
FLOATOBJ pteUnitAscent_x;
FLOATOBJ pteUnitAscent_y;
FLOATOBJ efWtoDAscent;
```

```
FLOATOBJ    efDtoWAscent;
LONG        lEscapement;
FLOATOBJ    efWtoDEsc;
FLOATOBJ    efDtoWEsc;
FLOATOBJ    efEscToBase;
FLOATOBJ    efEscToAscent;
ULONG       flInfo;
ULONG       hgBreak;
ULONG       fxBreak;
void *      pfdg;
void *      wcgp;
ULONG       cSelected;
RFONT *     rflPDEV_prfntPrev;
RFONT *     rflPDEV_prfntNext;
RFONT *     rflPFF_prfntPrev;
RFONT *     rflPFF_prfntNext;
void *      hsemCache;
CACHE       cache;
POINT      ptlSim;
ULONG       bNeededPaths;
FLOATOBJ   efDtoWBase_31;
FLOATOBJ   efDtoWAscent_31;
TEXTMETRICW * ptmw;
LONG        lMaxNegA;
LONG        lMaxNegC;
LONG        lMinWidthD;
ULONG       blsSystemFont;
ULONG       flEUDCState;
RFONT *     prfntSystemTT;
RFONT *     prfntSysEUDC;
RFONT *     prfntDefEUDC;
void *      paprftFaceName;
void *      aprfntQuickBuff[8];
ULONG       bFilledEudcArray;
ULONG       ulTimeStamp;
ULONG       uiNumLinks;
ULONG       bVertical;
ULONG       pchKernelBase;
ULONG       iKernelBase;
};
```

With such complicated data structures, we're assured that the GDI engine is sparing no effort to make text rendering efficient.

Other GDI Objects in the GDI Engine

So far we've discussed the representation of several major GDI objects in the GDI engine, which resides in kernel address space. More specifically, we've covered in quite some depth device context, device-dependent bitmap, DIB section, brush, pen, palette, region, path, logical font, physical, and realized font.

The GDI kernel debugger extension lists many more GDI objects in its **dumphmgr** command—for example, DD_DRAW_TYPE, CLIOBJ_TYPE, and SPOOL_TYPE. DirectDraw related objects will be covered in the next section. Other objects will not be covered, either because they are not so important to Win32 programming, or because we don't know if they have already become extinct as the Windows OS evolves, or because we don't have a way to create an instance of them.

We will see that what we have learned about the GDI internal data structure will deepen our understanding of Win32 GDI API programming.

[< BACK](#) [NEXT >](#)

3.9 DIRECTDRAW DATA STRUCTURE

Give me a handle, and I'll show you a data structure. This is what we achieved so far with GDI object handles. We've been able to show that there is a systemwide GDI object table, for some objects there are data structures the GDI keeps in user mode address space, and for all objects there are data structures the GDI engine keeps in the kernel mode address space. We are becoming very good at investigating the undocumented link between GDI and DDI with the help of the GDI kernel debugger extension.

Now enters DirectDraw, an API of the COM (Component Object Model) age. There are no more visible handles when you create a DirectDraw object, or a DirectDraw surface. You only get back interface pointers defined as LPDIRECTDRAW and LPDIRECTDRAWSURFACE, not handles like HDIRECTDRAW, HDIRECTSURFACE. What can we do about them?

Conceptually, a COM interface is a group of semantically related functions that provide access to a COM object. On the implementation level, a COM interface is a virtual function table, which stores the addresses of some semantically related functions. A COM interface pointer is normally defined as a pointer to a COM interface. But in reality, a COM interface pointer is not just any pointer to a COM interface. A COM interface pointer must be a pointer to COM object, or an instance of a COM class.

Let's try the following example with DirectDraw COM object creation:

```

HRESULT DirectDrawTest(HWND hWnd)
{
    LPDIRECTDRAW lpdd;

    HRESULT hr = DirectDrawCreate(NULL, &lpdd, NULL);
    if ( hr == DD_OK )
    {
        lpdd->SetCooperativeLevel(hWnd, DDSCL_NORMAL);

        DDSURFACEDESC ddsd;
        ddsd.dwSize = sizeof(ddsd);
        ddsd.dwFlags = DDSD_CAPS;
        ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE;

        LPDIRECTDRAWSURFACE lpddsprimary;

        hr = lpdd->CreateSurface(&ddsd, &lpddsprimary, NULL);
        if ( hr == DD_OK )
    }
}

```

```
{  
    char mess[MAX_PATH];  
  
    wsprintf(mess,  
        "DirectDraw object at %x, vtable at %x\n"  
        "DirectDraw surface object at %x, vtable at %x",  
        lpdd, * (unsigned *) lpdd,  
        lpddsprimary, * (unsigned *) lpddsprimary);  
    MessageBox(NULL, mess, "DirectDrawTest", MB_OK);  
  
    lpddsprimary->Release();  
}  
lpdd->Release();  
}  
  
return hr;  
}
```

The routine shown above creates a DirectDraw object and then a DirectDraw surface object and displays their addresses and their virtual function table pointers. If you put the code into a program and run it, a message box will pop up that displays something like this:

```
DirectDraw object at 7e2a10, vtable at 728405a0  
DirectDraw surface object at 7e3b58, vtable at 72840940
```

If the program is started under a debugger, you can verify that the two objects are created off heap in user address space, and the virtual function tables both point to the DirectDraw implementation module ddraw.dll. If you can spend a few more moments, you can even find out the addresses of the functions in the virtual tables and their symbolic names. For example, here is part of the DirectDraw object's virtual function table:

```
7298E8A4: ddraw.dll!DD_QueryInterface  
7298EB48: ddraw.dll!DD_AddRef  
7298EC16: ddraw.dll!DD_Release  
72980C5A: ddraw.dll!DD_Compact  
7297C82B: ddraw.dll!DD_CreateClipper  
...
```

Got it? COM uses the function address, and the function table too, in a way very similar to, but much more formalized than, the DDI interface between the GDI engine and graphical drivers.

Now let's check what impact using DirectDraw has on the GDI object table, using our faithful GDI kernel debugger extension, under the control of our home-grown debugger extension host. Run the

dumpdd command twice, once before running the above code, and the second time when the message box is still on, which means the DirectDraw objects are not released. Guess what—we've discovered two new species of object, one of DD_DIRECTDRAW_TYPE, another of DD_SURFACE_TYPE. GDI is still using handles to implement DirectDraw, although they are well hidden behind those interface pointers.

Apparently, DD_DIRECTDRAW_TYPE is meant to be a DirectDraw object, while DD_SURFACE_TYPE is meant to be a DirectDraw surface object. Let's look at a DirectDraw object first.

You can use the **dumpddobj DDRAW** command to list all DirectDraw objects. Their kernel mode data structure can be decoded by the **dddlocal** command, which names the structure as EDD_DIRECTDRAW_LOCAL. The GDI engine makes a distinction between a global, systemwide DirectDraw data structure and per-process DirectDraw data structure. Here is what the EDD_DIRECTDRAW_LOCAL structure looks like:

```
// Windows 2000, 72 bytes
typedef struct
{
    HGDIOBJ          hHmgr;           // 000
    void *           pEntry;          // 004
    ULONG            cExcLock;        // 008
    ULONG            Tid;             // 00c
    EDD_DIRECTDRAW_GLOBAL * peDirectDrawGlobal; // 010
    EDD_DIRECTDRAW_GLOBAL * peDirectDrawGlobal2; // 014
    EDD_SURFACE *     peSurface_DdList; // 018
    unsigned          unk_01c[2];       // 01c
    EDD_DIRECTDRAW_LOCAL * peDirectDrawLocalNext; // 024
    FLATPTR          fpProcess;        // 028
    FLONG             fl;              // 02c
    HANDLE            UniqueProcess;   // 030
    PEPPROCESS        Process;         // 034
    unsigned          unk_038[2];       // 038
    void *           unk_040;          // 040
    unsigned          unk_044;          // 044
} EDD_DIRECTDRAW_LOCAL;
```

The EDD_DIRECTDRAW_LOCAL structure stores a process identifier in the UniqueProcess field. The Process field stores the pointer to the process's kernel object. Furthermore, a DirectDraw object is bound to the creating thread through the Tid field, unlike most other GDI objects, whose Tid field is normally 0.

The GDI engine also keeps one copy of global data structure, managing system wide DirectDraw state information in an EDD_DIRECTDRAW_GLOBAL structure. Here we see two pointers to this

structure. One DirectDraw process normally creates multiple DirectDraw surfaces. Their kernel objects are chained into a linked list starting from the peSurface_DdList field. All DirectDraw objects currently in the system are also chained into a linked list, through the peDirectDrawLocalNext field.

The EDD_DIRECTDRAW_LOCAL is the root of all DirectDraw-related objects within a process, which also links to systemwide DirectDraw-related objects. This whole web of DirectDraw structures makes coordination among them possible.

Now let's take a look at the EDD_DIRECTDRAW_GLOBAL structure, which can be decoded by the **ddglobal** command.

```
// Win 2000, 476 (0x1DC) bytes
typedef struct
{
    HDEV      hdev;          // 0x000
    unsigned   unk_004;        // 0x004
    SPRITE *  pListZ;        // 0x008
    SPRITE *  pListY;        // 0x00c
    SURFOBJ * psoScreen;     // 0x010
    unsigned   unk_014[9];      // 0x014
    FLONG     flOriginalSurfFlags; // 0x038
    ULONG     iOriginalType;    // 0x03c
    unsigned   unk_040[5];      // 0x040
    SPRITESCAN * pRange;       // 0x054
    void *    pRangeLimit;     // 0x058
    SURFOBJ *  psoComposite;   // 0x05c
    unsigned   unk_060[66];      // 0x060
    REGION *  prgnUnlocked;    // 0x168
    unsigned   unk_16c[28];      // 0x16c
} SPRITESTATE;

// Win 2000, 1552 (0x610) bytes
typedef struct
{
    void *      dhpdev;        // 0x000
    DWORD       dwReserved1;    // 0x004
    DWORD       dwReserved2;    // 0x008
    unsigned   unk_00c[3];      // 0x00c
    LONG        cDriverReferences; // 0x018
    unsigned   unk_01c[3];      // 0x01c
    LONGLONG    llAssertModeTimeout; // 0x028
    DWORD       dwNumHeaps;     // 0x030
    VIDEOMEMORY * pvmList;     // 0x034
    DWORD       dwNumFourCC;    // 0x038
```

```
DWORD * pdwFourCC; // 0x03c

DD_HALINFO ddhalinfo; // 0x040
unsigned unk_1e0[44]; // 0x1e0
DD_CALLBACKS ddcallbacks; // 0x290
DD_SURFACECALLBACKS ddsurfacecallbacks; // 0x2c4
DD_PALETTECALLBACKS ddpalettecallbacks; // 0x304 ?
unsigned unk_314[48]; // 0x314
D3DNTHAL_CALLBACKS d3dnthalcallbacks; // 0x3d4
unsigned unk_460[7]; // 0x460
D3DNTHAL_CALLBACKS2 d3dnthalcallbacks2; // 0x47c
Unsigned unk_498[18]; // 0x498
DD_MISCCELLANEOUSCALLBACKS ddmiscellaneouscallbacks; // 0x4e0
unsigned unk_4ec[18]; // 0x4ec
D3DNTHAL_CALLBACKS3 d3dnthalcallbacks3; // 0x534
unsigned unk_54c[23]; // 0x54c

EDD_DIRECTDRAW_LOCAL * peDirectDrawLocalList; // 0x5a8
EDD_SURFACE * peSurface_LockList; // 0x5ac
FLONG fl; // 0x5b0
ULONG cSurfaceLocks; // 0x5b4
PKEVENT pAssertModeEvent; // 0x5b8
EDD_SURFACE * peSurfaceCurrent; // 0x5bc
EDD_SURFACE * peSurfacePrimary; // 0x5c0
BOOL bSuspended; // 0x5c4
unsigned unk_5c8[12]; // 0x5c8
RECTL rcBounds; // 0x5f8
HDEV hdev; // 0x608
unsigned unk_60c; // 0x60c
} EDD_DIRECTDRAW_GLOBAL;
```

The EDD_DIRECTDRAW_GLOBAL essentially holds all the information the GDI engine needs to know about the display driver for DirectDraw support. The dhpdev field is a handle to the device driver's PDEV structure returned by DrvEnable PDEV call. It's normally a pointer to the driver's private physical device data structure.

The EDD_DIRECTDRAW_GLOBAL structure stores several structures that the GDI engine retrieved from the display driver. The ddhalinfo structure holds the DD_HALINFO structure returned by DrvGetDirectDrawInfo, which describes capabilities of the hardware and driver. The ddcallbacks, ddsurfacecallbacks, and dd palettecallbacks fields hold the DD_CALLBACKS, DD_SURFACECALLBACKS, and DD_PALETTECALLBACKS structures returned by DrvEnableDirectDraw. There are also callback structures for DirectDraw 3D functions. These structures inform the GDI engine of the DirectDraw entry points the driver supports. So the GDI engine knows which routines to call to create a surface, set a color key, map video memory address, or flip a surface, etc.

EDD_DIRECTDRAW_GLOBAL also stores other interesting information, such as a list of DirectDraw objects, list of locked surfaces, pointer to current surface, etc.

The EDD_DIRECTDRAW_GLOBAL structure is a part of the PDEV_WIN32K structure we talked about in [Section 3.7](#). The PDEV_WIN32K structure also contains a SPRITESTATE structure.

Now that we understand how global and per-process DirectDraw information is kept in the GDI engine, let's turn our attention to what's behind a DirectDraw surface.

Each DirectDraw surface object has a corresponding object that is hidden from the user. Those objects are listed as DD_SURF_TYPE by the **dumpddobj** command. You can use the GDI extension command **dumpddobj SURF** to list all DirectDraw surface handles. For an individual surface handle, you can use **dddsurface** to look at its kernel address space data structure EDD_SURFACE.

```
typedef struct
{
    HGDIOBJ          hHmgr;           // 000
    void *            pEntry;          // 004
    ULONG             cExcLock;        // 008
    ULONG             Tid;             // 00c

    DD_SURFACE_LOCAL  ddsurfacelocal;  // 010
    DD_SURFACE_MORE   ddsurfacemore;   // 04c
    DD_SURFACE_GLOBAL ddsurfaceglobal; // 068
    DD_SURFACE_INT    ddsurfaceint;    // 0b4

    EDD_SURFACE *      peSurface_DdNext; // 0b8
    EDD_SURFACE *      peSurface_LockNext;
    unsigned           unk_0c0;          // 0c0
    EDD_DIRECTDRAW_GLOBAL * peDirectDrawGlobal;
    EDD_DIRECTDRAW_LOCAL * peDirectDrawLocal;
    FLONG              fl;
    unsigned           unk_0d0;          // 0d0
    ULONG              iVisRgnUniqueness;
    unsigned           unk_0d8;
    HANDLE             hSecure;
    unsigned           unk_0e0;
    HBITMAP            hbmGdi;          // 0e4
    unsigned           unk_0e8;
    ERECTL             rclLock;         // 0ec
    unsigned           unk_0fc[3];       // 0fc
} EDD_SURFACE;
```

In an EDD_SURFACE structure, after the normal GDI kernel object header, you can find four structures documented in Windows 2000 DDK, DD_SURFACE_LOCAL, DD_SURFACE_MORE, DD_SURFACE_GLOBAL, and DD_SURFACE_INT. The DD_SURFACE_GLOBAL structure contains information shared by multiple surfaces, such as pitch, height, width, and x/y coordinate of the surfaces. The DD_SURFACE_LOCAL contains surface-related data that is unique to an individual surface object, such as whether it's front buffer or back buffer, color keys, pixel format, surfaces attached and attached to. The DD_SURFACE_MORE structure contains extra per-surface information like pointer to video port information and overlay flags. The last structure, DD_SURFACE_INT, just contains a pointer to the DD_SURFACE_LOCAL structure.

After those documented DirectDraw surface structures come pointers to the next surface, DirectDraw global and local data. The hbmGdi field sometimes holds a GDI DDB handle.

Now we know quite a few DirectDraw kernel data structures; how are they used? A DirectDraw rendering call normally starts from a DirectDraw surface interface pointer—for example, Flip and Blt. From the surface interface pointer, the GDI finds the DD_SURF_TYPE GDI object handle and passes it to the GDI engine. The GDI engine locates the EDD_SURFACE structure and gets a pointer to EDD_DIRECTDRAW_GLOBAL structure, which has a DD_SURFACECALLBACKS structure in it. The DD_SURFACECALLBACKS structure stores a pointer to the display driver's entry-point handling surface flipping entry point, which will be called by the DirectDraw engine. The flipping routine accepts a DD_FLIPDATA structure, which can be assembled with information from destination and source EDD_SURFACE structures. For details, check DDK descriptions on DdFlip.

Prior to the final release of Windows 2000 (build 2195), DirectX shares the same handle table with GDI. The GDI kernel debugger extension command **dumphmgr** lists DirectX object together with normal GDI objects. The internal object type for DirectDraw object is 0x02, and for DirectDraw surface is 0x03.

But in the official final release of Windows 2000, Microsoft moves DirectX objects out of the GDI object handle manager to the new DirectX handle manager. New commands, **dumpdd** and **dumpddobj**, are added to the GDI debugger extension. The DirectX handle manager manages six types of objects: deleted object, DirectDraw object, DirectDraw surface object, Direct3D device object, DirectDraw video port object, and DirectDraw motion compensation object. According to these new commands, the DirectX handle manager manages 1,024 DirectX handles in a 16-KB table, a scaled down version of the GDI handle manager that manages 16,384 handles in a 256-KB table. It is not known whether the DirectX handle table can grow. Also unknown at this stage is whether the DirectX handle table is mapped to user mode address space, as the GDI handle table.

Separating DirectX objects from the GDI objects is definitely a good move, as it makes sure DirectX applications do not interfere with GDI-based applications in competing for limited GDI object handles.

3.10 SUMMARY

Finally, we have finished our exploration of the internal data structure behind the GDI and DirectDraw, surfacing with lots of deep understanding of how the GDI and the Windows graphics engine organize data structures internally. This chapter started with the simple goal to understand what a GDI object handle is. Then we found the GDI object table, figuring out how to locate the table and decode the table and the user mode data structures behind several types of GDI objects. The most important data structures GDI keeps are in kernel mode address space. To be able to read them, we developed a simple kernel mode spying driver, Periscope, fostering a GDI debugger extension in our own host program so that we can use its expertise to help us decode GDI's kernel data structure. The GDI debugger extension proves to be very helpful in exposing the otherwise totally hidden GDI kernel data structures.

From now on, for every GDI object handle, you should have a much clearer picture of how data are stored internally in the GDI, what resources are needed, and how approximation is done. You should also have a general idea of how data is transformed in the GDI engine and eventually consumed by graphic device drivers like display drivers and printer drivers. [Chapter 7](#) of this book contains a simple tool which uses knowledge developed in this chapter to summarize per-process GDI object usage.

Give me a GDI handle, I can show you a GDI data structure. Give me a Direct Draw interface pointer, I can show you a DirectDraw data structure. Now you're certified to say that.

Further Reading

The chapter is mainly on the Windows NT/2000 graphics system internal data structure. If you're interested in GDI internals for Windows 3.1 and Windows 95, Matt Pie trek's books are the only good references. Chapter 5 of his *Windows Internals*, published in 1993, covers Windows 3.1 GDI in 35 pages. Chapter 4 of his *Windows 95 System Programming Secrets*, published in 1995, covers both Windows 95 USER and GDI subsystems. Matt provides lots of details on how the GDI object table is organized in Windows 95 and even how to access it from the Win32 program, which is of course in assembly language or psuedoocode.

We used a small amount of the kernel mode device driver to spy on kernel address space. Books like *Windows NT Device Driver Development*, by Peter G. Viscarola and Anthony Mason, provide full coverage on how to develop Windows NT/2000 kernel mode device drivers. The Web page www.sysinternals.com is a good source on system-level GUI-based tools which require kernel mode drivers.

The latest versions of WinDbg and the GDI debugger extension can be down loaded from Microsoft, either from the Platform SDK page or the DDK page. By combining debug symbols files for the OS,

the GDI debugger extension, and our host program Fosterer, you can explore the GDI internal data structure on your own, or even customize the Fosterer program to probe other components of the operating system. For this purpose, the Microsoft DDK is a well-written document.

Sample Programs

The programs in [Chapter 3](#) (see [Table 3-14](#)) are not normal graphics programming examples, not even normal Windows programs; they are system-level tools to help us understand the Windows graphics system internal data structure and the Windows operating system internals in general. Make good use of them.

Table 3-14. Sample Programs for [Chapter 3](#)

Directory	Description
Samples\Chapt_03\Handles	GDI handles program, decoding GDI handles, locating the GDI object table, and decoding the GDI object table.
Samples\Chapt_03\QueryTab	Sample program for accessing the GDI object table in your program.
Samples\Chapt_03\Periscope	Kernel mode device driver allowing access to kernel address space data from the user address space program using file operations.
Samples\Chapt_03\TestPeriscope	Sample program for accessing kernel address space in your program.
Samples\Chapt_03\Fosterer	Host program for the GDI kernel debugger extension DLL, mother ship for exploring the GDI/DirectDraw kernel data structure.

Chapter 4. Spying in the Windows Graphics System

Seeing is believing. To be able to see the way things really happen makes a big difference in understanding things around us. A microscope brings to your attention tiny creatures, while a telescope brings to your attention things from far away. And, television shortens the distance between people living in different parts of world.

In Windows programming, programmers want to see what is really happening between programs and the operating system. [Chapter 2](#) gave us an architectural view of the Windows graphics system; [Chapter 3](#) exposed a data-structural view. But we still miss the dynamics of millions of calls that are happening in the system. Where do things start? Where do things end? Are things always running smoothly internally? Or are there accidents, violations, roadblocks, and leakages that are just not noticeable outside?

This chapter will arm programmers with API spying techniques and tools to allow understanding of the dynamic behavior of Win32 API calls, especially Win32 GDI/DirectDraw calls, graphics system service calls, and the DDI interface.

[Section 4.1](#) develops a generic Win32 API spying framework, which includes a DLL that is injected into a target process and a control program. [Section 4.2](#) extends the generic spying program to eavesdrop on every GDI call within a process. [Section 4.3](#) turns attention to COM interfaces used in DirectDraw, [Section 4.4](#) illustrates the hooking of GDI system services calls, and finally [Section 4.5](#) dives into kernel mode again to monitor the DDI interface.

4.1 SPYING ON WIN32 API CALLS

API hooking and spying is not an unusual practice in Windows programming. Professional and custom-made programming tools often use these techniques to get every pulse of system operation.

The best-known tool that uses API hooking and spying is Numega's Bounds Checker, an advanced error-detection package for Windows. BoundsChecker is able to detect Windows API errors, COM/OLE interface errors, memory errors, pointer and leak errors, and program faults. For example, BoundsChecker can detect function failure, invalid parameters, unimplemented functions, memory overrun, stack overrun, uninitialized memory, an array index out of range, a memory leak, a resource leak, an interface leak, etc. One of the basic techniques BoundsChecker uses is spying on thousands of Windows API calls. Calls to Windows API are taken over by Bounds Checker, so that before the call it can verify the parameters, stack information can be logged, and after the call the return value will be checked before returning to the application. When running a program under BoundsCheker, it acts as a debugger to the program, so BoundsChecker DLLs can be injected into the application's process address space to take control. When BoundsChecker is integrated with your compiler, your source code is instrumented to make calls to BoundsChecker DLLs. In either case, when you call a Win32 API, BoundsChecker gets it first.

Microsoft System Journal often carries articles on using API hooking and spying to support the mouse wheel, detect a memory allocation program in COM programs, or figure out what is causing a deadlock in a multithreading program. Microsoft even provides an API spying tool, APIMON, with platform SDK and Windows resource kits.

Hooking and spying is most easily done in user mode code, but kernel mode hooking and spying is also possible. The www.sysinternals.com Web site has quite a few tools that basically hook into the Windows NT kernel mode file-system hierarchy or device driver chain to monitor the file system, registry, and port access. With Windows 2000, even Microsoft acknowledges the need to hook into the display driver by officially supporting a mirroring driver for display drivers. You can imagine that Microsoft is constantly receiving complaints about why a user can't display a Windows screen on a remote machine easily. Now, with a mirroring driver, you can send another stream of data across the network without touching the display driver.

Commercial tools and Microsoft spying tools, or sample code you get from other sources, is not likely to solve all your API spying and hooking needs, not if you want an easy-to-use, customizable, expandable, powerful tool at your disposal. Here are some limitations of the tools you may have seen.

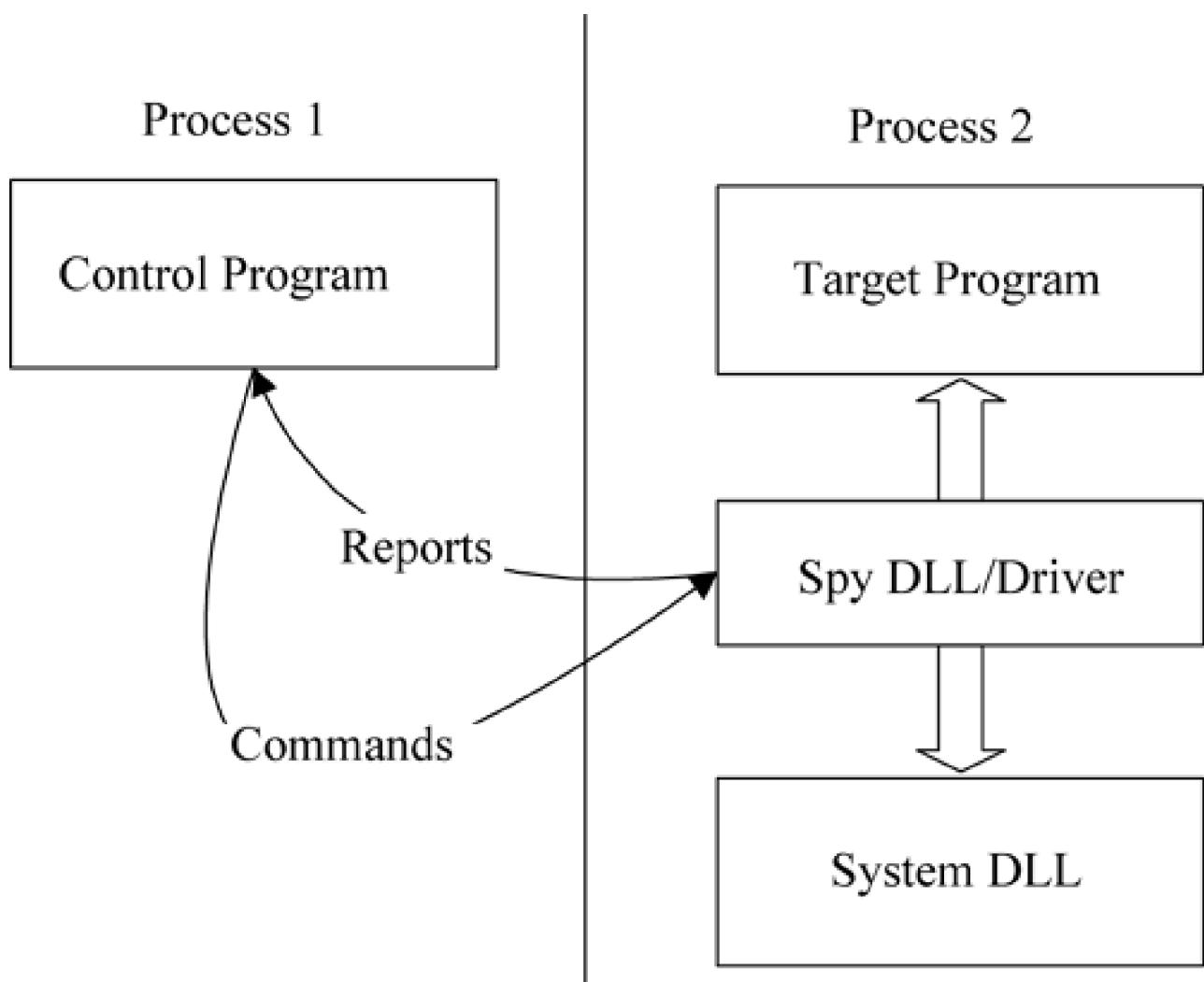
- Data type handling. Ready available tools handle fixed sets of data types, while data types in Windows programming get updated very often. It would be nice if a spying tool could translate binary raster operation code into names like SCRCOPY, dump bitmaps into files, or show whether a GDI handle is a logical pen handle.
- Performance measurement. Being able to measure how much time is spent on a Win32 API call can help you use the right features and avoid the unsuitable.
- Undocumented API, intramodule API calls, system service calls, kernel mode calls. Support for these features may be the weakness of existing tools.

If you want a deep knowledge of an area of Windows programming—for example, Windows graphics programming—an API spy under your control is a good investment.

Making of a Spy

A Windows spy program is normally made of two parts: a control program and a spy DLL or driver. The control program sends the spy DLL/driver to the right place to work, commands it, and possibly receives reports from the spy. The spy DLL/driver gets landed in a user process to spy on, hooks itself at specific places, gathers tiny pieces of information that are important to the problem at hand, and acts on them according to its task, or reports back to the control program. [Figure 4-1](#) illustrates a spy program at work.

Figure 4-1. Components of a spy program.



There are certainly variations to this general model. If you can find a reliable way to plant your spy, and your spy can act on its own, then maybe you don't need a control program. Certain double-byte-character environments live on top of the normal English Windows system. Instead of having to inject DLLs into every application having a graphical user interface, they just rename the system DLLs and replace them with their own versions to provide double-byte support on a single-byte system. If you want to spy on activities in kernel mode address space, you definitely need a kernel mode device driver—in other words, a kernel mode spy DLL. Your control program then needs to install the driver and control the driver. The Fosterer program developed in [Chapter 3](#) uses a kernel mode driver, Periscope, to read data from the kernel address space to help you understand kernel graphics system data structures. SoftICE/W, which is a system-level debugger from Numega, also relies on a kernel mode driver to provide seamless systemwide debugging features on a single machine.

There are several things you have to conquer when implementing an API spy program:

- Inject the spy DLL into target process.
- Hook into the API calling chain.
- Gather parameters, return values, and timing information.
- Dump data into a readable format.
- Set up a user interface to allow user setting of programs or modules to spy on, and Win32 API functions or COM methods to hook into.

In this section we will develop a general-purpose Win32 API spying program called Pogy. It's named after a navy submarine that has been involved in undersea scientific explorations. We want our Pogy to help us explore Windows operating system.

Pogy's control program is Pogy.exe, which uses several property-sheet pages as its user interface. Pogy's spying DLL is Diver.dll. Now let's go through its design briefly.

Injecting Spying DLL

The Win32 API supports installation of system-wide, or per-thread hooks, to monitor or modify system message-handling behavior. The API responsible for installing hooks is `SetWindowsHookEx`. Windows 2000 even expands the class of hooks you can install to 15. You can install a `WM_GETMESSAGE` hook, which monitors messages posted to message queues; or a `WH_SHELL` hook, which receives notification on creation and destruction of top-level windows.

A hook is normally implemented in a DLL, a requirement for a system-wide hook. The reason is that for a hook to function in other processes, its code needs to be loaded into the target process's address space. An executable file can only be loaded as data into another process, so a systemwide hook must be in a DLL.

Once a hook DLL is loaded into a process's address space, there is nothing to prevent it from pulling all sorts of stunts there. This is the foundation of some API hooking or spying techniques. Location, location, location—you may have heard your real estate agent mention it. You simply have to be at the right place.

`SetWindowsHookEx` is just one way of injecting a DLL into a target process. But it's an easy and well-documented method. The simplest painless way to inject a DLL into every process is to add your DLL to the following key on Windows NT/2000:

HKEY_LOCAL_MACHINE\Software\Microsoft
Windows NT\CurrentVersion\Windows\ApplInit_DLLs

Knowing unique ways to inject a DLL into a foreign process is a good measure of your advanced Windows programming knowledge. Matt Pietrek's classic *Windows 95 System Programming Secrets* shows a way to use the Win32 debugger API and dynamic patching of target process code to inject a DLL. Jeffery Richter's *Programming Applications for Microsoft Windows*, fifth edition, shows how to use a remote thread to do the same thing.

For our Pogy program, we use SetWindowsHookEx to set up a system-wide shell hook. A shell hook is an application-defined callback routine. Once a shell hook is registered with system, a systemwide shell hook will be called when certain shell-related events happen in the system, while a non systemwide shell hook is responsible only for a single thread. The hook procedure ShellProc is implemented in the spy DLL Diver .dll, as required for a systemwide hook. Diver exports a routine SetupDiver, callable from the control program Pogy.exe to perform installation, uninstallation, and setting up of communication between the two. Here is the hook part on the spy DLL side:

```
#pragma data_seg("Shared")
HWND h_Controller = NULL;
HHOOK h_ShellHook = NULL;
#pragma data_seg()
#pragma comment(linker, "/section:Shared,rws")

LRESULT CALLBACK ShellProc(int nCode, WPARAM wParam,
                           LPARAM lParam)
{
    if (nCode==HSHELL_WINDOWCREATED)
        if (...)

            StartSpy();

    assert(h_ShellHook);

    if (h_ShellHook)
        return CallNextHookEx(h_ShellHook, nCode, wParam, lParam);
    else
        return FALSE;
}

void _declspec(dllexport) SetupDiver(int nOpt, HWND hWnd)
{
    switch (nOpt)
    {
        case Diver_Install:
            assert(h_ShellHook==NULL);
            h_ShellHook = SetWindowsHookEx(WH_SHELL, (HOOKPROC)
                                          ShellProc, hInstance, 0);
            h_Controller = hWnd;
            break;

        case Diver_UnInstall:
            assert(h_ShellHook!=NULL);
            UnhookWindowsHookEx(h_ShellHook);

            h_ShellHook = NULL;
            break;
    }
}
```

}

A systemwide hook is registered with system (the window manager) only once; when SetWindowsHookEx is called, a handle to the hook is returned. The hook handle needs to be used in the hook procedure and finally to delete the hook by calling UnhookWindowsHookEx. Here comes a problem: A systemwide hook can be loaded into multiple process address spaces, which are normally hidden from each other, so where can the handle be stored? The answer is a shared data section within the DLL where the hook procedure is defined. A normal read/write data section in a Win32 EXE is private to the loading process; that is, each process will have its own copy of it. But a shared data section is special in that it's shared by all processes loading the DLL. In the code shown above, the two data_seg pragmas are used to mark the beginning and end of a named section, and the comment (linker) pragma is used to tell the linker to treat the section as readable, writable, and shared ("rws"). We put the hook handle and a window handle into the shared data section. Please note that data in a shared section needs to be initialized data.

The control program Pogy.exe is linked with the same spying DLL Diver.dll. When it's loaded, Pogy creates a window for communicating with the spy. It calls Setup Diver(Diver_Install,...) to inform the spy DLL of its window handle and let it create the shell hook. Call to SetWindowsHookEx returns a hook handle which is needed in calling the next hook in a hook chain. The spy DLL needs a place to hold both the controller's window handle and the hook handle, such that they can be accessed in all user processes. So once h_ShellHook and h_Controller are set, they can be accessed in every process.

But now, Diver.dll is only loaded into the controller's process. Only when a top-level window gets created or destroyed does a shell procedure get called. If a process other than the controller does that, operating system figures out that it needs to call a hook procedure not in its process. So the DLL hosting the hook procedure needs to be loaded into that process. After that, ShellProc in the loaded DLL with the process is called with HSHELL_WINDOWCREATED. ShellProc communicates with the controller program to determine if API spying should be started. Its real hook duty as required by the operating system is just not to forget to call the possible next hook in the chain, through CallNextHookEx. The SetupDiver also provides an option to un install the hook.

Hooking into API Calling Chain

Once the control program tells the spy DLL to start working, it initializes itself and creates a hidden window. The handle of the window is passed back to the controller. So now the controller and the spy have a way to send messages to each other, through the window handles.

The Windows operating system allows simple messages with two 32-bit parameters, using application-defined messages starting with WM_USER. If you want to send a block of data across a process boundary, you can't just pass pointers. You can't assume a pointer in one address space is readable in another address space. Luckily, you can use WM_COPYDATA message to send a block of data. The Windows operating system makes a special effort to make sure block data in messages like WM_SETTEXT, WM_GETTEXT, and WM_COPYDATA are copied properly across the process boundary.

Once the controller notices a communication window has been created in the spy DLL, it sends the spy DLL a list of functions to spy on. Each function has a caller module name, callee module name, function name, number of parameters, types of parameters, and function return value type. For example, if a user wants to spy on the GDI function SetTextColor in CLOCK.EXE, the caller module is "CLOCK.EXE," the callee module is "GDI32.DLL," the function name is "SetTextColor," there are two parameters (HDC and COLORREF), and COLORREF is the return type. The spy DLL builds its internal table of modules and functions using the data it receives.

[Chapter 1](#) of this book discussed briefly the format of PE file, which is used to represent Win32 modules, both on disk and loaded in RAM. It mentioned how linking between modules, static or dynamic, uses export and import

directories, with one internal variable holding the address of each imported function. So to hook into a Win32 API call, all you have to do is find the location that holds the imported function's address in the module's import directory and overwrite it with your spying function's address. If you still want the program to function as normal, you should of course save the original address before overwriting it.

If you're spying on more than one function, you can't just change all their import addresses to point to a single spying function. The spying function at a minimum needs to know which function this call is for. In our implementation, we build a little stub routine for each entry in the function table. The stub function pushes a function index on the stack, before calling the universal spying function ProxyProlog. The address of each stub routine is used in patching the module's import directory. Here is what the stub routine looks like:

```
push index    // 68 xx xx xx xx  
jmp ProxyProlog // E9 yy yy yy yy
```

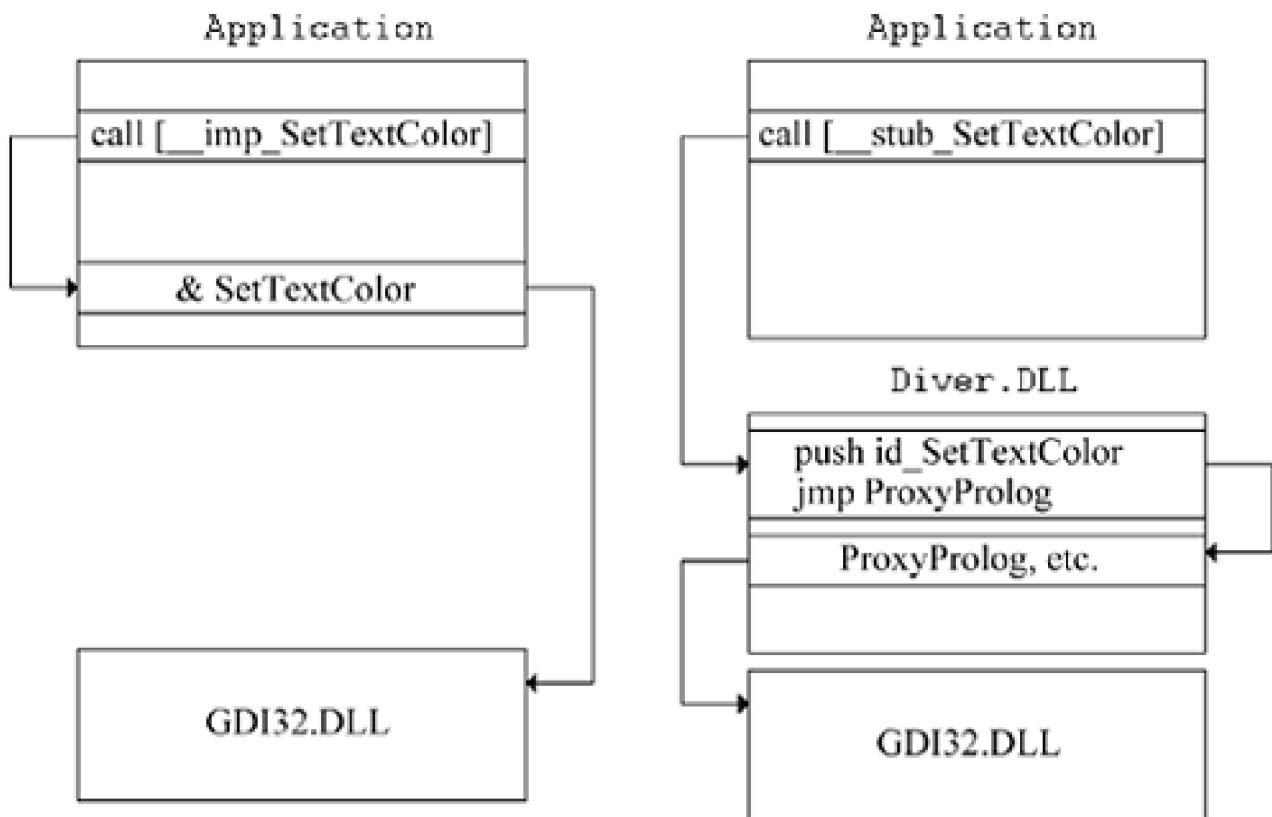
The ProxyProlog routine just needs to pop the index off the stack, and then use it to query the function table to get all its information.

NOTE

As a generic spying solution, we're avoiding any modification of registers. If you use a register to pass the index instead of the stack, your solution will not work on functions using that register to pass input parameters.

[Figure 4-2](#) illustrates the difference between calling a Win32 function before and after hacking using a stub function. The left side of the diagram shows the situation before the hacking; a variable in the import directory is used to make an indirect call to a Win32 API function. The right side shows the after-hacking scene. Now the application makes an indirect call to a stub function, which transfers control to a generic ProxyProlog routine in Diver.DLL. ProxyProlog, and related routines and data structures in Diver.DLL, are responsible to make sure that the original Win32 API function is called, and control is transferred to the caller when everything is done.

Figure 4-2. Hacking an API call through a stub function.



Information Gathering

For the functions we're spying on, ProxyProlog will get called before the real Win32 API functions get called. But ProxyProlog and routines related to it have a very tough task to do. They need to gather information about all the input parameters, record entering time, call the original API function, record returning time, record function return value, and finally return to the caller. As a basic rule for a spy, it should restore everything it touches. This means every register and CPU flag, short of rewinding CPU clock cycles.

Because of the complexity, the whole job is divided into several routines, using assembly language, C, and even C++ with virtual functions. Here is the spying team:

- ProxyProlog, implemented as a naked assembly routine, in the sense that the compiler is not going to add standard function entry and exit code for it. Its job is to save registers, record current time (time 1), call routine ProxyEntry, record time again (time 2), restore registers, and finally return control to the original Win32 API function the application is supposed to call.
- ProxyEntry, implemented as a C routine. Its job is to allocate a KRoutineInfo structure on a software stack, record basic information about this call, call a C++ virtual function KFuncTable::FuncEntryCallBack, patch the CPU stack to make sure that when the original Win32 API function returns Proxy Epilog gets called first, and patch the CPU stack again to make sure Proxy Prolog returns control to the original Win32 API function.
- KFuncTable::FuncEntryCallBack, implemented as a C++ virtual function. It does nothing in its minimum implementation. But it has all function-start timing and parameter information, so it can do performance measurement, parameter dumping, parameter validation, and even changing of parameters if we want.
- ProxyEpilog, implemented as a naked assembly routine, called immediately after the Win32 API function

returns. It saves registers, records time (time 3), calls routine ProxyExit, records time again (time 4), restores registers, and finally returns control to the original caller, finishing the task of spying on one API call.

- ProxyExit, implemented as a C routine. Its job is to pop up the KRoutineInfo structure on our software stack, call a C++ virtual function KFunc Table: :FuncExitCallBack, and patch the CPU call stack so ProxyEpilog will return to the original caller.
- KFuncTable::FuncExitCallBack, implemented as a C++ virtual function. It does nothing in its minimum implementation. But it has total function-start and end timing information and function return value to show its boss.

Here are the critical entrance parts, ProxyProlog and ProxyEntry.

```
typedef struct
{
    unsigned m_flag;
    unsigned m_edx;
    unsigned m_ecx;
    unsigned m_ebx;
    unsigned m_eax;

    unsigned m_funcid;
    unsigned m_rtnads;
    unsigned m_para[32];
} EntryInfo;

__declspec(naked) void ProxyProlog(void)
{
    // funcid, rtnadr, p1..pn
    // funcid also as callee address placeholder

    // save common registers, flags
    __asm push eax
    __asm push ebx
    __asm push ecx
    __asm push edx      // edx, ecx, ebx, eax
    __asm pushfd        // 4 bytes EFLAGS

    __asm _emit 0x0F      // time1
    __asm _emit 0x31
    __asm shr d eax, edx, 8 // EAX = EDX:EAX >> 8
    __asm push eax        // entering clock
    __asm sub eax, OverHead
    __asm push eax        // entering clock - overhead

    __asm lea   eax, [esp+8] // offset for flag on stack
    __asm push eax
```

```
__asm call ProxyEntry // C routine

__asm pop ecx // ecx = entering clock
__asm _emit 0x0F // time2
__asm _emit 0x31
__asm shr d eax, edx, 8 // EAX = EDX:EAX >> 8
__asm sub d eax, ecx // new overhead by ProxyEntry
__asm add d OverHead, eax

// restore common registers, flags
__asm popfd
__asm pop edx
__asm pop ecx
__asm pop ebx
__asm pop eax

// switch control to original callee
__asm ret // continues to original callee
}

void __stdcall ProxyEntry(EntryInfo *info, unsigned entertime)
{
    int id = info->m_funcid;

    assert(pStack!=NULL);
    KRoutineInfo * routine = pStack->Push();

    if ( routine )
    {
        routine->entertime = entertime;
        routine->funcid = id;
        routine->rtnaddr = info->m_rtnads;
        pFuncTable->FuncEntryCallBack(routine, info);

        // patch return address such that original function
        // will go to our epilog code before returning to caller
        info->m_rtnads = (unsigned) ProxyEpilog;
    }
    // make sure controls goes to original function
    // when ProxyProlog returns
    info->m_funcid = (unsigned)
        pFuncTable->>m_func[id].f_oldaddress;
}
```

Time measurement is achieved in the most accurate and efficient way on the Intel CPU, thanks to the Pentium's RDTSC instruction. RDTSC returns the number of clock cycles CPU has executed since the last time CPU got started, as a 64-bit value in EDX:EAX register. On a Pentium 200-MHz CPU, one clock cycle is 5 nanoseconds.

Managing 64-bit values is not so convenient, so the code here shifts the EDX :EAX register pair 8 bits to the right

and uses only the lower 32-bit value. The smallest duration we can measure is now $5 * 2^8 = 1280$ nanoseconds, still much more accurate than the millisecond accuracy you can get with GetTickCount. A 32-bit value in 1.28-microsecond accuracy lasts 1.58 hours, enough for normal testing.

For a single API call, the code reads clock cycle count 4 times, before calling Proxy Entry, before calling the hooked API, before calling ProxyExit, and before returning to the original caller. The time spent between time 1 and time 2 is roughly the overhead on function entrance; the time spent between time 2 and time 3 is the true cost of the Win32 API call; the time spent between time 3 and time 4 is overhead of handling function exit. The code maintains a global variable OverHead, which is the sum of all our spying overhead, and deducts it from time reporting.

The stack used in passing parameters and the storing function call return address grows downward; pushing a value decreases the stack pointer, popping a value increases it. The Win32 API uses a standard calling convention, in which the last parameter is pushed first onto the stack, then the one before it, until the first parameter is pushed. After the parameters block is a function return address. When our stub function is called, it pushes a function identifier or index on the stack, then calls ProxyProlog. Proxy Prolog adds a few common registers and a copy of the CPU flags register. All these values are mapped into a C-level structure EntryInfo, whose pointer is passed to ProxyEntry. ProxyEntry uses the pointer to EntryInfo to access function id and patch return addresses on the stack.

Here is the tricky part. After calling ProxyEntry, ProxyProlog restores several common registers and the flag register, then executes a “ret” instruction. To where is it returning control? What’s on top of the hardware stack was the function index pushed by stub routines, but later it is patched in ProxyEntry to be the original Win32 API address. So the last “ret” instruction in ProxyProlog actually returns control to the original API implementation. For example, if we’re hooking on the GDI function Delete Object, the stub code pushes an index (say 5) on the stack and calls ProxyProlog. Proxy Prolog calls ProxyEntry to record input parameters and patch the location of the index to be the real address of GDI’s DeleteObject function. So the last instruction in ProxyProlog transfers control to GDI’s DeleteObject function.

The spying exit handling is almost a mirror image of the entrance part. Here it is for completeness:

```
typedef struct
{
    unsigned m_rslt;
} ExitInfo;

_declspec(naked) void ProxyEpilog(void)
{
    __asm    push    eax      // API call result, also as
                  // placeholder for return address

    __asm    push    eax      // save common registers
    __asm    push    ebx
    __asm    push    ecx
    __asm    push    edx
    __asm    pushfd        // 4 bytes flags

    __asm    _emit  0x0F      // time3
    __asm    _emit  0x31
    __asm    shr    eax, edx, 8 // EAX = EDX:EAX >> 8
```

```
__asm push eax // leaving clock
__asm sub eax, OverHead
__asm push eax // leaving clock - overhead

__asm lea eax, [esp+28] // address of placeholder
__asm push eax
__asm call ProxyExit

__asm pop ecx // ecx = entering clock
__asm _emit 0x0F // time4
__asm _emit 0x31
__asm shrd eax, edx, 8 // EAX = EDX:EAX >> 8
__asm sub eax, ecx // new overhead by ProxyEpilog
__asm add OverHead, eax

__asm popfd // restore flags, common reg
__asm pop edx
__asm pop ecx
__asm pop ebx
__asm pop eax

__asm ret // to placeholder, org caller
}

void __stdcall ProxyExit(ExitInfo *info, unsigned leavetime)
{
    int depth;

    assert(pStack);

    KRoutineInfo * routine = pStack->Lookup(depth);

    if ( routine )
    {
        pFuncTable->FuncExitCallBack(routine,
            info, leavetime, depth);
        info->m_rslt = routine->rtnaddr;

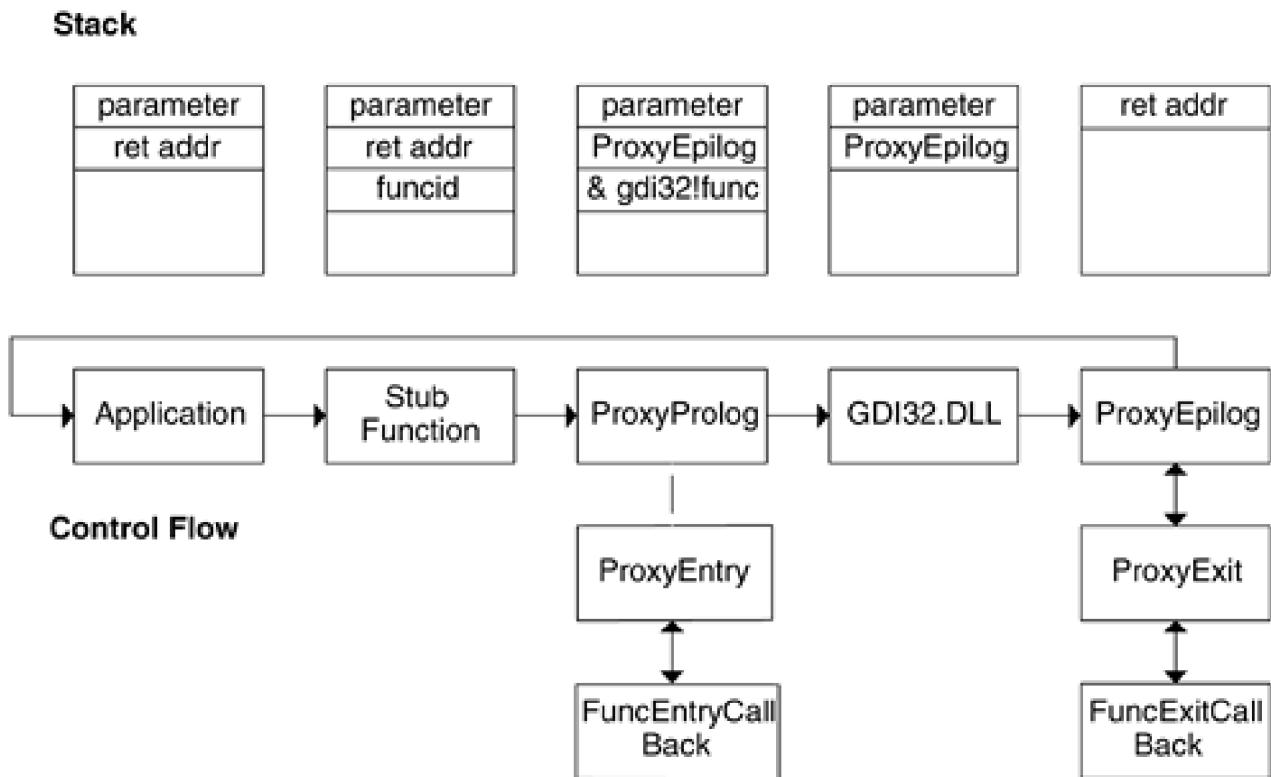
        pStack->Pop();
    }
}
```

When the Win32 API function being hooked returns, it does not return to the original caller. Instead, our routine ProxyEpilog gets called immediately. The reason is that routine ProxyProlog changes the return address on the stack to point to Proxy Epilog (through a simple assignment info->m_rtnaddr = (unsigned) ProxyEpilog). We are very careful here to save this return address on our software stack, so that we can use it later. Now the EAX register is critical; it holds a scalar function return value like a GDI handle returned by CreateSolidPen. ProxyEpilog saves it on the stack and passes the location to ProxyExit in a pointer to a ExitInfo structure. ExitInfo has just a single field, the function return value. ProxyExit locates the KRoutineInfo record from our software stack, calls

KFuncTable::FuncExitCallBack, and then patches the return value field on the stack, which is used by ProxyEpilog to return to the original caller through a ret instruction.

Figure 4-3 illustrates the control flow of hacking an API function, together with the changes happening on the CPU's stack. The bottom part shows control transfers from the application to the stub function, to ProxyProlog, to the Win32 API function, to Proxy Epilog, and finally back to the application. Both ProxyProlog and ProxyEpilog use help functions. The top part shows the changes on the stack.

Figure 4-3. API hacking control flow and stack changes.



One last thing to notice is the software stack implementation. For each API function call, the code needs a KRoutineInfo record to record information about the function call, which is used both by entrance handling routines and exit handling routines. Calling an API pushes one record on the stack; exiting from an API call pops the last record on the stack. This all sounds perfect until you have more than one thread running in the process. Consider this: The first thread makes an API which blocks on something, then a second thread makes another API call and blocks too; now the first thread wakes up and finishes the first API call. Consequently, our software stack is not truly last-in, first-out on a process basis. It's only truly last-in, first-out on a thread basis. Note that the CPU stack which the Win32 API calls rely on is perfect last-in, first-out, because each thread has a separate stack. To solve this problem, our software stack implementation marks each record with the current thread identifier. Push and pop on the software stack must be matched on a thread basis. A critical section is used to protect modification to the stack.

Data Dumping

The spying routines we have discussed do a good job of gathering API call information; now it's up to the data dumping part of the spying DLL to translate raw data into more meaningful and manageable forms.

There are several formats data can be dumped into, but simple text format is the easiest to generate and read. If you

process large amounts of spying data, you may prefer to use some spreadsheet or database software to help. Software like Microsoft Excel, Lotus 123, or Microsoft Access can easily import properly formatted text files into tables or spreadsheets.

You just need to make sure the columns within such text files are properly separated, either through fixed column width, tab, semicolon, comma, or other markers. For example, the SysCall program in [Chapter 2](#) is able to generate listings of GDI system service calls made in GDI32.DLL. But the order of listing is according to the order of debug symbols, not the system service call identifier, nor the address of the caller. You can create a database using Microsoft Excel, import the text file generated by SysCall as fixed-width text file, and properly adjust the column widths and types; now you have an Excel spreadsheet which can be easily sorted and analyzed. An example will be shown later in [Figure 4-5](#).

Our spying DLL uses text files as the output format, with a comma as the separator. File names follow the sequence pogy0000.txt, pogy0001.txt, and so on. The file creation code searches for the next available file name in this sequence, to make sure old logging files will not be overwritten.

Simple data dumping is rather easy. A single parameter in a Win32 API call is normally 4 bytes long; so is the scalar function return value stored in the EAX register. A no-brain solution would be dumping everything using 8 hexadecimal digits. So, TRUE is “0x00000001,” FALSE is “0x00000000,” raster operation SRCCOPY is “0x00CC0020,” and a text string is shown only by its address. This is hacker friendly, not ordinary user friendly at all.

The Win32 API defines a very rich set of types, or at least type macros. We have signed or unsigned types of different sizes, various pointers, endless handles, and more high-level types like BITMAPINFO, LOGFONT, DEVMODE, etc. Our API spy's design allows each of these types to be treated in its own special and unique way. For each Win32 API function, parameter types and return value types can be specified using type names. Values of the same type are decoded in a uniform way. Converting raw data to text format can be customized, and new data types can be added using plug-in DLLs.

To ease the burden of handling hundreds of type names, function names, and module names, an atom table is used to translate names in their text format into integer index. Instead of passing names like “COLORREF,” the code passes an integer value atom_COLORREF, which is obtained during the initialization phase by adding the name “COLORREF” to an atom table. All parts of the system use the same atom table, so if some other parts want to add “COLORREF” again to the atom table, the name is not added again; instead, its original integer value is returned. This is very much like the Win32 atom API. The code implements an atom table without using the Win32 atom API for performance and portability reasons.

The atom table is extracted into a C++ base class IAtomTable, much like a COM interface, except that we don't really need the IUnknown interface.

```
struct IAtomTable
{
    virtual ATOM      AddAtom(const char * name) = 0;
    virtual const char * GetAtomName(ATOM atom) = 0;
};
```

With an atom table, we can define a C++ base class that handles converting certain types of data into text format, IDecoder.

```
struct IDecoder
{
    virtual bool Initialize(IAtomTable * pAtomTable) = 0;

    virtual int Decode(ATOM typ, const void * pValue,
                      char * szBuffer, int nBufferSize) = 0;
};
```

In the last declaration, “struct” is similar to “class,” except that all data and functions defined in it are public. The COM keyword “interface” is defined as “struct,” in basetyps.h. The IDecoder::Initialize method adds a few data type names to the atom table. The IDecoder::Decode method decodes a block of data into a text buffer, and returns the size of data it consumes. This design allows it to handle a block of data instead of just a single 4-byte value. This is very useful in decoding parameters that can’t be meaningfully decoded separately. For example, the last two parameters to ExtText Out are a character count and a pointer to an integer array. Without the character count, a decoder does not know how many elements of the array it should decode. With IDecoder as defined above, you can define a new array type “CountedIntArray” and let the IDecoder::Decode method read two 32-bit values for the counted array. The return value of IDecoder::Decode is the number of bytes consumed, or zero if not handled.

The spy DLL provides a basic decoder (KBasicDecoder class) for simple decoding of common Win32 data types. Here is some of its code.

```
ATOM atom_char;
ATOM atom_BYT;
ATOM atom_COLORREF;
...
bool KBasicDecoder::Initialize(IAtomTable * pAtomTable)
{
    if ( pAtomTable==NULL )
        return false;

    atom_char    = pAtomTable->AddAtom("char");
    atom_BYT     = pAtomTable->AddAtom("BYTE");
    atom_COLORREF = pAtomTable->AddAtom("COLORREF");
    ...
    return true;
}

int KBasicDecoder::Decode(ATOM typ, const void * pValue,
                         char * szBuffer, int nBufferSize)
{
    unsigned data = * (unsigned *) pValue;

    if ( typ==atom_char )
    {
        wsprintf(szBuffer, "%c", data);
        return 4;
    }
```

```
if ( typ==atom_BYT )
{
    wsprintf(szBuffer, "%d", data & 0xFF);
    return 4;
}

if ( typ==atom_COLORREF )
{
    if ( data==0 )
        strcpy(szBuffer, "BLACK");
    else if ( data==0xFFFF )
        strcpy(szBuffer, "WHITE");
    else
        wsprintf(szBuffer, "%06x", data);
    return 4;
}

...
return 0; // unhandled types
}
```

During initialization, the spy DLL sets up an atom table, initializes an instance of KBasicDecoder, loads an .ini file to find customized IDecoder implementations, and loads and initializes each of them. A static routine, MainDecoder, is defined to manage the whole decoding process of a block of data. It goes through the chain of IDecoder implementation to find one able to decode certain types of data. The implementation for KFuncTable::FuncEntryCallBack and KFuncTable::FuncExitCallBack just calls MainDecoder.

Now we have an extensible Win32 data type decoder. Aren't we learning something from the WinDbg debugger extension design?

Spy Control Program

After describing the process of injecting the spy DLL, hooking into the Win32 API, gathering information and dumping data, what is still missing in our total spying solution? We still need parts of the spy control program that decide what program to attack, what modules to hack into, and what functions to put hooks on, together with the exact definition of the Win32 API usable by the control program.

Pogy, the spy control program, is driven by several standard Windows .ini files. They are in text format, everyone understands them, and the Win32 API supports their parsing. The control program is a Win32 application, so we have no intention to restrict usage of the Win32 features in it.

The main data file, Pogy.ini, has two sections. The target section lists applications to spy on, with per-application data files. The option section stores program options like whether to log API calls or display API call information in an event-logging window. It may also include optional data type decoding DLLs. Here is a sample Pogy.ini file.

```
[Target]
1=CLOCK.EXE (pclock.ini)
2=NOTEPAD.EXE (pnotepad.ini)
```

```
[Option]
LogCall=1
DispCall=0
Decoder1=pogygdi.dll!_Create_GDI_Decoder@0
Decoder2=pogyddraw.dll!Create_DDRAW_Decoder
```

According to this .ini file, we want to log API calls, but not display them. Two extra data type decoders are provided, one for GDI-related types and one for DirectDraw-related stuff. The user may spy one of the two programs, each with its own detailed .ini files.

A per-application .ini file lists modules within an application process to spy on. The user has to specify a caller module name, a callee module name, and the name of an .ini file for a set of the API. Here is an example:

```
[Module]
CLOCK.EXE, Gdi32.DLL, wingdi
CLOCK.EXE, User32.DLL, winuser
```

This example shows that we're interested in calls to GDI32.DLL and USER32.DLL; each has a separate the API set .ini file wingdi.ini and winuser.ini.

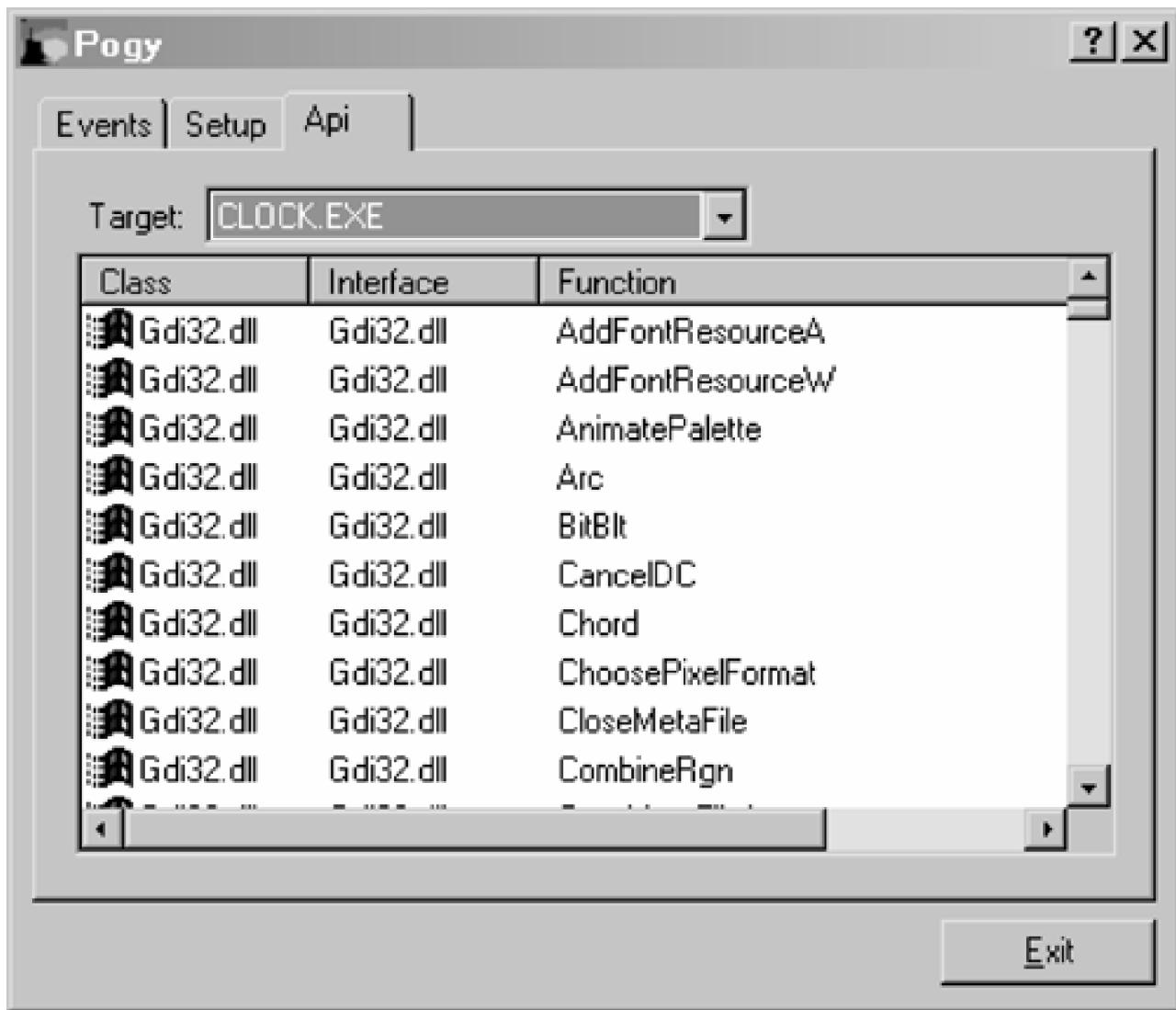
An API set .ini file is to our spy program what Windows header files are to C/C++ compiler; that is, it serves as the spy program's API specification. Wouldn't it be perfect if there were an automatic way to generate these .ini files based on Windows header files, or library files, or some kind of symbol files? But for the moment, let's just type in all the details like module name, function name, parameter type list, and function return type. Here is a small subset for the GDI API:

```
[wingdi]
int SelectClipRgn(HDC, HRGN)
int SetROP2(HDC, int)
BOOL SetWindowExtEx(HDC, int, int, LPSIZE)
BOOL SetBrushOrgEx(HDC, int, int, LPPOINT)
BOOL LPtoDP(HDC, LPPOINT, int)
HBRUSH CreateBrushIndirect(LPLOGBRUSH)
HBRUSH CreateDIBPatternBrushPt(LPVOID, UINT)
BOOL DeleteDC(HDC)
HBITMAP CreateBitmap(int, int, UINT, UINT, LPVOID)
HDC CreateCompatibleDC(HDC)
HBRUSH CreateSolidBrush(COLORREF)
HRGN CreateRectRgnIndirect(LPRECT)
INT SetBoundsRect(HDC, LPRECT, UINT)
BOOL PatBlt(HDC, int, int, int, int, DWORD)
BOOL SetViewportOrgEx(HDC, int, int, LPPOINT)
BOOL SetWindowOrgEx(HDC, int, int, LPPOINT)
int SetMapMode(HDC, int)
```

The user interface of the spy control program, Pogy, uses a property sheet with three property pages. The event

page logs events like shell window creation and destruction, API interception in the spy DLL, and API logging display if enabled. The setup page lets you toggle logging flags. The API page shows what the program reads from its .ini files. You can select the target application to spy on from the program listed in Pogy.ini. A list view window will show you details of API functions currently loaded. [Figure 4-4](#) shows the API property page of the program.

Figure 4-4. User interface of a generic Win32 API spying program.



Once Pogy is running, it installs the systemwide shell hook implemented in Diver.DLL. Any application's creation or destruction of top-level window will load our spy DLL into its address space. Diver.DLL figures out the name of the application's main executable file and sends a message to Pogy to check if it needs to spy on that process. If an order is given, it creates a hidden messaging window to receive information about the functions to spy on, hooks into the specified entry points, and starts to pump API logging data into a text file. The process ends when the target application quits.

After all these efforts, here is our final reward: a Win32 API call trace generated by the spying DLL under the control of the main program. [Figure 4-5](#) shows one API logging file when imported into Microsoft Excel.

Figure 4-5. API logging imported into Excel.

	A	B	C	D	E	F	G
1	Depth	Enter	Leave	Return	Caller	Callee	Parameter 1
2	1	183,258.00	183,262.00	2	CLOCK.EXE+16bd	Gd32.dll!SetBkMode	1010056
3	1	183,486.00	183,496.00	TRUE	CLOCK.EXE+16f5	Gd32.dll!DeleteObject	601004de
4	1	192,405.00	192,409.00	2	CLOCK.EXE+16bd	Gd32.dll!SetBkMode	1010058
5	1	192,482.00	192,488.00	TRUE	CLOCK.EXE+16f5	Gd32.dll!DeleteObject	611004de
6	1	192,552.00	192,566.00	1b0a0132	CLOCK.EXE+365b	Gd32.dll!CreateFontIndirectW	LOGFONTW"(135a3c0
7	1	192,608.00	192,653.00	18a0021	CLOCK.EXE+3671	Gd32.dll!SelectObject	1010058
8	1	192,655.00	762,630.00	TRUE	CLOCK.EXE+3689	Gd32.dll!GetTextExtentPointW	1010058
9	1	762,636.00	762,648.00	TRUE	CLOCK.EXE+36b1	Gd32.dll!GetTextExtentPointW	1010058
10	1	762,649.00	762,658.00	1b0a0132	CLOCK.EXE+36bd	Gd32.dll!SelectObject	1010058
11	1	762,663.00	762,710.00	TRUE	CLOCK.EXE+36f6	Gd32.dll!DeleteObject	1b0a0132
12	1	762,712.00	762,753.00	1c0a0132	CLOCK.EXE+3626	Gd32.dll!CreateFontIndirectW	LOGFONTW"(135a3c0
13	1	762,754.00	762,755.00	18a0021	CLOCK.EXE+3637	Gd32.dll!SelectObject	1010058
14	1	762,758.00	791,667.00	TRUE	CLOCK.EXE+365a	Gd32.dll!GetTextExtentExPointW	1010058

For each API call, which corresponds to one row in [Figure 4-5](#), the data shows the nesting level of the API call (currently only one) the function entering and leaving times, the function return value, the caller's address and the callee's function name, and all the input parameters. Some data are shown in decimal form, some in hexadecimal format, and a few in mnemonics.

Currently, the spy control program is only programmed to do API logging. It's possible to add code to enable parameter validation, function result checking, and even memory/resource leak detection. For parameter validation, we need to know the valid value range for each parameter and check for that. For example, SelectObject can only select a valid GDI handle, either that of a stock object or what's created in the current process, into a device context. Selecting an invalid GDI handle, or a handle owned by other process, is a warning sign, especially on Windows NT/2000. Function result checking is basically the same. For example, if SelectObject returns on a valid GDI handle, it's a sign of improper deselection of a GDI object, a possible cause for a GDI object leak. Detecting GDI object leakage is more complicated. All the object-creation calls need to be recorded with the handle value and caller's address. When a GDI object is deleted (by calling DeleteObject), its handle needs to be removed from our recorded handle list. When the program terminates, what is left in your list are leaked GDI handles; for these we should report the exact locations of their creators. Again, debug symbol files or map files can help us to translate addresses to more meaningful names.

4.2 SPYING ON WIN32 GDI

As is true with any part of Windows programming, doing an OK job is easy, but doing an excellent job is hard. So we've got a WIN32 API spy with a passing mark, but not such a great one. To achieve our purpose of helping us understand every piece of GDI, it needs much more work.

What we really need to do is make it work great for spying on the GDI and Direct Draw function calls, which is the focus of this book. This gives us freedom to use the WIN32 API functions implemented in KERNEL32.DLL and USER32.DLL without worrying whether they themselves could be the target of spying.

The GDI API Definition File

The first thing we need is a complete or almost complete .ini file, readable by the spy DLL, which describes as many GDI API calls as possible.

The window header files contain the essential information we need, together with information we don't care very much about. Preferably, we need an automatic tool to extract basic function prototypes from these header files and convert them into a simpler format. This would be a small project for students taking a compiler course, a simple, yet special parser for C header files.

The result is a small Windows console program, Skimmer, which searches for key macros like WINGDIAPI, WINUSERAPI, WINAPI, and APIENTRY, normal starting points for a WIN32 API prototype. After confirming that a function prototype is found, decorative words like CONST, FAR, IN, OUT are removed, together with parameter names, giving us a concise definition of a WIN32 API function.

Functions exported by GDI32.DLL that are documented are then defined in three header files on Windows 2000 DDK: inc\wingdi.h, for normal GDI features, inc\winddi.h, for user mode DDI functions, and src\print\genprint\winppi.h, for GDI functions supporting the EMF print processor. With Visual C++, you only get include\wingdi.h.

Running Skimmer on these three header files gives us three .ini files readily usable by our spy program. The only exception is the prototype for a DDI function EngGetFilePath; a little manual editing is needed. Some smart programmer used **WCHAR (*pDest)[MAX_PATH+1]** in declaring its second parameter; this is way too complicated for our simple-minded parser. Here is the smallest of the three, the GDI API for the EMF print processor:

```
[winppi]
HANDLE GdiGetSpoolFileHandle(LPWSTR,LPDEVMODEW,LPWSTR)
BOOL GdiDeleteSpoolFileHandle(HANDLE)
DWORD GdiGetPageCount(HANDLE)
HDC GdiGetDC(HANDLE)
HANDLE GdiGetPageHandle(HANDLE,WORD,LPDWORD)
BOOL GdiStartDocEMF(HANDLE,DOCINFOW*)
BOOL GdiStartPageEMF(HANDLE)
BOOL GdiPlayPageEMF(HANDLE,HANDLE,RECT*,RECT*,RECT*)
```

```
BOOL    GdiEndPageEMF(HANDLE,DWORD)
BOOL    GdiEndDocEMF(HANDLE)
BOOL    GdiGetDevmodeForPage(HANDLE,DWORD,PDEVMODEW*,PDEVMODEW*)
BOOL    GdiResetDCEMF(HANDLE,PDEVMODEW)
```

[Types]

HANDLE
LPWSTR
LPDEVMODEW
BOOL
DWORD
HDC
LPDWORD
DOCINFOW*
RECT*
PDEVMODEW*
PDEVMODEW

An API definition file as shown above has two sections. The first section lists function prototypes in a simplified form, while the second section lists unique data types used by those API functions.

The Win32 GDI relies on the window management module (USER32.DLL) to provide the environment in which most of drawing happens. The USER32 module does have some interesting API that we should consider spying on, things like Begin Paint, EndPaint, GetDC, etc. So use Skimmer to generate winuser.ini from winuser.h.

The GDI Data Decoder

As a hint to the spy DLL, Skimmer lists all the data types used in the particular group of API functions. To make spying logging data more readable, we need a special data decoder to handle GDI-specific data types, things like HGDIOBJ, LOGFONTW, and hopefully even DEVMODEW, if that is really important to someone.

With our knowledge of GDI data structures developed in [Chapter 3](#), it's a simple matter of programming to develop a GDI data decoder DLL. Here is the skeleton GDI decoder DLL:

```
class KGDIDecoder : public IDecoder
{
    ATOM atom_HGDIOBJ;
    ...
public:
    KGDIDecoder()
    {
        pNextDecoder = NULL;
    }
};

virtual bool Initialize(IAtomTable * pAtomTable);
virtual int Decode(ATOM typ, const void * pValue,
                  char * szBuffer, int nBufferSize);
```

```
};

bool KGDIDecoder::Initialize(IAtomTable * pAtomTable)
{
    if ( pAtomTable==NULL )
        return false;

    atom_HGDIOBJ = pAtomTable->AddAtom("HGDIOBJ");
    ...

    return true;
}

// GDI object type lookup table

int KGDIDecoder::Decode(ATOM typ, const void * pValue,
                       char * szBuffer, int nBufferSize)
{
    unsigned data = * (unsigned *) pValue;

    if ( (typ==atom_HDC) || (typ==atom_HGDIOBJ) ||
        (typ==atom_HPEN) || (typ==atom_HBRUSH) ||
        (typ==atom_HPALETTE) || (typ==atom_HRGN) ||
        (typ==atom_HFONT) )
    {
        TCHAR temp[32];

        unsigned objtyp = (data >> 16) & 0xFF;

        if ( ! Lookup( objtyp & 0x7F, Dic_GdiObjectType, temp ) )
            _tcscpy(temp, "HGDIOBJ");

        if ( objtyp & 0x80 ) // stock object
            wsprintf(szBuffer, "(S%s)%x", temp, data & 0xFFFF);
        else
            wsprintf(szBuffer, "(%s)%x", temp, data & 0xFFFF);

        return 4;
    }

    if ( typ==atom_PLOGFONTW )
    {
        LOGFONTW * pLogFont = (LOGFONTW *) data;

        if ( ! IsBadReadPtr(pLogFont, sizeof(LOGFONTW)) )
        {
            wsprintf(szBuffer, "& LOGFONTW{%d,%d,...,%ws} ",
                    pLogFont->lfHeight, pLogFont->lfWidth,
                    pLogFont->lfFaceName);
        }
    }
}
```

```
    return 4;
}

}

...
// unhandled
return 0;
}

KGDIDecoder GDIDecoder;

extern "C" __declspec(dllexport)
IDecoder * WINAPI Create_GDI_Decoder(void)
{
    return & GDIDecoder;
}
```

The KGDIDecoder class is a simple decoder for GDI data types, which implements the IDecoder base class, much as a COM class implements a COM interface. The Create_GDI_Decoder routine returns a pointer to a global instance of KGDI Decoder class, similar to what a singleton COM class's class factory does. The spy DLL loads the GDI decoder DLL during runtime, finds the address of the creator, calls it, and then uses the KGDIDecoder->Initialize method to set up the decoder. It then chains new decoders on top of old ones, calling them one by one to convert spying logging data into text format, until its request is handled.

From [Chapter 3](#), we know a GDI handle on Windows NT/2000 is made up of three parts: an 8-bit uniqueness value, an 8-bit type identifier, and a 16-bit index. The code here uses this knowledge to decode a GDI object handle into its type name and index part through a lookup table. For pointers to LOGFONTW structure, the code shows the logical font's height, width, and face name.

Configuring the spy DLL to use this new decoder improves its logging a lot. All GDI object handles are now clearly marked with their types. So you will see clear patterns where the application creates a GDI object, selects it, uses it, deselects it, and then finally deletes it. Adding more decoding capabilities will improve API spying even more.

Complete API Spying

So far we have hooked into the API calling chain by modifying a module's import directory. So if you modify module CLOCK.EXE's import directory to spy on the GDI function Select Object, all calls from that module will be captured, unless the code is tricky enough to call SelectObject indirectly using GetProcAddress. Many window drawing calls, like the caption, title bar, menu, and icons, are not performed directly by your program. Instead, the default window message procedure gets to handle most of these drawing tasks in its predefined ways. For MFC programs, if you're using the DLL version of the MFC library, most GDI drawing calls are issued from MFC DLLs (for example, MFC42.DLL or MFC42D.DLL for MFC version 4.2).

If you want to spy on GDI drawing calls from all these modules in your process through the patching module import directory, you need to list all the modules in spy target's .ini file. The spy DLL will go through each of them to patch their import directory.

Even enumerating all the modules in a process is not enough. New modules—for example, COM DLLs—may be

loaded during runtime without your prior knowledge. To make things more complicated, consider that when GDI32 calls an exported GDI32 function—for example, SelectObject—the call does not go through its import directory. It's considered a direct intramodule call; nothing like call [__imp_SelectObject] is involved.

To completely spy on an API call processwide—that is, to capture calls to it from within its residing module and outside, from loaded modules and modules yet-to-be loaded—we need to patch the API implementation itself. For example, if you find the address of the SelectObject function in GDI32.DLL and modify the function itself, all calls to SelectObject within that particular process will go through your modified code.

Modifying a piece of code is easy; the hard part is how to make an application still function the same way after the modification. As we saw in [Section 4.1](#), what we need to do is insert a few lines of assembly code to force the API call to jump to our function entrance handling code. After it returns, the original API call needs to be executed with exactly the same set of register values. After the API call returns, our function exit handling code needs to get control again, before finally returning to the original caller.

The main trouble is that, because we have to modify the API function entrance, when the function entrance handling returns it can't return to the patched function entrance. There are two basic designs to solve this problem. The first design is to restore the patch in the entrance handling routine, such that when it returns, control goes smoothly to the original API implementation. This works perfectly for single-shot API logging, but when do you patch the code again for subsequent logging? The only natural place to patch again would be in the exit handling routine, but this means we can't handle recursive calls and calls from other threads during the period an API is being called.

This leads us to the second design: Do not restore the patch, but instead relocate the original function entrance. When we patch the original API entrance, a minimum of 5 bytes is needed for a jump instruction to a piece of stub code, which damages a few instructions. Those damaged instructions can be copied into a buffer in the spying DLL's storage space, followed by an instruction to jump back to the first instruction after the damaged instructions. When all these things are done, we just need to let the spying DLL function entrance handling routine return to the relocated instructions. An example will make this very clear. Let's look at the first few instructions in SelectObject, in its assembly and machine-code form:

```
_SelectObject@8:  
55          push ebp  
8B EC        mov ebp, esp  
51          push ecx  
83 65 FC 00    and dword ptr [ebp-4], 0  
_SelectObject@8+8:  
...  
...
```

Five bytes starting at address `_SelectObject@8` are needed for our little surgery, which will damage the first 4 instructions, 8 bytes in total. We save the first 8 bytes of `_SelectObject@8` and patch it with a jump instruction to a piece of stub code in our control.

```
_SelectObject@8:  
E9 xx xx xx xx jmp Stub_SelectObject@8  
90          nop  
90          nop  
90          nop  
_SelectObject@8+8:  
...
```

...

Note here that only 5 bytes are needed, but to make the code look normal in a disassembler, three single no-operation instructions are added. The stub code looks like this:

```
Stub_SelectObject@8:  
    push index_selectobject  
    jmp ProxyProlog  
New_SelectObject@8:  
    push ebp  
    mov ebp, esp  
    push ecx  
    and dword ptr [ebp-4], 0  
    jmp _SelectObject@8+8
```

There are two interesting things to notice here. First, the Stub_SelectObject@8 is exactly the same as the code used by our previous import directory hacking. Second, the New_SelectObject@8 is a rebirth of the _SelectObject@8 routine before hacking. These two identities make it possible to reuse all the code for import-directory-based API hacking, except we have to set **pFuncTable->m_func[index_selectobject].f_old address** to be **New_SelectObject@8**, to ensure that when ProxyProlog returns, control goes to the same logical as the original API call.

We have not seen the most complicated part of this hacking-by-relocation design yet, which is the seemingly simple task of figuring out how many bytes need to be relocated. The minimum is known, 5 bytes; the exact number is hard to find because we need to make sure that only whole instructions are copied. For the Intel CPU, there is no easy formula to calculate instruction length based on the first few bytes. By continuing to add new instructions to the original 8086 instruction set, Intel has made the instruction map a mine field to traverse. The code to calculate the number of bytes, which covers whole instructions no less than 5 bytes, is basically a disassembler skeleton.

In the old Win16 days, this would be much easier with exported functions all having the same function prolog code. But with 32-bit code and compilers with better optimizations, especially optimization for CPUs with multiple instruction pipe lines, the start instruction of a function is unpredictable.

This presents us with another possible difficulty: Not all instructions can be relocated by simple copying. Control transfer instructions—for example, “jmp”—often use relative addresses that are sensitive to location. If you want to relocate them, you need to update the relative offset. Our current implementation does not support function prologs with relative jumps.

Hacking-by-relocation is implemented in a static library Patcher.lib, which is linked by spying DLL Diver.DLL. To specify that you want to use processwide hacking, give the caller and callee module the same name. For example, the following .ini file commands processwide hacking for GDI32 functions listed in wingdi.ini, and USER32 functions listed in winuser.ini:

```
[Module]  
Gdi32.dll, Gdi32.DLL, wingdi  
User32.dll, User32.DLL, winuser
```

With processwide hacking, you will see many more GDI API calls from other system DLLs, like USER32.DLL,

COMDLG32.DLL, COMCTL32.DLL, OLE32.DLL, and even GDI32.DLL itself. You will find that USER32.DLL calls GDI32.DLL for drawing operations, GDI32.DLL merges API calls into more generic calls or decomposes complicated API calls into simpler ones. You will experience the whole drama from backstage. [Figure 4-6](#) shows a small example.

Figure 4-6. Complete API spying reveals LoadBitmap implementation.

A	B	C	D	E	F	
Depth	Enter	Leaver	Return	Caller	Callee	
1						
2	1	6271986	6272992	HBITMAP(520504c8)	CARDS.dll+17e9	user32.dll!LoadBitmapA
3	2	6272096	6272176	(HDC)2d7	USER32.dll+c601	user32.dll!GetDC
4	2	6272179	6272457	HBITMAP(520504c8)	USER32.dll+c650	Gdi32.dll!CreateCompatibleBitmap
5	2	6272458	6272486	1	USER32.dll+c663	user32.dll!ReleaseDC
6	2	6272490	6272526	(SHBITMAP)f	USER32.dll+c9c2	Gdi32.dll!SelectObject
7	2	6272526	6272528	WHITE	USER32.dll+c9e1	Gdi32.dll!SetBkColor
8	2	6272529	6272532	BLACK	USER32.dll+c9f1	Gdi32.dll!SetTextColor
9	2	6272534	6272966	96	USER32.dll+ca1e	Gdi32.dll!SetDIBits
10	3	6272545	6272593	1	GDI32.dll+6baa	Gdi32.dll!SaveDC
11	3	6272594	6272607	(HBITMAP)4c8	GDI32.dll+6bbb	Gdi32.dll!SelectObject
12	3	6272616	6272626	(SHPALETTE)b	GDI32.dll+6bd7	Gdi32.dll!SelectPalette
13	3	6272629	6272902	96	GDI32.dll+6c0a	Gdi32.dll!SetDIBitsToDevice
14	3	6272903	6272915	(SHPALETTE)b	GDI32.dll+6c1c	Gdi32.dll!SelectPalette
15	3	6272916	6272925	(HBITMAP)4c8	GDI32.dll+6c25	Gdi32.dll!SelectObject
16	3	6272926	6272966	TRUE	GDI32.dll+6c32	Gdi32.dll!RestoreDC
17	2	6272967	6272968	BLACK	USER32.dll+ca3f	Gdi32.dll!SetTextColor
18	2	6272968	6272969	WHITE	USER32.dll+ca4a	Gdi32.dll!SetBkColor
19	2	6272969	6272983	(HBITMAP)4c8	USER32.dll+ca62	Gdi32.dll!SelectObject
20						

The API calls shown in [Figure 4-6](#) are listed in the order they started, with the first column being nesting level; the second, return value; the third, caller's address; and the fourth, the API being called. Parameters are not shown for space reasons. Note that the original logging files generated are sorted according to the time the function call starts, instead of finishes. [Figure 4-6](#) uses Excel's data-sorting feature to make the data easier to comprehend.

What's shown here is the secret implementation of the LoadBitmap function. LoadBitmap is a function provided by the window manager (USER32.DLL) to load a bitmap into a GDI device-dependent bitmap format. But we don't know how it's implemented. [Figure 4-6](#) shows that LoadBitmapA (the ANSI-character version of LoadBitmap) uses a bunch of GDI functions to convert a device-independent bitmap to a device-dependent bitmap. We can find CreateCompatibleBitmap for creating a new DDB, and SetDIBits for conversion. [Figure 4-6](#) also shows how SetDIBits is implemented in GDI; it just calls SetDIBitsToDevice.

4.3 SPYING ON DIRECTDRAW COM INTERFACES

The Microsoft DirectDraw API, together with the rest of the DirectX API, is based on Microsoft's COM (Component Object Model) technology. DirectDraw's functionality is exposed through a few COM interfaces—for example, IDirectDraw and IDirect Draw Surface.

A COM interface is defined as a group of semantically related functions, or methods. For example, methods in the IDirectDrawSurface interface deal with a Direct Draw drawing surface; methods in the IDirectDrawClipper interface manage clipping for a DirectDraw surface. Now let's figure out how to hook into these methods.

Virtual Function Table

All calls to methods in a COM interface go through an interface pointer, which is actually a pointer to a C++ object with unknown data representation. The only thing the client side of a COM object knows is that a COM object starts with a virtual function table pointer. This table contains pointers to implementation routines for all methods within the interface, in a predefined order. All COM interfaces are derived from the IUnknown interface, which defines three methods: QueryInterface, AddRef, and Release. This means the first three pointers within a COM virtual function table always implement these three methods.

Normally, a C++ virtual function table or a COM virtual function table is generated by a compiler in either read-only or read-write global data storage. This is very much like the internal variables used by the module's import directory to hold addresses of imported functions.

Technically, hacking into a COM interface or a DirectDraw interface is not difficult at all. We just need to find addresses for all the virtual function tables we are interested in, then overwrite function pointers within them to point to stub functions within our spying DLL.

To get the address of a normal COM interface's corresponding virtual function table is quite easy. Find its class GUID and interface GUID, call CoCreateInstance with them, and the operating-system COM implementation will load the right COM server, create a COM object, and return an interface pointer to you. Reading the first 4 bytes of the area pointed to by the interface pointer gives you the virtual function table address.

Most DirectDraw interfaces can't be created by the standard CoCreateInstance call. For example, the only way to create an IDirectDrawSurface interface pointer is to use the CreateSurface method on an IDirectDraw interface. This makes perfect sense for DirectDraw, because a DirectDraw surface is always managed by a DirectDraw object.

A spying DLL would prefer to disturb the system in the least amount. So creating a DirectDraw

object, and then a DirectDraw surface, just to get the IDirectDrawSurface virtual function table is not so attractive. The alternative is to query for virtual function tables offline in a separate program, keep them in an .ini file, and then reuse them during an actual spying mission. QueryDDraw is such a program. It tries to create as many different kinds of DirectDraw interface pointers as possible, logging their virtual function table address, number of methods, interface name, and interface GUID.

In the C++ world, there seems to be no way to get the number of methods in a class, but a DirectDraw program written in C must know it, because the virtual function table is simulated explicitly using a structure of function pointers. The following code fragment does the trick to get information on IDirectDraw interface.

```
#define CINTERFACE
#include <ddraw.h>
...
IDirectDraw * lpdd;
HRESULT hr = DirectDrawCreate(NULL, & lpdd, NULL);
DumpInterface("IID_IDirectDraw," IID_IDirectDraw,
lpdd->lpVtbl, sizeof(*lpdd->lpVtbl) );
```

Before including ddraw.h, the include file for DirectDraw, macro CINTERFACE, is defined. This enables the C-style definition of COM interfaces, where a virtual function table is explicitly simulated using an array of function pointers, and the pointer to the virtual function table is an explicit field in a structure (lpVtbl). This C-style definition of COM interfaces allow us to use the size of(*lpdd->lpVtbl) to compute the size of the virtual function table, and thus the number of functions in the table.

Calling a C++ method is a little bit different from a normal C or Pascal function call. An extra implicit pointer to the object is passed to the method being called, commonly known as “this” pointer. Although the C++ compiler supports passing “this” pointer through the register for performance reasons, a COM interface or a Direct Draw interface always uses stack to pass its “this” pointer. As to the spying DLL’s parameter-dumping routine, it just needs to be informed that there is an extra parameter.

The DirectDraw API Definition

Our next task is generating an .ini file for all DirectDraw methods, which needs to be readable by our spy control program. This requires a few modifications to our simple C header file parser Skimmer. It now checks for “DECLARE_INTERFACE_” to find the starting point of a COM interface declaration, undoes “STDMETHOD” and “STDMETHOD_” macros to restore function return type and function name, and handles “THIS” and “THIS_” macros to add “this” pointer as an extra parameter.

Here is an edited version of our complete, precise, and concise DirectDraw API definition for a

dummy spy:

[ddraw]

```
HRESULT DirectDrawEnumerateW(LPDDENUMCALLBACKW,LPVOID)
HRESULT DirectDrawEnumerateA(LPDDENUMCALLBACKA,LPVOID)
HRESULT DirectDrawEnumerateExW(LPDDENUMCALLBACKEXW,LPVOID,DWORD)
HRESULT DirectDrawEnumerateExA(LPDDENUMCALLBACKEXA,LPVOID,DWORD)
HRESULT DirectDrawCreate(GUID*,LPDIRECTDRAW*,IUnknown*)
HRESULT DirectDrawCreateClipper(DWORD,LPDIRECTDRAWCLIPPER*,
IUnknown*)
```

[COM_ddraw]

```
728405a0 728318a8 23 {6c14db80-a733-11ce-a5-2100-20af-0b-e5-60}
```

```
IID_IDirectDraw
```

```
728408e0 728318a8 24 {b3a6f3e0-2b43-11cf-a2-de-00aa-00b9-33-56}
```

```
IID_IDirectDraw2
```

```
728407a0 728318a8 28 {9c59509a-39bd-11d1-8c-4a-00-c0-4f-d9-30-c5}
```

```
IID_IDirectDraw4
```

```
72840940 7282f034 36 {6c14db81-a733-11ce-a5-2100-20af-0b-e5-60}
```

```
IID_IDirectDrawSurface
```

[IDirectDraw]

```
HRESULT QueryInterface(THIS,REFIID,LPVOID*)
ULONG AddRef(THIS)
ULONG Release(THIS)
HRESULT Compact(THIS)
HRESULT CreateClipper(THIS,DWORD,LPDIRECTDRAWCLIPPER*,IUnknown*)
HRESULT CreatePalette(THIS,DWORD,LPPALETTEENTRY,
LPDIRECTDRAWPALETTE*,IUnknown*)
HRESULT CreateSurface(THIS,LPDDDSURFACEDESC,
LPDIRECTDRAWSURFACE*,IUnknown*)
...
```

The first section shows the normal exported functions DDRAW.DLL; the facts listed here. Quite a few functions are exported from ddraw.dll. Those documented in ddraw.h are shown here; some are common COM export functions like DIIGetClassObject; others may be documented in other header files if not undocumented.

The second section lists COM interfaces information generated by the QueryDDraw program. For each DirectDraw COM interface, we have its virtual function table address, first virtual function (QueryInterface) address for verification, number of methods, interface GUID, and interface name. This section needs to be regenerated for every OS and its service packs.

Sections after the first two sections list details of method prototypes, one section for each interface.

What's shown here is just for the IDirectDraw interface. IDirectDraw2 interface adds only one method on top of IDirectDraw interface, while IDirectDraw4 interface has four new methods over IDirectDraw2 interface.

Virtual Function Table Hacking

Information in the DirectDraw API definition file is read by the spy control program and fed to the spying DLL. The spying DLL builds a table of interfaces, each with its name, GUID, virtual function table address, QueryInterface address, and number of methods in the interface. During initialization time, it loads the COM DLL—ddraw.dll in this case—locates each targeted virtual function table, and verifies that its first item matches the QueryInterface method's address that we know about.

The spying DLL then calls the following routine to hack all DirectDraw methods listed.

```
BOOL HackMethod(unsigned * vtable, int n, FARPROC newfunc)
{
    DWORD cBytesWritten;

    WriteProcessMemory(GetCurrentProcess(),
        & vtable[n], & newfunc, sizeof(newfunc), &cBytesWritten);

    return cBytesWritten == sizeof(newfunc);
}
```

The newfunc parameter points to the same stub routine as we described in [Section 4.1](#). After hacking, everything runs the same as import-directory hacking. Here is a tiny example on the IDirectDraw interface:

```
HRESULT(0), ddraw.dll!SetCooperativeLevel,
    0x893b28, HWND(800cc), 17
HRESULT(0), ddraw.dll!SetDisplayMode,
    0x893b28, 640, 480, 24
HRESULT(0), ddraw.dll!CreateSurface, 0x893b28,
    LPDDSURFACEDESC(12fe54),
    LPDIRECTDRAWSURFACE*(12ff2c), IUnknown*(0)
ULONG(0), ddraw.dll!Release, 0x893b28
```

This calling sequence shows that after a DirectDraw object is created, an application calls IDirectDraw interface's SetCooperativeLevel, SetDisplayMode, Create Surface methods, and finally the Release method to destroy the object. The first parameter shown, 0x893b28, is the "this" pointer. We definitely would benefit from a DirectDraw data type decoder DLL.

4.4 SPYING ON GDI SYSTEM CALLS

Having discussed three types of API spying, we are going to talk now about some undocumented stuff. Yes, we mean spying on the GDI system service calls, the interface from the user mode GDI client to the kernel mode graphics engine. We mentioned before that DirectDraw, Direct3D, and OpenGL all use GDI to make system service calls to the graphics engine.

In [Chapter 2](#), we discussed the architecture of the Windows NT/2000 graphics system. The role of graphics system service calls is very important. They are responsible for passing user mode drawing requests to the kernel mode graphics engine and device drivers. But system service calls are not documented, especially graphics system calls.

[Chapter 2](#) introduced “SysCall,” a program that searches for system call routines in Win32 subsystem client DLLs, namely Gdi32.DLL, USER32.DLL, and Kernel32.DLL. With the help of debugger symbol files, the program is able to list all occurrences of system calls with their function index, number of parameters, address, and symbolic name. It can even list system call service table in the kernel address space. But because system call service routines in the graphics engine match routines making system calls in user mode, it would be much easier for us to spy on the user mode side of this undocumented interface.

The listing file generated by “SysCall” is not quite what our spy program wants to see. Some enhancements are needed to make it generate a list of function prototypes. Unlike other ways of hacking, we need the addresses of those functions to be included with their prototypes. Otherwise, the spy program has to rely on debugger symbol files during runtime, which will limit its usefulness.

We have added a new menu command, “GDI32 system calls for Pogy,” to “Sys Call.” Here is a small portion of the routines making system calls in GDI32.DLL:

```
[gdisyscall]
D NtGdiCreateEllipticRgn(D,D,D,D), 77F725AB, 1020
D NtGdiDdGetBltStatus(D,D), 77F726A7, 1047
D NtGdiGetDeviceGammaRamp(D,D), 77F728D7, 10a8
D NtGdiSTROBJ_dwGetCodePage(D), 77F72CB7, 1274
D NtGdiGetTextExtentExW(D,D,D,D,D,D,D), 77F43C51, 10c9
D NtGdiGetColorAdjustment(D,D), 77F728BB, 10a1
D NtGdiFlush(), 77F413F9, 1093
D NtGdiDdSetOverlayPosition(D,D,D), 77F7274F, 105d
D NtGdiPATHOBJ_bEnumClipLines(D,D,D), 77F72CD3, 1279
D NtGdiEngCreateBitmap(D,D,D,D,D), 77F4B5CD, 1240
D NtGdiColorCorrectPalette(D,D,D,D,D), 77F7258F, 1011
D NtGdiDdDestroySurface(D,D), 77F5AAB2, 1041
```

D NtGdiDdRenderMoComp(D,D), 77F72717, 1057

You may have noticed here that we don't have the exact parameter and return type information. The only known fact is the number of parameters, which can be figured out from the number of bytes each routine pops out from stack when it returns. So we simply mark every data type as "D," short for "DWORD," until we get more evidence to change them to more meaningful types.

To hack into those routines, the spying DLL needs a few changes. First, the function address is known, so nothing like GetProcAddress for the Win32 API is needed, although the code should verify that the address contains a valid code fragment for a system call, in the format:

```
NtGdi_SysCall_xx
mov eax, function_index
NtGdi_SysCall_xx+5:
lea edx, [esp+4]
int 0x2E
ret parameter_number * 4
```

We could have used the hacking-by-relocation method that is used for implementing processwide API hacking, but we noticed that the instructions after NtGdi_SysCall_xx+5 have limited versions, one for each parameter count. The number of parameters a GDI system call accepts varies from 0 to 15. So we need only 15 routines to replace the code after the first instruction setting up the function index. This is the code after hacking:

```
NtGdi_SysCall_xx:
mov eax, function_index
NtGdi_SysCall_xx+5:
jmp Stub_NtGdi_SysCall_xx

Stub_NtGdi_SysCall_xx
push func_id
jmp ProxyProlog
```

When ProxyProlog returns, we make sure control goes to one of those replace routines making the actual system calls.

```
// For system calls with 2 parameters
__declspec(naked) void SysCall_08(void)
{
    __asm lea edx, [esp+4]
    __asm int 0x2E
    __asm ret 0x08
```

}

Hacking system calls would have been much simpler, had we not reused the core spying routines ProxyProlog, ProxyEntry, ProxyEpilog, and ProxyExit.

The new graphics system calls spying is really exciting, because it's different from the normal API spying. Quite a few books and magazine articles have talked about it in great depth. Doing GDI API spying together with graphics system calls spying is even more exciting. It is possible that no one has ever done this before.

The GDI API is the interface between application and user mode OS support; the graphics system call is the interface between GDI and the kernel mode graphics engine. So we are spying on both ends of the GDI client DLL GDI32.DLL. The difference between them shows what the GDI client DLL is actually doing. [Table 4-1](#) provides an edited version of a mixed GDI call and graphics system call logging.

Keep in mind that a function call of a higher nesting level is called by a function of a lower nesting level below it. For this logging, we can confirm quite a few points we made in previous chapters.

- Part of device context data structure is implemented in user mode, so simple queries on a device context can be handled purely and efficiently in user mode without calling kernel mode system service.
- The GDI object table is managed by the graphics engine, so object creation and destruction calls system service. Brushes and rectangle regions are special; GDI caches deleted objects for reuse. We see here that deleting an HPEN calls NtGdiDeleteObjectApp, while deleting an HBRUSH does not always go to a system call.
- CreateDiscardableBitmap is just CreateCompatibleBitmap.
- Drawing commands are normally passed directly to system calls.
- GDI system calls still use basically the same data types as Win32 GDI API calls.

What's provided here is really a toolbox for you to use to explore GDI on your own. You can design your own experiment in the area of the GDI API in which you're interested, set up the right API spying options, run the test, and analyze the results.

Table 4-1. Sample Win32 GDI Call and System Call Log

Nesting Level	Result	Function Call
1	(SHFONT)21	SelectObject((HDC)407, (HFONT)3e1)
1	WHITE	SetBkColor((HDC)305, a9c8a2)

1	0	SetTextAlign((HDC)305, 0)
1	BLACK	SetTextColor((HDC)305, BLACK)
2	TRUE	NtGdiDeleteObjectApp((HPEN) 4d9)
1	TRUE	DeleteObject((HPEN)4d9)
1	TRUE	DeleteObject((HBRUSH)3e8)
1	(SHBRUSH)10	GetStockObject(0)
2	(SHPEN)17	NtGdiGetStockObject(7)
1	(SHPEN)17	GetStockObject(7)
3	(HFONT)3e1	NtGdiHfontCreate(0x12f8c0, 0x164, 0x0, 0x0, 0x137468)
2	(HFONT)3e1	CreateFontIndirectExW(ENUMLOGFONTEXDVW*(12f8c0))
2	TRUE	NtGdiGetWidthTable((HDC)407, 0xb, 0x137b28, 0x10b, 0x137d3e, 0x1378b8,
1	TRUE	GetTextExtentPointW((HDC)407, LPCWSTR(135a394), 11, LPSIZE(12fac4))
3	HBITMAP(3d9)	NtGdiCreateCompatibleBitmap((HDC) 407, 0x20, 0x24)
2	HBITMAP(3d9)	CreateCompatibleBitmap((HDC)407, 32, 36)
1	HBITMAP(3d9)	CreateDiscardableBitmap((HDC)407, 32, 36)
2	TRUE	NtGdiRectangle((HDC)2e9, 0x60, 0x3, 0x64, 0x7)
1	TRUE	Rectangle((HDC)2e9, 96, 3, 100, 7)

[< BACK](#) [NEXT >](#)

4.5 SPYING ON THE DDI INTERFACE

Spying in the user mode portion of the windows graphics system has been covered extensively in the last four sections. We are now able to spy on both the incoming and outgoing interfaces of GDI32. The new territory for us is the kernel mode graphics engine.

The graphics engine exposes its functionality to the Win32 subsystem DLL through system calls. [Chapter 2](#) presents the “SysCall” program that shows the complete list of graphics and windows management system calls. Routines making system calls in GDI32 and USER32 and system call service routines from Win32k.sys have a near perfect match. The only difference is that a few system services do not seem to be called in user mode system DLLs.

Spying on the kernel graphics system calls does not provide us with enough new information, because we can spy on user mode system calls quite easily. The benefit of spying on the kernel side of this interface is that the spying would be systemwide, instead of process-specific. On the other hand, it may generate too much noise for your experiment.

What's really interesting in the kernel graphics world is the DDI interface between the graphics engine and device drivers. We mentioned in [Chapter 2](#) that the graphics engine does a substantial job in transforming GDI calls into DDI calls. They are not on the same level of abstraction. In [Section 2.7](#) we presented a simple printer driver that generates HTML documents instead of printer commands. The HTML pages contain lists of DDI calls and hexadecimal dumps of parameters, and 24-bit color bitmaps rendered at 96 dpi. The simple HTML driver is good place to play with the DDI interface.

But this works only with a printer driver with fixed settings. We would certainly prefer a solution which can spy on all graphics drivers, displays, printers, plotters, and even fax machines. In [Chapter 3](#), internal data structures of GDI and the graphics engine were covered in overwhelming detail. We mentioned that for every device context, its kernel object has a pointer to PDEV structure, which keeps all information regarding a physical graphics device for the graphics engine. The PDEV is created after loading a display driver, calling its entry points DrvEnableDriver, DrvEnablePDEV, and finally DrvCompletePDEV. So the PDEV has all the information filled in by a graphics device driver for those calls, including DDI entry points provided by the driver. For Windows 2000, the last block of data in a PDEV structure contains 89 function pointers; for Windows NT 4.0, it can contain up to 65 function pointers.

For API spying, function pointers are very easy to work with. We can handle pointers in DLL's import table, the C++/COM virtual function table. The function pointer array in the PDEV structure is just like a virtual function table. Among the 89 function pointers, quite a few are not used, reserved, or not normally implemented by a device driver. We could do a good spying job if only we could handle 20 to 30 DDI calls. Given this small number, compared with around 300 GDI calls, we could just write 20 to 30 DDI proxy routines, instead of having to deal with assembly routines in kernel address

space.

As with user mode API spying, we need a two-piece solution for DDI spying. A kernel mode device driver needs to be loaded into kernel mode, hook into DDI calling chain, dump calling parameters, and report back. A user mode control program is needed to start, stop, send commands to the kernel device driver, and receive logging information from it.

The kernel mode device driver, DDISpy.SYS, is an enhanced version of the Periscope driver we used in [Chapter 3](#). It accepts a single IO control to read a block of data from kernel address space, which helps greatly in our exploring of the GDI internal data structure. DDISpy accepts four IO control calls, as listed in [Table 4-2](#).

For each DDI call, there is a corresponding proxy routine, which does data logging, calling the original function, and possibly logging the return result. We are not as worried about performance here, because we want to measure the overall GDI performance in user mode. Nor are we worried about preserving registers, knowing that the DDI calling convention only uses registers for function value. No more assembly here, only C code. Here is the not-so-boring part of the DDI proxy:

Table 4-2. DDISpy IO Controls

IO Control Id	Parameters	Function
DDISPY_READ	Address, size	Same as Periscope; read a block of data from kernel address space
DDISPY_START	DDI function table address, count	Overwrite contents of the function table to start spying on DDI calls
DDISPY_END	DDI function table address, count	Restore contents of the function table, to stop spying on DDI calls
DDISPY_REPORT	Size	Send logging data back to the control program

```
typedef struct
{
    PFN pProxy;
    PFN pReal;
} PROXYFN;

PROXYFN DDI_Proxy [] = // in order of DDI function index
{
    (PFN) DrvEnablePDEV,      NULL,
    (PFN) DrvCompletePDEV,    NULL,
    (PFN) DrvDisablePDEV,     NULL,
    (PFN) DrvEnableSurface,   NULL,
    (PFN) DrvDisableSurface,  NULL,
```

```
(PFN) NULL,           NULL,
(PFN) NULL,           NULL,
(PFN) DrvResetPDEV,  NULL,
...
};

// Logging buffer and logging routines

void DDISpy_Start(unsigned fntable, int count)
{
    unsigned * pFuncTable = (unsigned *) fntable;

    // clear buffer

    for (int i=0; i<count; i++)
        if ( pFuncTable[i] > 0xa0000000 ) // valid pointer
            if ( DDI_Proxy[i].pProxy != NULL ) // we have a proxy
            {
                // remember real function to call
                DDI_Proxy[i].pReal = (PFN) pFuncTable[i];

                // hack it quick
                pFuncTable[i] = (unsigned) DDI_Proxy[i].pProxy;
            }
}

void DDISpy_Stop(unsigned fntable, int count)
{
    unsigned * pFuncTable = (unsigned *) fntable;
    for (int i=0; i<count; i++)
        if ( pFuncTable[i] > 0xa0000000 ) // valid pointer
            if ( DDI_Proxy[i].pProxy != NULL ) // we have a proxy
            {
                // restore hacking
                pFuncTable[i] = (unsigned) DDI_Proxy[i].pReal;
            }
}

#define Call(name)  (*(PFN_ ## name) \
                  DDI_Proxy[INDEX_ ## name].pReal)

void APIENTRY DrvDisableDriver(void)
{
    Write("DrvDisableDriver");
```

```
Call(DrvDisableDriver)();  
}  
...  
BOOL APIENTRY DrvTextOut(SURFOBJ *ps0,  
    STROBJ *pstro,  
    FONTOBJ *pfo,  
    CLIPOBJ *pco,  
    RECTL *prclExtra,  
    RECTL *prclOpaque,  
    BRUSHOBJ *pboFore,  
    BRUSHOBJ *pboOpaque,  
    POINTL *pptlOrg,  
    MIX mix)  
{  
    Write("DrvTextOut"); // ...  
  
    return Call(DrvTextOut) (ps0, pstro, pfo, pco, prclExtra,  
        prclOpaque, pboFore, pboOpaque, pptlOrg, mix);  
}  
...
```

Perhaps the use of the “Call” macro here can be excused, if you agree it makes the code easier to read. The macro here directs the pointer to the real DDI function in the DDI_Proxy table, casts it to the right DDI function pointer type, and calls it. Have you noticed the drawback of this high-level-language-based API hacking? The stack frame is duplicated when calling the actual DDI function.

So far, the spy control program, DDIWatcher, is quite easy. It's quite similar to the TestPeriScope program in [Chapter 3](#). The most critical routine is shown here, which is called after the kernel driver has been installed.

```
KDDIWatcher::SpyOnDDI(void)  
{  
    unsigned buf[2048];  
  
    HDC hDC = GetDC(NULL); // we needed a device context  
  
    typedef unsigned (CALLBACK * Proc0) (void);  
  
    Proc0 pGdiQueryTable = (Proc0) GetProcAddress(  
        GetModuleHandle("GDI32.DLL"), "GdiQueryTable");  
  
    assert(pGDIQueryTable); // get GDI handle table addr
```

```
unsigned * addr = (unsigned *) (pGDIQueryTable() +  
    (unsigned) hDC & 0xFFFF) * 16); // table entry for hDC  
  
addr = (unsigned *) addr[0]; // pointer to kernel object  
  
scope.Read(buf, addr, 32); // read 8 DWORDS  
  
#ifdef NT4  
    unsigned pdev = buf[5]; // PDEV *  
    unsigned ftable = pdev + 0x3F4; // Function table  
#else  
    unsigned pdev = buf[7]; // PDEV *  
    unsigned ftable = pdev + 0xB84; // Function table  
#endif  
  
// read the function table for validation, ..DrvScope  
scope.Read(buf, (void *) ftable, 25 * 4);  
  
unsigned cmd[2] = { ftable, 25 };  
unsigned long dwRead;  
  
// start DDI API spying  
IoControl(DDISPY_START, cmd, sizeof(cmd), buf, 100, &dwRead);  
  
// add drawing calls, or move window around on desktop  
// stop DDI API spying  
IoControl(DDISPY_END, cmd, sizeof(cmd), buf, 8, &dwRead);  
  
cmd[1] = sizeof(buf);  
// read logging data  
IoControl(DDISPY_REPORT, cmd, sizeof(cmd), buf,  
    sizeof(buf), &dwRead);  
  
// presenting logging data  
}
```

Now we have a DDI interface spying solution that works on any graphics device driver, one driver at a time. It modifies the GDI engine data structure in RAM. Even if it blue-screens your machine, which is not likely, you will still be able to get your machine back after rebooting. What's needed is much better DDI data type logging and a better user interface.

4.6 SUMMARY

This chapter provides various tools for you to use to explore GDI's control flow. [Section 4.1](#) introduces a general framework for API spying. [Section 4.2](#) provides a solution to spying on every GDI call made within a process, both from within GDI32.DLL and outside modules. [Section 4.3](#) switches focus to COM interfaces which are used by Direct Draw. [Section 4.4](#) digs into how to monitor graphics system service calls. Finally, [Section 4.5](#) ends with hooking into the DDI interface with a new kernel mode driver.

With the tools developed in this chapter, you can spy on the Win32 GDI/Direct Draw API to see how applications use GDI/DirectDraw calls, per module or processwide. You can monitor undocumented graphics system service calls to figure out what GDI32.DLL relies on the graphics engine. If you're more interested in the real details of how graphics engine talks to device driver, we have a simple yet powerful DDI spying solution for you. We even provide tools to automatically generate API definition files for API spies and a mechanism to write your own plug-ins to enhance or customize data type handling.

Although we're focusing on GDI and DirectDraw in this chapter, the solutions provided here are generic and can be applied to other parts of the Win32 API and other COM interfaces.

Get me an API function, I can show you where it goes, or at least I have the right set of tools to check it out. Now you're qualified to say that.

Further Reading

Matt Pietrek's book, *Windows 95 System Programming SECRETS*, devotes a whole chapter on writing a Win32 API spy. Details are given on intercepting function calls through the import directory, different ways of injecting a DLL into another process, using the Win32 debug API to control target process, stub functions, function prototype encoding, logging, etc. The book describes a very sophisticated method of inserting machine codes to a target process to let it load the spying DLL, which has the benefit of being able to spy from the very beginning of a process's life cycle. The Windows hook method used in this chapter works only for a process creating a window, and only after the window is created. Our book is much more complete in hooking into all API calls, instead of just hacking the import table, hooking into system service calls, DirectDraw interfaces, and even the DDI interface.

Jeffrey Richter's book, *Programming Applications for Microsoft Windows*, 5th ed., also has a chapter-long treatment of DLL injection and API hooking. Different ways of injecting DLLs into a foreign process are discussed, and finally a remote thread is used as the best choice. The book covers different techniques of API hooking without going into detail about implementing a generic API logging mechanism. API hooking by overwriting code is mentioned as a non-workable solution

in a multithread environment. [Section 4.2](#) of this book makes this technique work by relocating the code modified, so that restoring the original code is not needed.

To understand the assembly code used in this chapter, consult an Intel CPU reference manual, which can be downloaded from Intel's web page (www.intel.com).

A book on compilers, especially code generation, may help you understand the tricks we are playing with return addresses on the stack.

Sample Programs

As in [Chapter 3](#), the sample programs in [Chapter 4 \(Figure 4-3\)](#) are not normal graphics programming samples. They are hard-core system-level tools to help us to understand the Windows graphics system and Windows operating system internals in general. Again, make good use of them.

Table 4-3. Sample Programs for Chapter 3

Directory	Description
Samples\Chapt_04\Patcher	Library for patching function prolog code to jump to stub code.
Samples\Chapt_04\Skimmer	Program to extract API definition from SDK header files.
Samples\Chapt_04\Diver	Spying DLL to be injected into target process to generate an API logging file.
Samples\Chapt_04\Pogy	API spying control program, installs Windows hook to inject Diver.
Samples\Chapt_04\PogyGDI	GDI data type decoder, loaded by Diver.
Samples\Chapt_04\QueryDDraw	Helps program in generating DirectDraw API definition file.
Samples\Chapt_04\DDISpy	Kernel mode driver for spying on the DDI interface.
Samples\Chapt_04\DDIWatcher	Test program for using DDISpy to monitor the DDI interface.

[< BACK](#) [NEXT >](#)

Chapter 5. Graphics Device Abstraction

The main Windows graphics programming API is GDI, which stands for Graphics Device Interface. DirectDraw is Microsoft's new two-dimensional game-oriented programming API, while Direct3D is for games and applications that need three-dimensional display. These graphics APIs are device-independent programming interfaces, which allow applications written in them to run on different graphics devices.

To make a graphics API device independent, it needs a good graphics device abstraction which can represent a variety of graphics devices, hide their differences, yet not sacrifice performance.

This chapter will examine GDI's main mechanism of graphics device abstraction: device context. We will look at features provided by modern video display cards, how device context provides an abstract graphics device, and the interaction of device context with the OS window management module.

5.1 MODERN VIDEO DISPLAY CARD

A video display card is the main target for Windows graphics API, because it's practically a mandated primary user interface to interact with a computer. If you have not indulged yourself too much in high-end computer games, or bought a new machine recently, you may be surprised to learn how sophisticated video display cards have become. While the PC industry has barely seen 64-bit machines on the horizon, your display card is claiming to use 128-bit processor architecture. While you may still be running Windows NT using 32 MB of RAM to juggle all your programs, your display card has kept the same amount of ultrawide 128-bit RAM all to itself. The most unbelievable thing may be that a display card can reach 9 giga floating-point operations per second, while your Windows NT kernel mode code is simulating floating point using integer operations.

Now let's look at the basic components of a modern video display card.

Frame Buffer

All modern video display cards are raster based, which means information is stored in two-dimensional arrays of pixels in a region in the display card's random-access memory (RAM). Such a memory region is called a frame buffer.

Frame buffers have different sizes. Screen size is also commonly known as screen resolution. This is totally different from the dot-per-inch (dpi) resolution widely used for printers. When people refer to screen resolution, they are talking about the number of pixels it can display; when they talk about printers, they are more interested in the number of independently addressable pixels per inch.

The smallest frame buffer the Windows OS supports is the standard VGA size, 640 pixels per line by 480 lines, which was first used by IBM for its PS/2 machines. Normally, you see 640 × 480 mode only when your machine is booted in safe mode, or when running old software that forces you to switch your screen to that setting. The largest frame buffer size you see may be 1600 by 1200, or even 1920 × 1200. Note that width and height of most screen resolutions has a 4:3 ratio—for example, 640 by 480, 800 by 600, 1024 by 768, or even 1600 by 1200. These resolutions match the display monitor's width-to-height ratio, which makes individual pixels on the screen have the same spacing in the vertical and horizontal directions.

Depending on the number of colors a frame buffer needs to display, each pixel may be represented by different numbers of bits. A monochrome frame buffer needs only a single bit per pixel; a 16-color frame buffer needs 4 bits per pixel. Neither of them would be seen in new-generation display cards. A frame buffer nowadays has a minimum of 256 colors, with each pixel being represented using 8 bits or 1 byte. Some display cards use the so-called high-color modes, having 15 or 16 bits per pixel to represent 32,768 (32 K) or 65,536 (64 K) colors, although they both use two bytes for each pixel. More and more display cards claiming true color modes use 24 bits to represent 224 (16 M) different colors. Some display modes even use 32 bits per pixel, but not to represent 232 different colors; 32-bpp (bits per pixel) modes normally use 8 bits to store the alpha channel, leaving 24 bits to represent color information.

A frame buffer needs to be mapped to CPU address space for the graphics device driver to draw into it. In the old days of PCs, only 20-bit address lines were used, which gives an address space of 1 MB. A display card was allocated only 64 or 128 KB of the total 1-MB address space. An early 1024-by-768, 256-color super-VGA-class display card would need a frame buffer of 768 KB, much too big for the 128-KB pigeonhole. So, instead of storing a

frame buffer in a continuous chunk of 1024 by 768 by 1 byte, a hardware vendor had to divide it into eight planes of 1024 by 768 by 1 bit. Each plane is now 96 KB, which make your PC able to use the display card. Now that each pixel is divided into eight planes, writing a single pixel to a frame buffer needs to tell the hardware register to map each plane into CPU address space, update one bit, and then move to the next plane. The hardware vendor may also divide a large frame buffer into several banks, or use planes and banks at the same time.

You can imagine that this was a big mess for a supposedly device-independent GDI to operate on. As a result, Microsoft introduced the device-dependent bitmap (DDB), which let hardware vendors provide support for transferring the bitmap quickly to and from its own unique frame buffer format.

With Windows NT/2000, the whole system is basically running in 32-bit address space, including the graphics subsystem. The 4-GB address space provides lots of room for the display cards' frame buffers. With Microsoft's push for DirectX, Microsoft requires new display cards to support linear packed-pixel frame buffers. Packed pixel means all pixels should be together, not divided into different planes. Linear means the whole frame buffer can be mapped into 32-bit linear address space.

As bits per pixel and resolution become higher, substantial memory is required to store a whole frame buffer. [Table 5-1](#) summarizes memory requirements for a single frame buffer in different size and pixel formats.

The highest mode my display card supports is 1920 by 1200, 32-bit frame buffer at 60 Hz refresh frequency, which means that every second the display card is reading the whole 9000-KB frame buffer, converting it to video signal 60 times. This requires the display card's memory throughput to be at least 540 MB per second, which explains why it needs 128-bit high-performance synchronous memory.

For 1024-by-768-pixel screen resolution at 24 bpp, a minimum of 2304 bytes are required to represent a single scan line. Microsoft specification requires the scan line in frame buffer to be 32-bit aligned for better memory performance. So 2304 bytes is fine for the alignment requirement. But there is no need for a scan line to be exactly 2304 bytes long; it only needs to be at least 2304 bytes long. This gives hardware vendors the flexibility to implement scan-line alignment policy. The size of a single scan line in a frame buffer is known as its pitch. [Figure 5-1](#) illustrates a display frame buffer. A frame buffer is an array of scan lines, whose width is "pitch" bytes long. Within a scan line, a continuous number of bits or pixels are used to represent each pixel. For a 24-bpp frame buffer, 3 bytes in the order of blue, green, and red are used to represent a single pixel.

Figure 5-1. Frame buffer geometry.

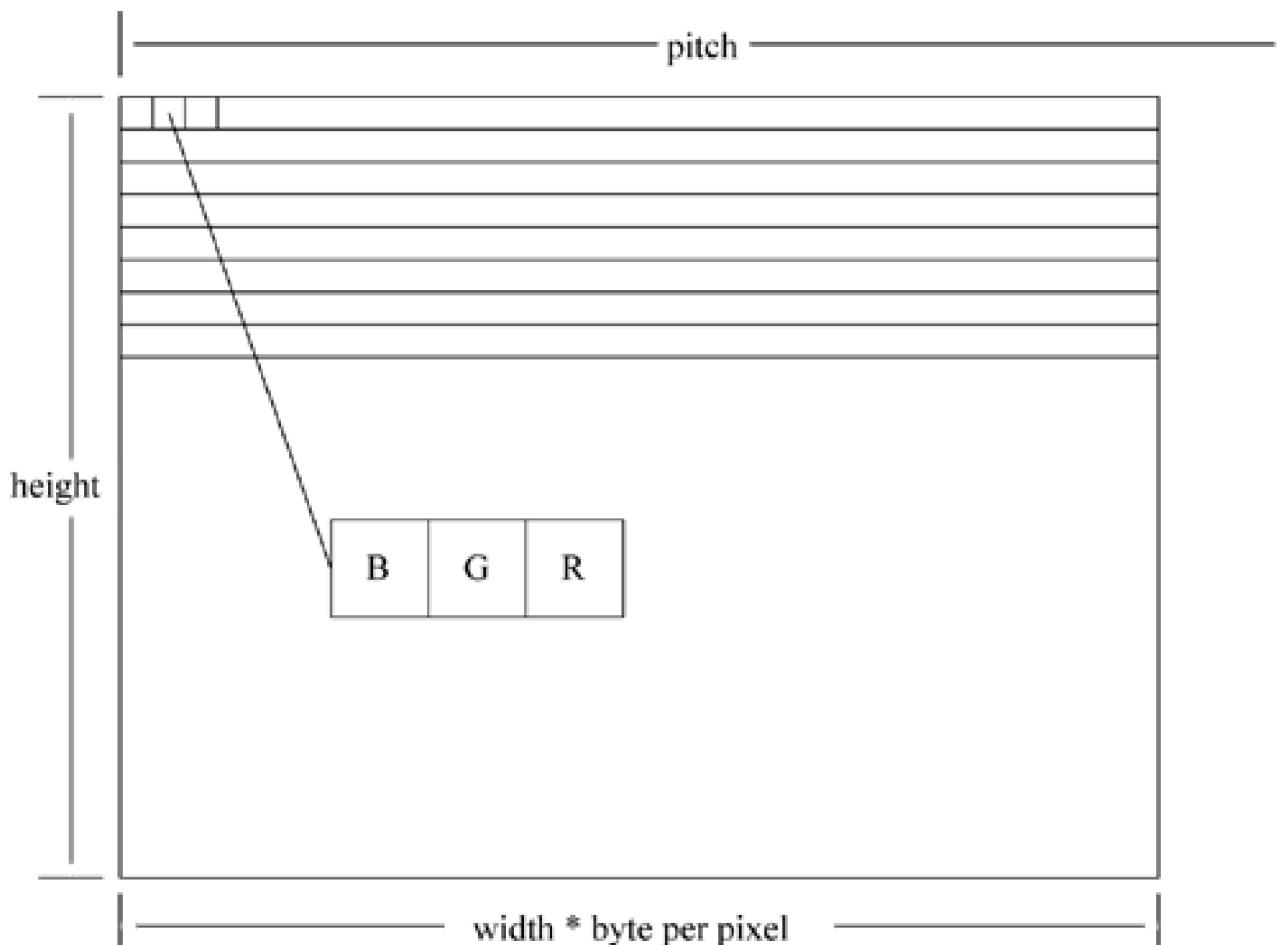


Table 5-1. Frame Buffer Geometry

Resolution	Width/Height Ratio	Frame Buffer RAM Size			
		8 bit	15, 16 bit	24 bit	32 bit
640 × 480	4:3	300 KB	600 KB	900 KB	1200 KB
800 × 600	4:3	469 KB	938 KB	1407 KB	1875 KB
1024 × 768	4:3	768 KB	1536 KB	2304 KB	3072 KB
1152 × 864	4:3	972 KB	1944 KB	2916 KB	3888 KB
1280 × 1024	5:4	1280 KB	2560 KB	3840 KB	5120 KB
1600 × 1200	4:3	1875 KB	3750 KB	5625 KB	7500 KB
1920 × 1080	16:9	2025 KB	4050 KB	6075 KB	8100 KB
1920 × 1200	8:5	2250 KB	4500 KB	6750 KB	9000 KB

Given a frame buffer's starting address, its pitch, a pixel's size, and its relative position in a frame buffer, the following routine calculates the pixel's address.

```
char *GetPixelAddress(char * buffer, int pitch,
int byteperpixel, int x, int y)
{
    return buffer + y * pitch + x * byteperpixel;
}
```

Pixel Format

When you look at an object, light reflected by it reaches your eyes. Light itself is an electromagnetic wave, like radio waves, microwaves, infrared radiation, x-rays, and gamma rays. The human eye can perceive a small section of the whole electromagnetic wave spectrum called visible light, with wavelengths between approximately 400 to 700 nanometers. Different colors correspond to different wavelengths across the spectrum of visible light.

Our eyes contain cells called cones, which are sensitive to these wavelengths and allow us to see color. Three different types of cones are affected by light in the red, green, and blue parts of the spectrum. These form the primary colors. Different light sources give out different parts of the spectrum, which appear as different colors.

In the computer display industry, a color is normally described as an additive combination of red, green, and blue primary color components. We can think of a color as a point within a three-dimensional space, with its axes being the three components. This is the so-called RGB color space.

Computer graphics books normally scale each component to be a floating-point number within 0 and 1, such that an infinite number of colors can be described. But in the digital world that display cards are in, each component is normally scaled to be an integer within 0 and 255, using eight bits or one byte of memory space. In this way, a color is described using three bytes—one for red, one for green, and one for blue. The 24-bit digital RGB color space can describe 16,777,216 different colors.

A monochrome frame buffer uses a single bit of memory to represent each pixel. Eight pixels are packed into a single byte, with the most significant bit (msb) being the first pixel and the least significant bit the last pixel. Although a monochrome frame buffer is no longer likely to be used for your computer display, it still has lots of applications in Windows programming. A multiple-plane frame buffer uses multiple mono chrome frame buffers to represent all the planes in a frame buffer. Monochrome bitmaps are widely used as an in-memory bitmap format. Glyphs in fonts are normally converted to monochrome bitmaps before being displayed on the screen or sent to the printer.

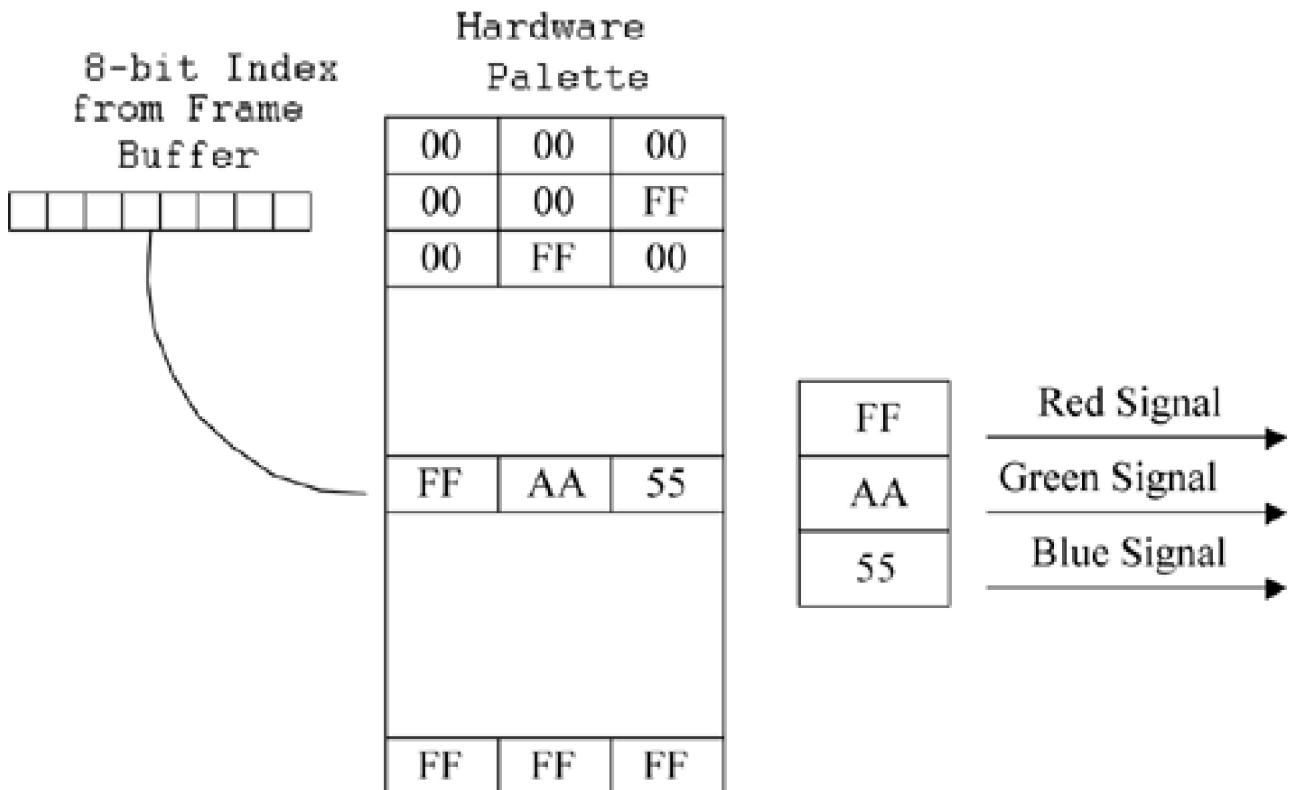
Monochrome printers also use some form of monochrome bitmaps in printer language or internally in the firmware.

In an 8-bpp frame buffer, 256 different colors can be represented at the same time. If these colors are fixed, we would have to select a set of colors forming a uniform distribution in the RGB color space, which are too limited to represent the colorful world. Instead of displaying a fixed set of colors, a display card uses a color-lookup table called a palette. For an 8-bpp frame buffer, a palette has 256 entries, each being a 24-bit RGB value. The frame buffer now stores indexes to this palette. With this indirection through a palette, although a frame buffer still can represent only 256 different colors at any one time, the colors can be picked from 16 million candidates available for a 24-bit frame buffer. For example, you can set up a 256-level grayscale palette to display an x-ray image, or a mostly warm color palette to show a picture of a sunset.

When a display card needs to refresh from a palette-based frame buffer, it needs to read indexes from the frame buffer, pass them to a color-lookup table, and send its output to the video port. This can be implemented very efficiently in hardware logic. To control the hardware palette, the device driver needs to provide entry points to upper-level software to access its hardware palette.

If graphics drawing commands use RGB values instead of palette indexes, they need to translate color RGB value to palette indexes for pixel writing, and from palette indexes to RGB value for pixel reading. Translating RGB value to palette indexes involves table searching to find the exact match or the closest match. If an exact match cannot be found, a color can be simulated using an array of pixels with color from the palette, using a dithering algorithm. Translating palette indexes to RGB value is simple table indexing. [Figure 5-2](#) illustrates palette lookup in an 8-bpp frame buffer.

Figure 5-2. Palette lookup in a display card with 8-bit frame buffer.



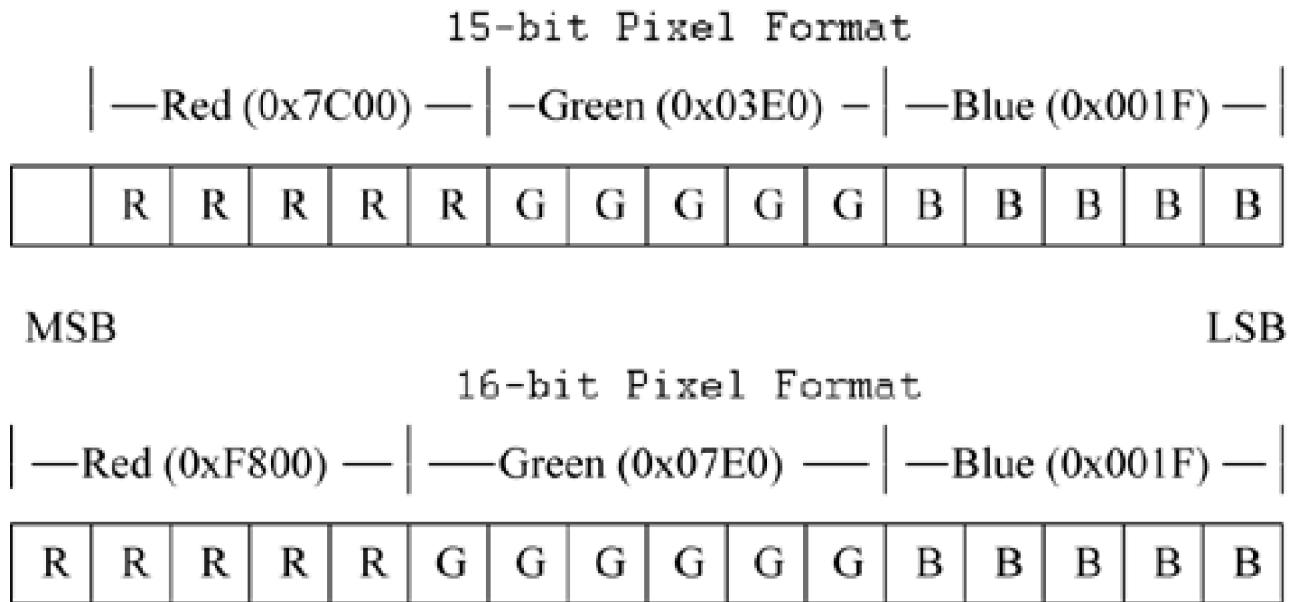
In a 15-bit high-color frame buffer, each primary color component is represented using 5 bits. One pixel is stored in a 16-bit word with its most significant bit undefined, followed by 5 bits for red, 5 bits for green, and the least significant 5 bits for blue. This 15-bit pixel format is denoted as 5:5:5. A 15-bit color frame buffer can describe 32,768 different colors.

The 16-bit high-color format is a small improvement over the 15-bit format. Instead of wasting the most significant bit within a 16-bit word, the green component is changed to be 6-bit since human eyes are more sensitive to green light. A 16-bit frame buffer still uses one 16-bit word for each pixel, normally in 5:6:5 format.

Compared with the true-color frame buffer, using high-color pixel formats allows us to use less memory to support an adequate number of colors in quite high screen resolutions. For example, a video display card with a mere 2 MB of memory can support up to 1152 by 864 pixels in 16-bit mode. The down side is the frame buffer access time. Writing a color pixel in 24-bit RGB format to a high-color frame buffer is not a simple memory store or copy. You need to reduce the 8-bit color component to be 5 or 6 bits, combine them according to the pixel format, and then store the data. Converting a pixel in a high-color frame buffer to a 24-bit RGB format requires masking out each color component and boosting it to 8 bits.

Although different arrangements of pixel order are possible, Microsoft requires hardware manufacturers to implement a frame buffer in a fixed order. It further requires a display card, which supports a 15-bit frame buffer, but not a 16-bit frame buffer, to report a 15-bit frame buffer as a 16-bit frame buffer. These requirements make software more compatible with different hardware. [Figure 5-3](#) shows pixel formats for 15-bit and 16-bit frame buffers, together with bit masks to access red, green, and blue components.

Figure 5-3. Pixel formats in high-color frame buffers.



For high-quality photographic applications or games, 15-bit and 16-bit frame buffers are not enough to represent the variety of colors, and the smooth transition in colors, they require. For example, if you use a 15-bpp frame buffer to display a gray scale image, only 32 levels of gray can be displayed, because each RGB component is stored using 5 bits of information, allowing for 2^5 levels of intensity. These applications simply need the best, a 24-bit or 32-bit true-color frame buffer. Both 24-bit and 32-bit frame buffers use 8 bits to represent their red, green, or blue components. For early display cards, the extra 8 bits in a 32-bit pixel were normally left unused. For new display cards for Windows 98 or Windows 2000 machines, the extra 8 bits can serve as a transparency component.

The transparency component is normally known as the alpha channel. It specifies the ratio in which a source pixel should affect a destination pixel when drawn into a destination surface. The lowest alpha value, 0, means the pixel should be transparent, not drawn into the destination surface. The highest alpha value, 255, in an 8-bit alpha channel, means the pixel is opaque, which would overwrite the destination pixel completely. Anything between them means the new destination pixel should be computed as the weighted sum of the source pixel and the original destination pixel.

The 24-bit pixel format can be denoted as “RGB” format, the 32-bit pixel format as “ARGB.” [Figure 5-4](#) illustrates both of them.

Figure 5-4. 24-bit and 32-bit pixel formats.

24-bit Pixel Format



MSB

LSB

32-bit Pixel Format



Having 16,777,216 different colors that are representable in a 24-bit or a 32-bit color frame buffer would seem to be enough for everyone, from weekend photographer to master-class professional. But people start to notice that with high-color or true-color frame buffers, they have lost the flexibility a hardware palette brings to them. With a palette, making small changes can affect the whole screen display immediately. For example, if a graphic artist wants to adjust color saturation of a picture a little bit, he or she only needs to adjust RGB values for a maximum of 256 palette entries. But with a traditional high-color or a true-color frame buffer design, you have to change every pixel in the frame buffer, which amounts to 2304 KB of data in a 1024 by 768 by 24-bit frame buffer.

Another problem high-quality color display faces is matching the color on screen with what's to be printed on paper. When color is produced on an electronic device, our eyes perceive it differently than when it's printed on paper. Graphic artists use the so-called gamma correction to add another translation to a frame buffer's color pixels. To make the translation table size reasonable, each of the RGB components is translated separately, using a unique translation table, which requires three tables of 256 bytes each. A gamma correction is called a gamma ramp. The hardware chip used to implement such translation is called RAM digital-to-analog converter, or RAMDAC. Microsoft requires display cards to provide downloadable RAMDAC for true-color frame buffers, to implement gamma correction in hardware.

Double Buffering, z-Buffering, and Texture

Games rely heavily on animation—small images or even screens that change over time. For each frame in an animation sequence, a game program needs to erase certain parts within the frame buffer and redraw new objects. In a single frame buffer design, a video display signal is generated from the frame buffer while the program is clearing and redrawing it. This creates an ugly flickering effect.

The solution to this problem is using two buffers, a front buffer and a back buffer. A user is always shown a finished front buffer, while the application is working in the back buffer. Once the drawing is done, the front and back buffers are swapped. Now the user is shown a new finished buffer, and the program can start to work on the new back buffer. In this way, a user never sees work-in-process, so a smooth animation sequence is presented.

This technique of using a front and a back buffer is called *double buffering*. For the new generation of display cards, hardware is required to support double buffering for the whole frame buffer. Double buffering simply doubles the required video RAM size. According to the information in [Table 5-1](#), a video display card now needs 17.5 MB to support a 1920 by 1200 by 32-bit frame buffer.

When hardware is asked to swap the front and back buffers, it needs to wait for *vertical retrace*, the exact time when one cycle of refreshing is finished and a new cycle of refreshing starts. Failing to synchronize buffer swapping with vertical retrace will cause an unpleasant tearing effect. While waiting for the next vertical retrace, neither of the two buffers can be written to, so the CPU is wasting time waiting. A triple buffering scheme can be used to reclaim the synchronization time, by allowing writing to the third frame buffer. Now the first frame buffer is what's shown to the user, the second buffer is waiting to be shown, and the third buffer is where the rendering is happening.

A scene in a 3D game is composed of various objects at different distances from the viewer. An object closer to the viewer blocks objects that are further away, which will be hidden or partially hidden. To draw a 3D scene as described here, programs need to sort drawn objects according to their distance from the viewer. This process is very complicated and time consuming. To make things more complicated, pixels on a drawing object may have different distances from the viewer, depending on the location of the object. When two objects meet, they may partially block each other.

An efficient solution to this problem, again, is trading memory for time, using a per-pixel depth buffer, commonly known as *z-buffer* for its being the third dimension. A z-buffer stores the distance from a pixel on an object to the viewer, or its depth. When a new pixel needs to be drawn, its depth is compared with the corresponding depth in the z-buffer. Only pixels having smaller depth actually get drawn, updating the z-buffer at the same time.

The memory size for a z-buffer depends on how many levels of distance a program wants to distinguish. An 8-bit z-buffer allows 256 levels of distance, too few for complicated usage. A 16-bit z-buffer boosts the level of distance to 65,536, which is very common for display cards on the market now. But when games get more and more detailed, even a 16-bit z-buffer is not enough. Objects can be drawn with the wrong depth order, in what's called *z-aliasing*. We will see more and more display cards with 24-bit or 32-bit z-buffers. Some display cards support floating-point z-buffers to improve depth accuracy.

A 16-bit z-buffer adds another 0.6 to 4.4 MB to video RAM size. A 32-bit z-buffer doubles that figure to 1.2 to 8.8 MB.

Three-dimensional characters or scenes are made up of 3D surfaces, normally built from thousands of triangles. Bitmaps called *textures* are then applied to individual triangles, to make a surface look like human skin, clothing, a sandy beach, or a brick wall. With hardware acceleration, video display hardware is responsible for applying texture instead of your PC's CPU. Quick access to vast amounts of texture is the key to performance here, which requires the display card to store texture bitmaps in video memory, instead of fetching them from system memory through the slow and crowded system bus.

So video display RAM is not used only for front buffers, back buffers, and z-buffers, it also holds tons of texture bitmaps. [Table 5-2](#) shows possible configurations in which your display RAM could be carved out.

[Table 5-2](#) shows that if you have 32 MB of RAM in a display card and you set the display resolution to 1152×864 in 32-bpp mode, the front buffer consumes 3888 KB, the two back buffers need 7777 KB, plus the 32-bit z-buffer wants another 3888 KB; now you only have 17,216 KB for texture bitmaps. But if you change the screen resolution to 1600×1200 , which is still much lower than the highest 1920×1440 , only 2768 KB are left for texture bitmaps. One way to solve this problem is with texture compression to reduce the size of bitmaps. Another way to make it faster is to load textures from system RAM to display RAM. The current PC hardware architecture uses a PCI (peripheral component interconnect) bus to transmit data, including data from system RAM to video RAM. The PCI bus has only a 100-Mbps transfer rate. The new AGP (accelerated graphics port) bus designed by Intel is a dedicated, high-speed bus which allows quick access of texture bitmaps residing in system RAM. For example, AGP 2X has a burst rate of 528 Mbps.

Table 5-2. Display RAM Budget

Usage	8 MB RAM	16 MB RAM	32 MB RAM
Front buffers	1,875 KB, 800 × 600 × 32	3,072 KB, 1024 × 768 × 32	3,888 KB, 1152 × 864 × 32
Back buffers	3,750 KB, 800 × 600 × 32 × 2	6,144 KB, 1024 × 768 × 32 × 2	7,776 KB, 1152 × 864 × 32 × 2
z-buffers	938 KB, 800 × 600 × 16	2,304 KB, 1024 × 768 × 24	3,888 KB, 1152 × 864 × 32
Textures	1,629 KB	4,864 KB	17,216 KB

Video display cards may also support overlay surfaces, which are display surfaces superimposed on a main computer display. Overlaying may be used to superimpose a TV screen onto your normal computer screen.

Hardware Acceleration

A modern display card must do more than provide a dumb frame buffer to allow software to draw into it and generate video signals from it. Otherwise, even the fastest general-purpose PC CPU would not have enough power to drive a 3D animation sequence at a reasonable frame rate.

Here is a list of features commonly supported by video display hardware:

- Cursor display, including cursors with alpha channel.
- 2D graphics: lines and curves with possible fractional coordinates, area fills, bitmap bitbltting, alpha blending, gradient fills, multiple buffering, and loadable RAMDAC.
- Text display, including antialiasing text using multiple-level glyphs.
- 3D graphics: triangle setup engine, 3D graphics pipeline, various texture blending options, per-pixel perspective-correct texture mapping, z-buffer, stencil buffer, edge antialiasing, per-pixel antialiasing, anisotropic filtering, palletized texture, etc.
- Video: MPEG decoding, DVD decoding, smooth scaling with filtering, multiple video windows with color space conversion and filtering, per-pixel color keying, overlays, etc.

Display Device and Settings Enumeration

Windows 2000 supports multiple display devices on a single system, which can be used for main desktop display, secondary monitor display, and display mirroring for NetMetting.

A multiple-display monitor supports plugging several display cards into your PC, each attached to a different monitor. These monitors can either form a large virtual desktop, of which each monitor displays one portion, or work independently. The first situation is useful for applications needing a bigger desktop than what's offered by a single monitor—for example, desktop publishing, wide-format printing, or CAD/CAM applications. The second situation is helpful for playing video games, debugging applications, teaching, or making presentations.

Display mirroring is critical to share your screen display with another person over the network, in which all the drawing calls are sent as network packages to another location, where the exact screen display is pieced together. The Windows GDI/DDI (device drive interface) interface was initially designed as a local display protocol, in that GDI drawing commands are supposed to be sent to the display driver on the same machine. This is different from the XWindow display protocol used in the UNIX world, which is designed to be a remote-display protocol. On a UNIX workstation running XWindow, it's very easy for a person working at home to log onto a machine miles away in a company's office and have the screen display data sent back to that person's home, such that the machine can be operated remotely. Applications are available that allow you to run Microsoft Windows as an XWindow terminal on your PC. You can even share your XWindow display with multiple parties, allowing each of them to change it. Before display mirroring is officially supported, an application developer needs to hack into or rewrite a display driver to secretly tunnel display commands over the network. Now Windows 2000 supports a separate mirroring driver, which can see data sent to the actual display driver.

Each display device has a unique name that can be referred to in a user application, in the form "\.\DISPLAYx," where x is number starting from 1. Note, in your C/C++ program, that the string should be written as "\\\.\DISPLAYx," because the "\" starts an escape sequence in a character string.

Windows 2000 provides a new function, `EnumDisplayDevices`, to enumerate display devices installed on the system. The sample routine shown below enumerates display devices and adds their names to a combo box:

```
void AddDisplayDevices(HWND hList)
{
    DISPLAY_DEVICE Dev;

    Dev.cb = sizeof(Dev);

    SendMessage(hList, CB_RESETCONTENT, 0, 0);

    for (unsigned i=0;
        EnumDisplayDevices(NULL, i, & Dev, 0); i++ )
        SendMessage(hList, CB_ADDSTRING, 0,
                    (LPARAM) Dev.DeviceName);
    SendMessage(hList, CB_SETCURSEL, 0, 0);
}
```

For each display device, `EnumDisplayDevices` fills a `DISPLAY_DEVICE` structure with its device name (e.g., "\.\DISPLAY1"), device description string (e.g., "NVIDIA RIVA TNT"), state flags (e.g., "ATTACHED_TO_DESKTOP | MODESPRUNED | PRIMARY_DEVICE"), plug-play device identifier, and registry key for the device.

Once you have a display device's name, `EnumDisplaySettings` can be used to enumerate all frame buffer formats supported by the device, together with the corresponding display frequency. The following sample routine does the enumeration and adds a summary string to a list box for each setting.

```
int FrameBufferSize(int width, int height, int bpp)
{
    int bytepp = ( bpp + 7 ) / 8;      // bytes per pixel
    int bytesp = ( width*bytepp+3)/4*4; // bytes per scanline
```

```
return height * bytes;           // bytes per frame buffer
}

void AddDisplaySettings(HWND hList, LPCTSTR pszDeviceName)
{
    DEVMODE dm;

    dm.dmSize      = sizeof(DEVMODE);
    dm.dmDriverExtra = 0;

    SendMessage(hList, LB_RESETCONTENT, 0, 0);

    for (unsigned i = 0;
        EnumDisplaySettings(pszDeviceName, i, & dm);
        i++)
    {
        TCHAR szTemp[MAX_PATH];

        wsprintf(szTemp, _T("%d by %d, %d bits, %d Hz, %d KB"),
            dm.dmPelsWidth,
            dm.dmPelsHeight,
            dm.dmBitsPerPel,
            dm.dmDisplayFrequency,
            FrameBufferSize(dm.dmPelsWidth, dm.dmPelsHeight,
            dm.dmBitsPerPel) / 1024
        );

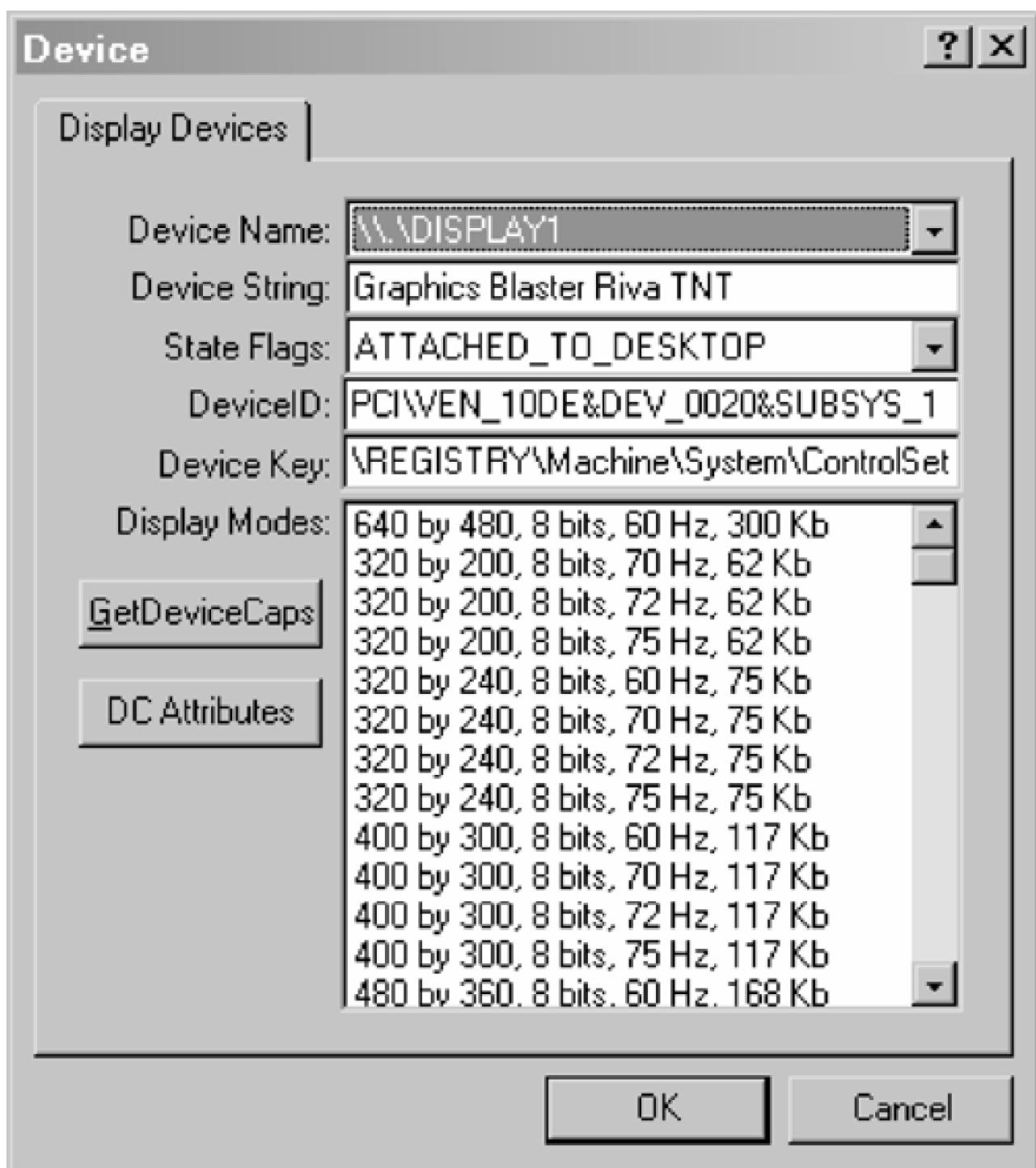
        SendMessage(hList, LB_ADDSTRING, 0, (LPARAM) szTemp);
    }
}
```

For each display setting, `EnumDisplaySettings` fills a `DEVMODE` structure with its frame buffer width, height, bits per pixel, display frequency, and a few other fields. The code shown also calculates the size of a single frame buffer based on these results.

Note here that `DEVMODE` is a very tricky structure to use in your program. What's documented in Win32 API is called the public `DEVMODE`. The graphics device driver can attach private `DEVMODE` data after the public `DEVMODE` fields by declaring the size of extra data in the `dmDriverExtra` field. Before calling `Enum DisplaySettings`, the code here sets `dmSize` field to the size of (public) `DEVMODE`, and `dmDriverExtra` to zero to tell the display driver no extra data is expected.

A sample program, `DEVICE`, that uses `EnumDisplayDevices` and `EnumDisplay Settings`, can be found on the companion CD. Its first property-sheet page, named “Display Devices,” lists display devices installed on the system, and frame buffer frames supported. [Figure 5-5](#) is a sample display.

Figure 5-5. Enumerating display devices and modes.



5.2 DEVICE CONTEXT

Video display cards are only one class of graphics devices supported by the Windows graphics system. Another major class of graphics devices is hard-copy devices, like printers, plotters, and fax machines.

The Windows NT/2000 graphics system uses a layered architecture. The top layer is composed of 32-bit client DLLs, exposing API functions to user applications. For example, GDI32.DLL exposes GDI API for traditional 2D graphics; DDRAW.DLL exposes DirectDraw API for 2D game programming; and D3DRM.DLL, D3DIM.DLL exposes Direct3D API for 3D game programming. Client DLLs are mapped into the user mode portion of an application's address space. The middle layer is the graphics engine, which lives in kernel mode address space to support graphics APIs for the whole system. The graphics kernel mode address space provides hundreds of graphics system service routines, which can be invoked by the client DLLs. On Windows NT/2000, the graphics engine and the kernel mode portion of window management form a huge kernel mode DLL WIN32K.SYS. The bottom layer of the graphics system consists of graphics device drivers provided by hardware manufacturers, which implement the device driver interface (DDI) as specified in Microsoft's Device Driver Kit (DDK). For detailed discussion of Windows graphics system architecture, graphics system services, DDI interface, and graphics device drivers, refer to [Chapter 2](#) of this book.

To interact with a graphics device driver, the Windows NT/2000 graphics system uses an internal data structure commonly known as a *Device Context*. A device context is actually a complicated web of structures and objects linked using pointers, spreading in both user mode application address space and kernel mode system address space. [Chapter 3](#) of this book has detailed descriptions of the internals of device context and other graphics system data structures.

A device context serves two important purposes within the graphics system. The main purpose is to provide an abstraction of a graphics device, such that anything above the device driver, which includes the graphics engine, the Win32 client DLLs, and the user application, can be device independent. Another usage of a device context is for storing commonly used drawing attributes, like foreground color, raster operation, pen, brush, font, etc., such that individual drawing calls do not have to carry these settings over and over again.

The Win32 GDI client DLL hides away actual device contexts from user applications. Application programs only get device context handles, which are 32-bit values with undocumented format. A handle is returned when a device context is created by GDI, then passed back to GDI for any further queries of drawing operations. This handle mechanism provides more information hiding than "this" pointer in C++ and COM interface pointer. It also allows great flexibility for Microsoft to provide different but compatible implementations on different systems. For example, a Win32 program could run on both Windows 95/98 machines and Windows NT/2000 machines, although device context handles are implemented totally differently on these two classes of systems.

Device Context Creation

A device context is created using the Win32 CreateDC function:

```
HDC CreateDC(LPCTSTR pszDriver,  
             LPCTSTR pszDevice,  
             LPCTSTR pszOutput,
```

```
CONST DEVMODE * pdvmlInit);
```

The first parameter, pszDriver, is not really a graphics device driver name for Win32 programs. It is a leftover from the Win16 API. The only acceptable value for it is "DISPLAY" for creating a display device context, NULL or "WINSPOOL" for a printer device context.

The second parameter, pszDevice, is the name of a graphics device. It could be a display device name returned by EnumDisplayDevices, or a printer name as shown in control panel printer applet. For example, "\\.\DISPLAY1" or NULL means the primary display device; "\\.\DISPLAY2" is normally the secondary display device; "\\.\DISPLAY3" could be a NetMetting driver that allows mirroring screen display onto other people's monitor.

The third parameter used to be the port name where a printing job should be sent. Win32 API uses a new function StartDoc call to pass the port name, so this parameter should always be NULL.

The last parameter, pdvmlInit, is a pointer to a DEVMODE structure, which specifies device-creation parameters. Normally NULL is passed to tell the device driver to use the current device settings that the operating system keeps in the registry. For a display device, pdvmlInit could point to the DEVMODE structure returned by Enum Display Settings, which specifies the height, width, bits per pixel, and video output frequency. For a hard-copy device, pdvmlInit could point to the DEVMODE structure returned by the DocumentProperties function.

CreateDC returns a GDI device context handle, which could be used in subsequent calls to GDI, until the last call DeleteDC that frees system resources used for the device context and makes the device context handle invalid.

The actual process of creating a device context is very complicated. From the device name, the operating system could find out the device driver name through registry and load the driver. For a video display, only the first CreateDC needs to load the display driver, which will then be reused. For a printer device, the printer driver is actually loaded and initialized by the CreateDC call. The printer spooler and the printer driver user interface DLL are also informed about the creation of a new printer device context.

When a graphics device driver is loaded, its main entry point DrvEnableDriver is called. DrvEnableDriver fills a structure that holds its version number and entry points of all DDI functions implemented by the driver. The graphics engine then calls DrvEnablePDEV to ask the driver to describe its attributes and features and create its data structure for a physical device. The pdvmlInit and pszDevice parameters in the CreateDC call are passed to DrvEnablePDEV. DrvEnablePDEV fills two important structures, GDIINFO and DEVINFO, with attributes, capabilities, pixel formats, and default settings about the device. The graphics engine creates its internal data structure for a physical device, which keeps information returned by both DrvEnableDriver and DrvEnablePDEV. Now it knows the graphics device's features and capabilities, and which entry points to call for different DDI drawing calls. In the final stage of device context creation, the graphics engine calls driver's DrvEnableSurface to ask the driver to create a drawing surface, where the rendering actually happens. For details, refer to [Chapter 2](#) on the DDI interface and [Chapter 3](#) on the internal data structures.

Querying a Device's Capabilities

A device context keeps or has links to a large amount of information about a graphics device and its driver. Some of this information can be queried using Win32 API calls; other information is kept internally by GDI to facilitate its interaction with the driver.

Function GetDeviceCaps is provided to query for a graphics device or its driver's attributes or capabilities, through an integer index:

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

GetDeviceCaps provides a way for an application to know the specific details about a graphics device—for example, frame buffer format, color-handling capabilities, resolution, palette, physical size, paper margins, alpha blending and gradient fill capabilities, ICM (image color management) support, and DDI features and limitations. Most of the capabilities should not be the application program's business to worry about; they are just hints to the graphics engine during its interaction with the device driver. Some of the flags are really for 16-bit graphics device drivers used by Windows 3.1, Windows 95, and Windows 98. For example, the CC_ELLIPSES flag in CURVECAP query is supposed to indicate whether a device driver can draw an ellipse. This flag does not exist in Windows NT/2000 DDI interface at all, because all elliptical curves will be converted into Bezier curves before reaching a graphics device driver. When an application queries for indexes like CURVECAP, Windows NT/2000 just gives a standard answer, to make sure no old application will be broken.

[Table 5-3](#) explains indexes and return values in GetDeviceCaps.

For screen display, calling GetDeviceCaps to check whether a device context has a hardware palette is very critical. A program drawing into such a device context needs to create the right logical palette, select it into device context before drawing calls, and respond to palette-related messages.

For printing to a hard-copy device, a well-behaved program should always check for the exact paper dimensions and margins before formatting the document. Note that an application can change page orientation from the default portrait mode to landscape mode, which switches paper width and height, left and top margins.

For applications running under Windows NT/2000, it's less important to check a device context's drawing capabilities like CURVECAPS, LINECAPS, POGYON CAPS, CLIPCAPS, etc. The graphics engine is doing a great job helping the graphics device driver to render all GDI drawing commands.

Checking a device context's capabilities can also help the application fine-tune its performance. For example, if a device does not provide support for gradient fill, an application could either simulate gradient fill, simplify gradient fill, or not use it at all. An application running in 8-bit, 256-color mode may use clipart and images with fewer colors instead of 24-bit color.

On the Windows NT/2000 system, the graphics engine knows much more than what's reported by GetDeviceCaps, which was initially designed for a 16-bit GDI. A graphics device driver needs to report how styled lines should be drawn, numerous half toning settings, device's color in Commission Internationale de L'Eclairage (CEI) color space, and some internal graphics capabilities. For example, the graphics engine needs to know whether a device supports opaque rectangle in text drawing call, whether EMF spooling is supported, whether arbitrary brush can be used to paint text opaque rectangle, whether text drawing can support fractional coordinates, whether 4-bit antialiasing text output is supported, etc.

Using GetDeviceCaps is quite straightforward. In the DEVICE sample program, the "GetDeviceCaps" button on the "Display Devices" page pops up a dialog box that shows all available device-capability flags in a list view, as shown in [Figure 5-6](#).

Figure 5-6. Querying graphics device capabilities.

Index	Value
TECHNOLOGY	1
DRIVERVERSION	0x4000
HORZSIZE	320 mm
VERTSIZE	240 mm
HORZRES	1152 pixels
VERTRES	864 pixels
LOGPIXELSX	96 dpi
LOGPIXELSY	96 dpi
BITSPIXEL	32 bits
PLANES	1 planes
NUMBRUSHES	-1
NUMPENS	-1
NUMMARKERS	0
NUMFONTS	0
NUMCOLORS	-1
PDEVICESIZE	0
CURVECAPS	1ff
LINECAPS	fe

Attributes in Device Context

A drawing command in GDI needs two kinds of information: attributes and settings that are common across different commands, and information that is unique to individual commands. Apparently, it would be very inefficient to specify those common attributes and settings over and over again. Windows GDI uses device context to store the following common attributes and settings:

Table 5-3. GetDeviceCaps: Indexes and Return Values (Windows NT/2000)

Index	Sample	Return value and meaning
DRIVERVERSION	0x4001	Driver version, 16-bit in 0xXYZZ format, where X is the OS major version, Y is the OS minor version, and ZZ is the driver version number. Reported by the driver.
TECHNOLOGY	DT_RASDISPLAY	Reported by the driver.
		DT_PLOTTER for plotters.
		DT_RASDISPLAY for raster display cards.
		DT_RASPRINTER for raster printers.
		DT_RASCAMERA for the raster camera.
		DT_CHARSTREAM for the pure device font driver.
HORZSIZE	320	Width of the physical surface, in millimeters. Not accurate for the display. Reported by the driver.
VERTSIZE	240	Height of the physical surface, in millimeters. Not accurate for the display. Reported by the driver.
HORTRES	1024	Width of the physical surface, in pixels. Reported by the driver.
VERTRES	768	Height of the physical surface, in pixels. Reported by the driver.
BITSPIXEL	8, 16, 24, 32	Number of adjacent bits in each color plane. Reported by the driver.
PLANES	1	Number of color planes. Reported by the driver.
NUMBRUSHES	-1	Number of device-specific brushes.
NUMPENS	-1	Number of device-specific pens. Number of pens for plotter.
NUMFONTS	0	Number of device-specific fonts.
NUMCOLORS	-1	Number of entries in the device's palette, or -1 for no palette.
CURVECAPS	0x1FF	Curve-drawing capabilities. Standard answer: CC_CHORD CC_CIRCLES CC_ELLIPSES CC_INTERIORS CC_PIE CC_ROUNDRECT CC_STYLED CC_WIDE CC_WIDESTYLED
LINECAPS	0xFE	Line-drawing capabilities. Standard answer: LC_POLYLINE LC_MARKER LC_POLYMARKER LC_WIDE LC_STYLED LC_WIDESTYLED LC_INTERIORS
POLYGONALCAPS	0xFF	Polygon-drawing capabilities. Standard answer: PC_POLYGON PC_RECTANGLE PC_WINDPOLYGON PC_SCANLINE PC_WIDE PC_STYLED PC_WIDESTYLED PC_INTERIORS. Note that the PC_POLY POLYGON and PC_PATHS are missing from the standard answer, which does not mean the device can't handle them. It's just a bug in reporting.
TEXTCAPS	0x7807	Text-string drawing capabilities. Reported by the driver.
CLIPCAPS	1	Clipping capabilities. Standard answer: CP_RECTANGLE. Note that this does not imply the device can't handle clipping by complex region.
RASTERCAPS	0x7e99	Raster capabilities. Partly reported by the driver, partly standard answer: RC_BITBLT RC_BITMAP64 RC_GDI20_OUTPUT RC_DI_BITMAP RC_DIBTO_DEV RC_BIGFONT RC_STRETCHBLT RC_FLOODFILL RC_STRETCHDIB RC_OP_DX_OUTPUT.
ASPECTX	36	Relative width of a device pixel, in the range of 1 to 1000. Reported by the driver.

ASPECTY	36	Relative height of a device pixel, in the range of 1 to 1000. Reported by the driver.
ASPECTXY	51	Relative diagonal of a device pixel, $\text{Sqrt}(\text{ASPECTX}^2 + \text{ASPECTY}^2)$. Reported by the driver.
LOGPIXELSX	96 or 120	Logical width resolution in dots per inch (dpi). Reported by the driver.
LOGPIXELSY	96 or 120	Logical height resolution in dots per inch (dpi). Reported by the driver.
SIZEPALETTE	0, 16 or 256	Number of entries in the system palette, valid only when RASTERCAPS has the RC_PALETTE flag.
NUMRESERVED	2 or 20	Number of reserved entries in the system palette, valid only when RASTERCAPS has the RC_PALETTE flag. Answer generated based on the system setting.
COLORRES	24	Number of bits used to represent a color pixel.
PHYSICALWIDTH	0	For a hard-copy device, the width of the whole physical page in a device unit. Reported by the driver.
PHYSICALHEIGHT	0	For a hard-copy device, the height of a whole physical page in a device unit. Reported by the driver.
PHYSICALOFFSETX	0	For a hard-copy device, the unprintable left margin in a device unit. Reported by the driver.
PHYSICALOFFSETY	0	For a hard-copy device, the unprintable top margin in a device unit. Reported by the driver.
SCALINGFACTORX	100	For a hard-copy device, the scaling factor i_x -axis.
SCALINGFACTORY	100	For a hard-copy device, the scaling factor i_y -axis.
VREFRESH	60	Video refresh rate for the current display mode. Reported by the driver.
DESKTOPHORZRES	1024	Width of the entire desktop in pixels, different from the single-screen width in a multiple-monitor setting.
DESKTOPVERTRES	768	Height of the entire desktop in pixels, different from the single-screen height in a multiple-monitor setting.
BLTALIGNMENT	0	Preferred x-alignment for bit-block transfers to the device. A value of zero indicates the device is hardware accelerated, so any alignment may be used. Reported by the driver.
SHADEBLENDCAPS	0	Alpha blending and gradient fill capabilities. Reported by the driver.
COLORMGMT-CAPS	2	Color-management capabilities. Reported by the driver: CM_GAMMA_RAMP,

- Coordinate space, mapping modes, and world coordinate transformation.
- Foreground color, background color, palette, and color management settings.
- Line-drawing settings.
- Area-filling settings.
- Font, character spacing, and text alignment.
- Bitmap stretching settings.
- Clipping.

- Others.

Each attribute has a set of acceptable values and a default value, which a device context will be set to when it is created to form a functional device context. For each attribute, there is normally a Win32 API function to retrieve the attribute, and another function to set the attribute. [Table 5-4](#) lists device context attributes, default values, and access functions.

Table 5-4. Device Context Attributes (Windows 2000)

Attribute	Default	Access Functions
Associated window handle	NULL	WindowFromDC (read only)
DC origin		
Bitmap	1-by-1 bitmap	GetCurrentObject, SelectObject (memory device context only)
Graphics mode	GM_COMPATIBLE	GetGraphicsMode, SetGraphicsMode
Mapping mode	MM_TEXT	GetMapMode, SetMapMode
View port extent	{1,1}	GetViewportExtEx, SetViewportExtEx, ScaleViewportExtEx
View port origin	{0,0}	GetViewportOrgEx, SetViewportOrgEx, OffsetViewportOrgEx
Window extent	{1,1}	GetWindowExtEx, SetWindowExtEx, ScaleWindowExtEx
Window origin	{0,0}	GetWindowOrgEx, SetWindowOrgEx, OffsetWindowOrgEx
Transformation	Identity matrix	GetWorldTransform, SetWorldTransform, ModifyWorldTransform
Background color	System background color	GetBkColor, SetBkColor
Text color	Black	GetTextColor, SetTextColor
Palette	DEFAULT_PALETTE	GetCurrentObject, EnumObjects, SelectPalette
Color adjustment		GetColorAdjustment, SetColorAdjustment
Color space		GetColorSpace, SetColorSpace
ICM mode		SetICMMode
ICM profile		GetICMProfile, SetICMProfile
Current pen position	{0,0}	GetCurrentPositionEx, MoveToEx, LineTo, BezierTo, ...
Binary raster operation	R2_COPYOPEN	GetROP2, SetROP2
Background mode	OPAQUE	GetBkMode, SetBkMode
Logical pen	BLACK_PEN	SelectObject, GetCurrentObject
DC pen color		GetDCPenColor, SetDCPenColor
Arc direction		GetArcDirection, SetArcDirection
Miter limit	10.0	GetMiterLimit, SetMiterLimit
Logical brush	WHITE_BRUSH	SelectObject, GetCurrentObject
DC brush color		GetDCBrushColor, SetDCBrushColor
Brush origin	{0,0}	GetBrushOrgEx, SetBrushOrgEx
Polygon filling mode	ALTERNATE	GetPolyFillMode, SetPolyFillMode
Bitmap stretching mode	STRETCH_ANDSCANS	GetStretchBltMode, SetStretchBltMode
Logical font	System font	SelectObject, GetCurrentObject, GetCharWidth32, GetKerningPairs, GetTextMetrics, ...
Intercharacter spacing	0	GetTextCharacterExtra, SetTextCharacterExtra
Font mapper flags		SetMapperFlags
Text alignment	TA_TOP TA_LEFT	GetTextAlign, SetTextAlign
Text justification	{0,0}	SetTextJustification. Set only

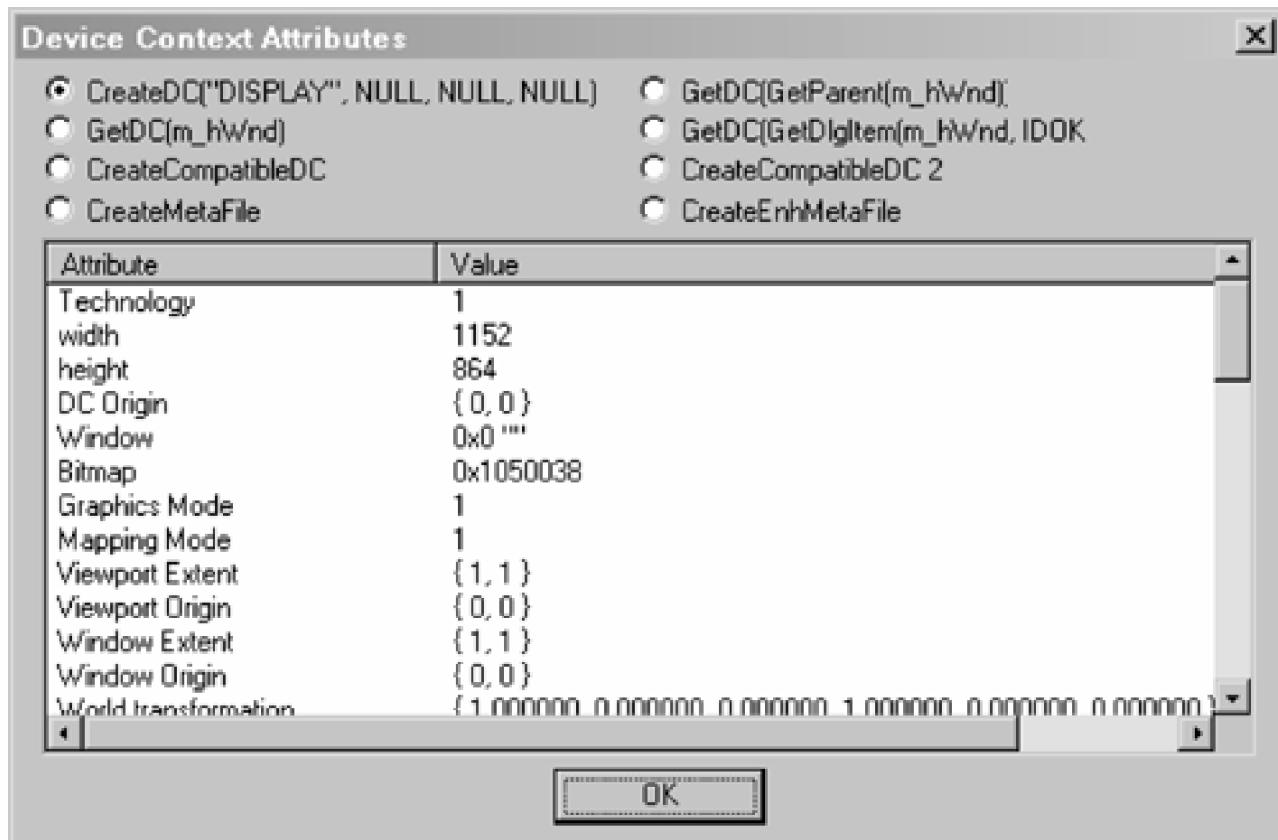
Layout		GetLayout, SetLayout
Path		BeginPath, ClosePath, EndPath, GetPath
Clipping region	Client area, whole device surface	SelectObject, GetClipBox, GetClipRgn, SelectClipRgn, ExcludeClipRect, IntersectClipRect
Meta region		GetMetaRgn, SetMetaRgn
Bounding rectangles		GetBoundsRect, SectBoundsRect

The details of these device context attributes will be explained in sections and chapters to follow. Here we're just getting a general impression of the amount of information a device context holds.

What's shown here are device context attributes supported by Windows 2000, which is a superset of almost all device context attributes supported by different Windows platforms. Most of the device context attributes are inherited from a 16-bit Windows API. Some attributes are fully supported only on Windows NT/2000—for example, world coordinate transformation. Windows 98 and Windows 2000 add quite a few new attributes, like DC pen, DC brush, and image color management (ICM) attributes.

In the DEVICE program the “DC Attributes” button on the “Display Devices” page pops up a dialog box that shows all accessible device context attributes in a list view, as in [Figure 5-7](#). The dialog box allows several ways of getting a device context handle. The code can create a new device context using CreateDC, or retrieve the device context associated with different windows.

Figure 5-7. Device context attributes.



Device Context Associated with a Window

A device context created by CreateDC call can be seen as a drawing surface that covers the whole device surface, which for screen display is the whole screen, and for the printer is the whole page. But CreateDC is not the normal way of creating a device context in the Microsoft Windows environment; it's usually reserved for creating a device context for a hard-copy device like a printer.

Displaying in a Multiple Window Environment

The main target for graphics display is the screen display, a resource shared by multiple applications under the Windows operating system. Normal Windows applications work in windowed mode, in which each confines its display to a region of the screen.

The region for a window is normally rectangular, with its parameters specified in the CreateWindow API call. By design, the operating system supports a window of any shape, a rounded rectangle, an ellipse, or a polygon. To change the shape of a window to a nonrectangular shape, just create a region object and call SetWindowRgn with its handle. The coordinates used in the region are in display device coordinates relative to the origin of the window.

Here is a simple example of creating a normal window and then changing its shape to be elliptic.

```
const TCHAR szProgram[] = _T("Window Region");
const TCHAR szRectWin[] = _T("Rectangular Window");
const TCHAR szEptcWin[] = _T("Elliptic Window");

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE, LPSTR, int)
{
    HWND hWnd = CreateWindow(_T("EDIT"), NULL,
                           WS_OVERLAPPEDWINDOW,
                           10, 10, 200, 100, GetDesktopWindow(),
                           NULL, hInstance, NULL);
    ShowWindow(hWnd, SW_SHOW);

    SetWindowText(hWnd, szRectWin);
    MessageBox(NULL, szRectWin, szProgram, MB_OK);

    HRGN hRgn = CreateEllipticRgn(0, 0, 200, 100);

    SetWindowRgn(hWnd, hRgn, TRUE);

    SetWindowText(hWnd, szEptcWin);
    MessageBox(NULL, szEptcWin, szProgram, MB_OK);

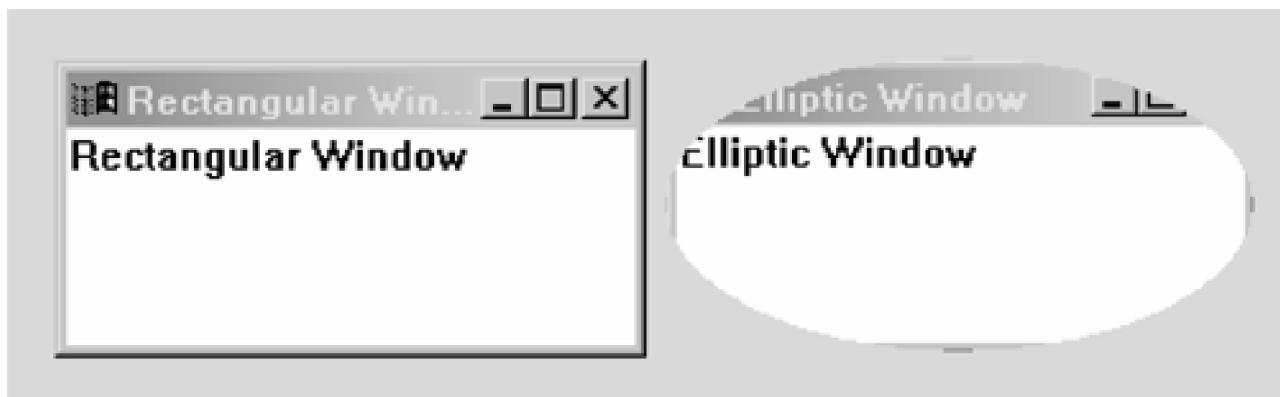
    DestroyWindow(hWnd);

    return 0;
}
```

[Figure 5-8](#) illustrates a normal rectangular window and an elliptic window. The rectangular window on the left has the

normal top-level window border and title bar nicely drawn, while the elliptic window has its border and title bar clipped to the elliptic region.

Figure 5-8. Different shapes of windows.



Note

Using nonrectangular windows and customizing nonclient areas is a new trend in Windows user interface design. These nontraditional windows use window regions defined by bitmaps or vector graphics, and paint the window using them. Nonclient area messages are handled to replace the default window handling for nonclient area.

Multiple windows on a display could overlap each other. In the old Microsoft Windows implementation, windows in the front completely or partially block windows in the back, although in the latest implementation in Windows 2000, Microsoft is moving toward treating windows as display objects which can be layered together using different operators. Overlapping gives each window another attribute, its visible region.

A window itself can be divided into two portions, a nonclient area and a client area. The nonclient area includes its border, title bar, menu bar, tool bar, scroll bars, and other fancy stuff. The client area of a window is what's left over after taking out its nonclient area, which is normally a rectangular region in the center. Drawing in the nonclient area is normally handled by the default window message handler, Def Window Proc, which is provided by the operating system's window management module (USER32.DLL). DefWindowProc has access to window style, window text, keyboard focus state, and other information it needs to draw the nonclient area. A user application's main concern is drawing in the client areas of its windows.

In a multitasking environment like Windows, screen display is a volatile canvas. At any time, an application can pop up a window, display some information, and then vanish into thin air. Applications that are still running are fully responsible to restore the damage done to their display on the screen. In this case, the operating system will send drawing request messages to all the windows affected. The windows receiving the messages do not have a sense of what's happened to their display, so they need to be told exactly what part of the screen needs updating, or else they can waste precious resources to repaint the whole window.

In summary, here are the factors a device context needs to take care of in order to draw in a window:

- Window origin: upper left corner of the window.

- Window dimension: width and height of the window.
- Window region: a subset of the rectangular area defined by its origin and dimension.
- Visibility: only the nonblocked part of the window region gets drawn.
- Whole window or only client area: Does the application want to customize non-client area drawing, or does it only care about the client area?
- Update region: the area that really needs updating.

Getting a Device Context Associated with a Window

The process of creating a device context associated with a window cannot be done by an application using CreateDC. Instead, Win32 API provides several functions to get a device context associated with a window. Here they are:

```
HDC GetWindowDC(HWND hWnd);
HDC GetDC(HWND hWnd);
HDC GetDCEx(HWND hWnd, HRGN hrgnClip, DWORD flags);
HDC BeginPaint(HWND hWnd, LPPAINTSTRUCT lpPaint);
```

GetWindowDC returns a device context that has been prepared to draw on the entire window, including its title bar, menu, and scroll bars. The origin for the device context will be the same as the window's origin. GetDC returns a device context that has been prepared to draw only on the client area of a window. The origin of the device context will be the origin of the client area. Neither GetWindowDC nor GetDC honors a window's WS_CLIPCHILDREN and WS_CLIPSIBLINGS style flags. In other words, you can use handles returned by either of them to paint over child windows or sibling windows.

After you're done with a device context, it must be released to the operating system. ReleaseDC must be used to release the resource associated with a device context handle from GetWindowDC, GetDC, or GetDCEx. For BeginPaint, the corresponding call EngPaint must be used.

A device context has several user-accessible parameters that reflect its association with a window. WindowFromDC returns the handle of a window that a device context is associated with. The origin of a device context returned by GetDCOrgEx, which is always {0,0} if created through CreateDC, is the screen coordinates of a window's origin, or its client area.

Besides these documented read-only attributes, a device context has several undocumented fields. A rectangle is used to keep the origin and dimensions of the associated window, or its client area. We call this rectangle display "[Rectangle](#)," although its official name seems to be "erclWindow," because it may refer only to the client area of a window. A region object, referred to as a visible region, is used to keep track of the visible portion of a window's region.

If the window is fully visible, the device context's visible region contains only a single rectangle that is the same as its display rectangle. If a corner of the window is blocked by another window, the visible region of the device context will be updated to contain two rectangles. For example, the visible region for the device context returned by GetWindowDC may become a region containing two rectangles: {10,10,210,62} and {10,62,92,110}. Note that the

visible-region attribute of a device context is set before GetWindowDC or GetDC returns, but it is still updated by the operating system as other windows are created and destroyed, and change focus, size, or position. This design ensures that a device context handle is associated with a window, and is kept for a long time, without worrying about changes in the window's attributes.

If a window has a nondefault window region set by SetWindowRgn—an ellipse, in the above example—a device context's visible region becomes the visible subset of the intersection of the window's region with the display rectangle. In other words, a device context's visible region only contains pixels that meet all three conditions—they are in the window's region set by SetWindowRgn, they are within its display rectangle, and they are visible. In our elliptic window sample, the visible region contains dozens of rectangles, or to be more precise, dozens of scans, which are used to represent the region more efficiently in Windows NT/2000.

The third method to get a window-associated device context is GetDCEEx. GetDCEEx accepts two more parameters compared with GetDC or GetWindowDC: a handle to a region object and a flag. Although the region is implied as a clipping region in Microsoft documentation, it's definitely different from the clipping region the application program can control using SelectClipRgn or ExtSelectClipRgn.

Table 5-5 summarizes different tweaks GetDCEEx allows. Normally, we would refer you to MSDN online documentation, but for GetDCEEx, MSDN Library omits two cases.

With so many flags, GetDCEEx can be used to replace other functions. For example, GetDCEEx(hWnd, NULL, DCX_WINDOW | DCX_NORESETATTRS) can easily replace GetWindowDC(hWnd), and GetDCEEx(hWnd, NULL, DCX_NO RESETATTRS) can replace GetDC(hWnd). With additional flags, you can force the system not to use a device context owned by a window or a class, and you can remove sibling and child windows from the visible region. You can also modify the visible region using an extra region parameter, modify the visible region with the window's update region, and even reset a window's update region.

Table 5-5. GetDCEx Functionality

Flags	Function
DCX_WINDOW	Return a device context with the window's rectangle, instead of the client area's rectangle.
DCX_CACHE	Use a device context cached by the window manager, even if the window's class has a CS_OWNDC or CS_CLASSDC style flag.
DCX_PARENTCLIP	Use the parent window's rectangle and visible region, ignoring the parent window's WS_CLIPCHILDREN and CS_PARENTDC style flags.
DCX_CLIPSIBLINGS	Remove all the sibling windows' regions from the visible region.
DCX_CLIPCHILDREN	Remove all the child windows' regions from the visible region.
DCX_NORESETATTRS	Do not reset the device context's attributes to the default values.
DCX_LOCKWINDOW-UPDATE	Ignore the lock set by the LockWindowUpdate. Used for allowing drawing during tracking.
DCX_EXCLUDERGN	Exclude the region identified by hrgnClip from the visible region.
DCX_INTERSECTRGN	Intersect the region identified by hrgnClip with the visible region to generate a new visible region.
DCX_EXCLUDE-UPDATE	Exclude the window's update region from the visible region.
DCX_INTERSECT-UPDATE	Intersect the window's update region with the visible region to generate a new visible region.
DCX_VALIDATE	Validate the window—or, in other words, reset its update region.
0x10000	An undocumented flag which can magically make Get DCEx call succeed.

This leads to the last method used to get a window-associated device context, BeginPaint, which is the official function to get a device context to handle a WM_PAINT message. Looking only at the device context returned, BeginPaint can be implemented as:

```
HDC BeginPaint0(HWND hWnd, LPPAINTSTRUCT lpPaint)
{
    DWORD flags =0;

    if ( GetWindowLong(hWnd, GWL_STYLE) & WS_CLIPCHILDREN )
        flags |= DCX_CLIPCHILDREN;

    if ( GetWindowLong(hWnd, GWL_STYLE) & WS_CLIPSIBLINGS )
        flags |= DCX_CLIPSIBLINGS;

    return GetDCEx(hWnd, NULL, flags | DCX_INTERSECTUPDATE |
        DCX_VALIDATE);
}
```

BeginPaint checks the window's style to see if child and sibling windows should be excluded from the combined visible region, which is then intersected with the window's update region. The window itself gets fully validated—that is, its update region resets to an empty region. The DCX_CACHE and DCX_NORESETATTRS flags are not used here, so GetDCEx needs to check the window class's style to determine how the situation should be handled. The

real BeginPaint implementation handles other tasks. If a caret is in the region to be painted, BeginPaint hides the caret to prevent it from being erased or taken as a background image to update. BeginPaint sends a WM_ERASEBKGND message to the window's message handler. If an application needs to handle it, it has a chance to paint a uniform background, or a more fancy bitmap background. If the system's default message handler DefWindowProc gets the message, and if the window class has a valid background brush, the brush is used to clear the region to be painted. BeginPaint also needs to fill a PAINTSTRUCT structure with the device context generated, a bounding box of the region to be painted, and some other flags.

By now we should be very clear about the differences between a device context associated with a window, and a device context created by CreateDC. Namely, the former has a display rectangle that is a subset of the whole device surface, a combined visible region which is generated considering factors like window's region, child and sibling window clipping, visible areas, and window's update region.

Common Device Context

Now there is still one more confusing thing to explain: Where does a device context returned by GetDC, GetWindowDC, GetDCEEx, or BeginPaint come from?

In the old days, the Windows operating system was running in real mode on machines with only 640 KB of memory. So a device context, using close to 200 bytes of memory, was considered a large structure. Creating a device context through loading a device driver, finding numerous entry points, and setting various attributes on a 20-MHz machine was considered slow. The window management module (USER) calls CreateDC five times to create a cache of five device contexts. GetDC, GetWindowDC, and BeginPaint literally fetch device contexts from this device context cache. An application is required to release the device context handle as soon as its drawing task is done, such that other windows get to use them if needed. In 16-bit Windows, running out of the five cached device contexts will cause the applications display to fail.

A normal device context retrieved from the cached device context pool is called a *common device context*.

The five-device-context limitation applies only to 16-bit Windows implementation. Windows 95, 98, NT, and 2000 do not have this limitation. If the system is running out of cached device contexts, a new device context will be created and used.

Windows NT/2000 is again different from Windows 95 and 98. Windows NT/2000 is based on a pure-32-bit protected subsystem design, where each process runs in its own address space. Although most of the GDI resource is shared systemwide, each GDI object handle is process bound, meaning that a device context created by one process can be used only by that process. Windows 95 and 98 are based on an improved 16-bit GDI implementation that moves big data structures like device contexts to a separate 2-MB heap, much roomier than the single 64-KB GDI heap used in 16-bit Windows.

Class Device Context

The CS_CLASSDC flag in WNDCLASS's style field tells the window management module to create a device context shareable by all windows of the particular class. Such a device context is called *class device context*. The actual device context is created when the first window instance of that class is created, and initialized once to its default values.

When GetDC, GetWindowDC, or BeginPaint is called on a window belonging to such a class, the device context attached with the window class is returned, with an updated display rectangle, visible region, and empty clipping

region. All other attributes in the class device context are kept the same—for example, logical pen, text color, mapping mode, etc. After finishing drawing, ReleaseDC or EndPaint returns the device context to the window class, without destroying it or resetting its attributes. A class device context is destroyed only when the last window of the class is destroyed.

If you have ever heard that ReleaseDC and EndPaint can be omitted for class device context because they do nothing, forget about it. This kind of suggestion should be considered harmful, because it can cause big trouble for a small gain. For one thing, EndPaint restores the caret turned off by BeginPaint.

Class device context is useful for control windows that are drawn using the same attribute values, because it minimizes the time required to prepare a device context for drawing and releasing afterward. Another benefit of class device context is its minimum memory usage.

Class device context is provided only for backward compatibility. Its advantage is diminished by today's larger RAM and faster CPU, and by the protected address space design of Win32 processes. Class device context is not recommended in Win32 programming.

Private Device Context

The CS_OWNDC flag in WNDCLASS's style field tells the window management module to create one device context for each window created using this class. So every window will have a dedicated device context during the lifetime of the window—that is, a *private device context*.

A private device context gets initialized once to its default values. Each call to GetDC, GetWindowDC, or BeginPaint retrieves a window's private device context, with a new display rectangle and visible region. The application can then set device context's attribute and issue drawing commands. ReleaseDC or EndPaint returns the device context to the window, without changing the device context, so the next time a device context is requested, attributes like pen and brush are still the same.

The MSDN documentation on private device context is not clear (check the Private Display Device Contexts portion). It mentions that the application must retrieve the handle only once, and then it goes on to say the application could call BeginPaint to incorporate the update region.

Private device context goes to an extreme to improve performance by sacrificing memory resource. A device context uses three types of resource: a GDI handle, memory in the user application's address space, and memory in system kernel address space. Private device context is useful only for windows with complicated settings that take a long time to prepare, and for windows that need frequent updating. It's recommended only when performance considerations are much more important than memory and GDI handle resource usage.

Parent Device Context

The CS_PARENTDC flag is not related to the problem that private device contexts and class device contexts are trying to solve. When this flag is on, GetDC or BeginPaint for a child window uses its parent window's display rectangle and visible region to prepare a device context; that's why it's called a *parent device context*.

A parent device context is allocated from the cached device context pool, so it is still initialized to the default device context attributes. The difference is that a parent device context inherits the parent window's display rectangle and visible region, to save the time and space needed to calculate the individual child window's display rectangle and visible region.

The CS_PARENTDC flag is honored only in the simple case when a child window wants to use the parent window's parameters to draw itself. It's ignored if the parent window uses private or class device context, or if the parent window clips its child windows, or if the child window clips its child window or sibling windows.

Other Device Contexts

So far we have only touched on device context created using CreateDC, and device context associated with a window. These two types of device contexts provide full-featured access to a real hardware device. They allow querying and sending drawing commands to either a video display card or a printer.

Device contexts have other variations under Windows environment: namely, information context, memory device context, and metafile device context.

Information Context

In certain situations, the only thing an application needs to do is query a few attributes of a graphics device. For example, when a word processor starts up with a document, it needs to query the default printer for paper size and margins to properly format the document in a WYSIWYG fashion. In this situation, Windows allows creating a limited device context called an information context using CreateIC:

```
HDC CreateIC(LPCTSTR pszDriver,  
             LPCTSTR pszDevice,  
             LPCTSTR pszOutput,  
             CONST DEVMODE * pdvmlInit);
```

CreateIC works the same way as CreateDC, except that the former is said to be more time and space efficient. Drawing calls through an information context handle returned by CreateIC simply get ignored. An information context is deleted by calling DeleteDC, not the nonexistent DeleteIC.

Memory Device Context

Device context is supposed to provide a device-independent way for an application to communicate with graphics devices. But the device contexts returned by the functions described above only allow graphics drawing on a physical graphics device, like video display cards and printers. These devices are supported by graphics device drivers, which receive low-level drawing calls from the graphics engine. Sometimes, it's very useful to be able to draw into a graphics device completely simulated in memory by bitmaps.

Memory device context in Win32 API is designed to allow drawing onto a bitmap associated with it, or drawing the bitmap onto another graphics device's surface. A memory device context is created by:

```
HDC CreateCompatibleDC(HDC hDC);
```

The hDC parameter represents an existing device context the memory device context will be "compatible" with. Memory device context needs a bitmap as the actual drawing surface; the default bitmap when it's first created is

always a single-pixel bitmap. Win32 provides functions to create a bitmap and attach it with a memory device context using SelectObject. DeleteDC handles the deletion of a memory device context.

A memory device context inherits a lot from its reference device context. In fact, GetDeviceCaps on a memory device context will return the same results as when it's applied to its reference device context. A memory device context claiming to have DT_RASPRINTER as its TECHNOLOGY attribute could be confusing to routines that accept both memory and nonmemory device contexts.

Memory device context is a very useful technique. We will cover it in detail in [Chapter 10](#), when we discuss the device-dependent bitmap (DDB) and DIB section.

Metafile Device Context

Another type of in-memory device context is the metafile device context. While memory device context allows using GDI drawing commands to form a bitmap, metafile device context allows recording the exact GDI commands into an in-memory data stream or a disk file, which can then be played back like a recorded music or video clip. The main difference between them is that a memory device context creates bitmaps at a fixed size and resolution to store the final result of drawing, while a metafile stores both vector and raster drawing commands, which can then be scaled to different sizes precisely.

Two functions are provided to create metafile device context, one for Win16 metafile generation, the other for Win32 enhanced metafile generation:

```
HDC CreateMetaFile(LPCTSTR lpszFile);
HDC CreateEnhMetaFile(HDC hdcRef, LPCTSTR lpFileName,
                      CONST RECT * lpRect,
                      LPCTSTR lpDescription);
```

Both Windows metafile (Win16 metafile) and enhanced metafile are widely used in commercial applications to store clipart. Enhanced metafile is also the key data structure in printer spooler implementation on Windows 95, 98, NT, and 2000. We will be devoting [Chapter 16](#) of this book to metafiles.

To summarize, device context is a nice concept employed in Windows API to provide a device-independent way of graphics output. In this section we have covered different classes of device contexts, how they are created, their attributes, and methods to query their attributes.

[Table 5-6](#) summarizes the various device contexts and their characteristics.

5.3 FORMALIZING DEVICE CONTEXT

We've had a long section in which we looked at various types of device contexts and their generic attributes. Now we need some condensation on the conceptual level, using some insights gained from looking at GDI's implementation on Windows 2000.

A device context is a data structure used by a Windows graphics system that serves two purposes. First, a device context provides a device-independent environment, which allows graphics drawing onto various graphics devices, both physical (like a video display card) and logical (like a metafile). Second, a device context provides a repository of settings, graphics objects, which can be reused by drawing commands.

The fundamental drawing surface supported by a device context is a two-dimensional array of pixels, which are individually addressable, readable, and writeable. This drawing surface model matches perfectly with raster display cards and raster-based printers, but it's not universal. Devices that do not meet this requirement will see certain graphics operations not supported. For example, printers using Postscript languages allow only drawing onto surface, not read from surface. Implementing binary or ternary operation on a Postscript printer would be difficult. As another example, a metafile device does not allow reading a pixel's color value.

Table 5-6. Device Contexts at a Glance

Device Context Type	Creator, Destructor	Usage
Generic device context	CreateDC, DeleteDC	Access to whole device surface, allow drawing to primary display, secondary display, mirroring device and hard-copy devices.
Device context associated with a window	GetWindowDC, GetDC, GetDCEx, BeginPaint, EndPaint, ReleaseDC	Allow drawing to part of the display surface which corresponds to a window's visible region, or its client area, with child and sibling windows' region removed. Device context returned by BeginPaint further restricts the drawing area to what intersects with the window's update region. It can be further classified as common, window, class, private, or parent device contexts.
Information context	CreateIC, DeleteDC	Allow queries on device and device driver capabilities.
Memory device context	CreateCompatibleDC, DeleteDC	Allow drawing onto an in-memory bitmap surface, drawing from it to another device context.
Metafile device context	CreateMetaFile, CreateEnhMetaFile, DeleteDC	Allow recording of GDI commands into a data stream or file, which can then be played back.

Normally, a device context allows drawing into every pixel on the drawing surface, although a printer will have a hard time printing pixels on the edge of a piece of a paper; that's why device context allows a user application to query for paper margins. To support graphics drawing in a multiple-windows, multitasking environment, a device context can be associated with a window. Under the control of a window management module, a sophisticated scheme is used to determine exactly where drawing is allowed. A device context uses the following attributes to determine a subset of the device surface to draw on:

- *Window rectangle:* defines a rectangular region of device surface to draw on. This corresponds to a window's bounding box as initially specified in a Create Window call. Any movement and resizing of the window is automatically tracked by the system and gets reflected in a device context's window rectangle. GetDCOrgEx returns the top, left corner of the window rectangle.
- *System region:* a region calculated considering several factors. System region starts with the window's region, which is normally a rectangle but could be any shape as defined by SetWindowRgn. Regions occupied by child or sibling windows are then removed from a device context's system region, if the corresponding flag is set in window class style. Any region that's not visible according to the z-order of all windows on the desktop is then removed. The system region is then intersected with the window's update region, or dirty region, if BeginPaint is used to retrieve the device context. A device context's system region automatically tracks any movement in the window, or any changes in the window's z-order.
- *Meta region and clipping region:* application-defined subsets of device surface where the application wants drawing to happen. The meta region and clipping region for a device context are always reset to the whole device surface when a device context is created or retrieved from the window management module. The meta region is not well-documented in Win32 API. It serves another layer of clipping. Numerous APIs are provided to change a device context's clipping region.
- *Rao region:* precalculated intersection of a system region, a meta region, and a clipping region. The system region, meta region, and clipping region are independent fields of a device context, although some documents imply a system region is the initial value of a clipping region when a device context is retrieved. But the actual drawing calls are allowed to affect only the intersection of the system region, meta region, and clipping region. To avoid recalculating the intersection for every drawing call, the graphics engine precalculates the intersection, updates it when either system region, meta region, or clipping region changes, and stores the result in the Rao region field. The field and related functions are named after a Microsoft engineer, Rao Remala, who insisted this field is needed in a device context, according to *Undocumented Windows*.

A device context is normally associated with a graphics device driver, which is fully responsible for sending commands to a graphics device. The interface protocol between the graphics engine and a graphics device driver is called Device Driver Interface (DDI), as documented in Microsoft Device Development Kit (DDK). A graphics device driver supplies the graphics engine with a table of callback functions, which implement necessary DDI interface calls. The graphics engine creates a logical device structure to hold related information about a graphics device driver.

A graphics device driver can be instantiated multiple times with different settings like a frame buffer pixel format, or various printing settings. This allows dynamic switching of display modes and spooling of multiple jobs at the same time. With each instantiation of a device driver, a DEVMODE structure is passed to the driver with a graphics engine or an application's request, and two structures are returned with device and driver's attributes and capabilities. The graphics engine stores the information in a physical device structure. The graphics driver also creates its own data structure and passes its handle to the graphics engine. A physical device structure holds lots of information about a device driver—for example, its dimensions, capability, limitations, hatch brush customization, halftoning patterns, etc.

A device context structure has pointers to the logical device structure and physical device structure. The physical device structure serves the user application's query for device capabilities through GetDeviceCaps. It also tells the graphics engine how GDI drawing calls should be broken down into DDI calls acceptable to the device driver. For example, is Bezier curve supported by the driver, or should it be broken down into line segments? The callback function pointers in a logical device structure direct the graphics engine to the right function handling individual DDI calls. So these two structures isolate device dependency in the whole Windows graphics system.

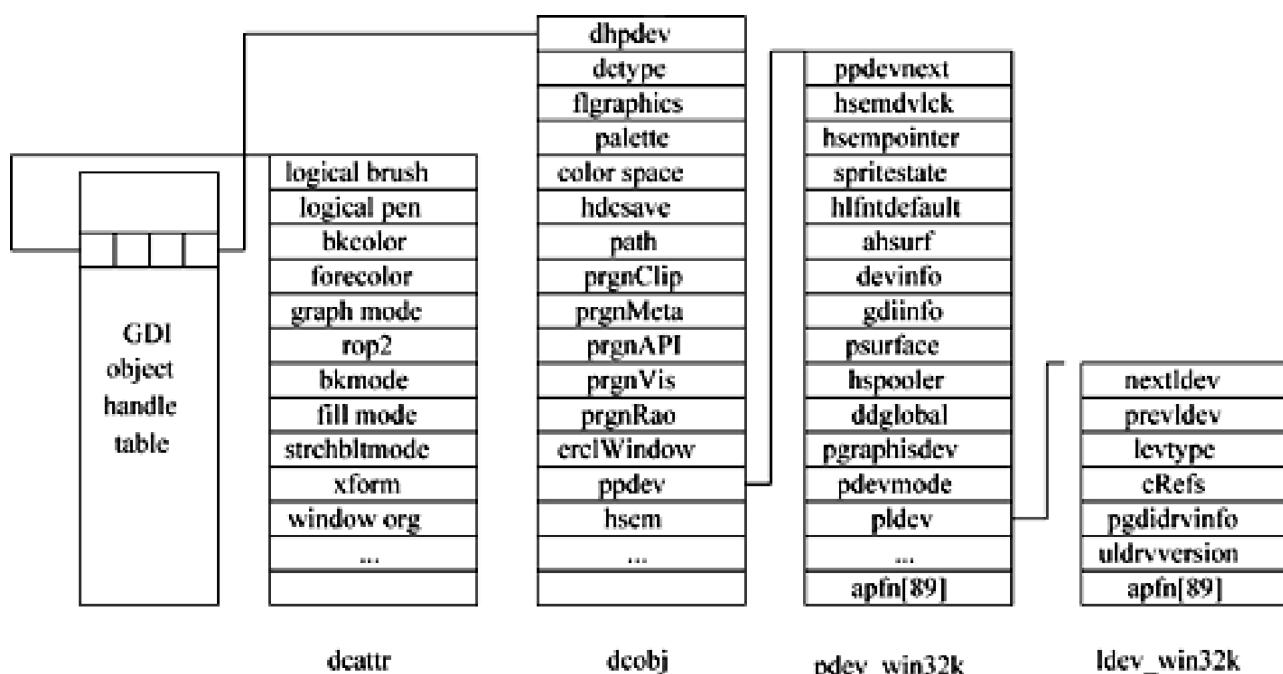
A device context also holds numerous other fields to support its association with a window, a bitmap, or a metafile,

and settings and objects used in Win32 API. Some of these fields are accessible using Win32 API, others are half or completely hidden from the user application. Some are read-only, like device context origin; others are both readable and writeable. For performance reasons, some commonly accessed device context attributes are stored in user address space, which can be efficiently accessed without switching to kernel mode and switching back. A big chunk of a device context is still kept in kernel mode address space, where the graphics engine and normal graphics device driver operate.

Like other GDI objects we are going to see in the next chapter, a device context is heavily guarded by GDI. When a device context is created, only its handle is returned to the user application, which serves as its reference when the user application calls GDI. GDI could map a device context handle to an entry in the GDI object table (discussed in the next chapter), which has pointers to both a device context's user mode and kernel mode structure.

[Figure 5-9](#) shows some details of a device context on Windows 2000.

Figure 5-9. Windows 2000 device context and related data structures.



For a device context, DCATTR is its user mode structure, which contains almost all the settings and objects you can set using Win32 API, except things like palette and color space. A DCOBJ is a device context's kernel mode structure, which holds its display rectangle, clip region, system region (named as prgnVis here), Rao region, and other regions. The PPDEV field is a pointer to the physical device structure, PDEV_WIN32K. The GDIINFO, DEVINFO, and AHSURF fields in a PDEV_WIN32K structure store information reported by the device driver. The PLDEV field points to the logical device structure, LDEV_WIN32K. The LDEV_WIN32K structure is mostly a function table of up to 89 driver callback function pointers, APFN, which is also duplicated in the PDEV_WIN32K structure.

A device context is very much like an object in an object-oriented system such as C++. The various attributes kept in a device context are the data members of such an object. The function table embedded deep in a device context is just like the virtual function table in a C++ object with virtual functions. Each graphics device driver provides a concrete implementation of the abstract device context by implementing DDI functions in the function table. Once a device context is properly set up, an application can use a common set of functions to draw into it, without worrying about how the drawing calls are implemented.

The information shown in [Figure 5-9](#) is by no means complete. Refer to [Chapter 3](#) for details, or explore on your own using either WinDbg with its GDI debugger extension, or the Fosterer program provided in [Chapter 3](#). [Chapter 4](#)

presents a program to overwrite the DDI function table in the PDEV_WIN32K structure, for spying on DDI interface.

[Chapter 2](#) presents a simple printer driver that implements the DDI interface.

[< BACK](#) [NEXT >](#)

5.4 SAMPLE PROGRAM: GENERIC FRAME WINDOW

Starting in this chapter, we're going to touch almost all GDI drawing primitives to explain how they work. The programs we've written so far normally use dialog boxes or property sheets as their main user interface, which will not be appropriate to demonstrate GDI API.

The purpose of this section is to develop a generic framework for writing Windows programs according to the following pattern:

- The main window of the program has a title bar, a menu bar, a system menu, and a border that can be dragged to resize the main window.
- The main window has a toolbar to link frequently accessed functions with intuitive bitmaps, with tooltip.
- The main window has a status window, which is divided into several panes, for status information display.
- The main window has a main drawing window (here called canvas window) in the rest of its client area to draw whatever the program wants to draw.

A program written in this generic framework is basically functionally equivalent to what the MFC application wizard will generate, if you choose single document, no document/view architecture, no database support, and no Active X controls.

To make the framework really reusable, several C++ classes with virtual functions are defined. Once again, all C++ classes in this book start with the letter "K", such that they can be used together with MFC classes, which normally start with the letter "C".

Toolbar Class

Here is the class for toolbar.

```
class KToolbar
{
    HWND    m_hToolTip;
    UINT    m_ControlID;
public:
    HWND    m_hWnd;

    KToolbar()
    {
        m_hWnd    = NULL;
        m_hToolTip = NULL;
        m_ControlID = 0;
    }

void Create(HWND hParent, HINSTANCE hInstance,
```

```
    UINT nControlID, const TBBUTTON * pButtons,
    int nCount);

void Resize(HWND hParent, int width, int height);
};
```

The KToolbar class is simple enough that we don't think it needs any virtual function. Its main method is the Create method, which accepts an array of TBBUTTON definitions, creates a toolbar child window with the buttons, and creates a tool tip window. Each tool in the tooltip window corresponds to the bitmaps on the toolbar. Each TBBUTTON definition has a resource string identifier stored in its dwData field, so the Create method can load the string and add to the tooltip window. The Resize method resizes the toolbar window to the new width of its parent window's client area.

Status Window Class

Status window class is also quite simple. Here is the KStatusWindow class declaration.

```
typedef enum
{
    pane_1,
    pane_2,
    pane_3
};

class KStatusWindow
{

public:
    HWND m_hWnd;
    UINT m_ControlID;
    KStatusWindow()
    {
        m_hWnd = NULL;
        m_ControlID = 0;
    }

    void Create(HWND hParent, UINT nControlID);
    void Resize(HWND hParent, int width, int height);
    void SetText(int pane, HINSTANCE hInst, int messid,
                int param=0);
    void SetText(int pane, LPCTSTR message);
};
```

The Create method creates a status window as a child window of its parent. The Resize window resizes the window's width to be the same as its parent window's client area width, and divides up the status window into three panes. The two SetText methods provide two ways an application can display messages in a status window.

Canvas Window Class

The canvas window is the main place where graphic drawing happens. The KCanvas class is derived from the KWindow class we discussed in [Chapter 1](#). The class has four virtual functions that can be overridden in derived classes.

```
class KStatusWindow;

class KCanvas : public KWindow
{
    virtual LRESULT WndProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam);
    virtual void OnDraw(HDC hDC);

    HINSTANCE m_hInst;

public:
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

    KStatusWindow * m_pStatus;

    KCanvas();

    void SetStatus(HINSTANCE hInst, KStatusWindow * pStatus)
    {
        m_hInst = hInst;
        m_pStatus = pStatus;
    }

    virtual ~KCanvas();
};
```

The WndProc virtual method handles all the messages directed to a canvas window. Its default implementation just handles WM_CREATE and WM_PAINT messages. For WM_PAINT messages processing, it calls BeginPaint, KCanvas::OnDraw, and then EndPaint. The OnCommand method handles WM_COMMAND message passed from the main window. Later in the book, we will develop a derived class from this class to handle zooming and scrolling commands.

Frame Window Class

The main window of a program is abstracted into a KFrame class, also derived from the KWindow class. In Windows terminology, the KFrame class supports SDI (single document interface) frame window, which will later be derived to support MDI (multiple document interface) frame window.

```
class KStatusWindow;
class KCanvas;
class KToolbar;

class KFrame : public KWindow
```

```
{  
typedef enum { ID_STATUSWINDOW = 101,  
    ID_TOOLBAR = 102  
};  
  
HINSTANCE      m_hInst;  
  
KToolbar      * m_pToolbar;  
KCanvas       * m_pCanvas;  
KStatusWindow * m_pStatus;  
  
const TBBUTTON * m_pButtons;  
int           m_nButtons;  
  
int           m_nToolbarHeight;  
int           m_nStatusHeight;  
  
virtual LRESULT WndProc(HWND hWnd, UINT uMsg,  
    WPARAM wParam, LPARAM lParam);  
virtual LRESULT OnCreate(void);  
virtual LRESULT OnSize(int width, int height);  
virtual LRESULT OnCommand(WPARAM wParam, LPARAM lParam);  
  
public:  
  
KFrame(HINSTANCE hInstance,  
    const TBBUTTON * pButtons, int nCount,  
    KToolbar * pToolbar,  
    KCanvas * pCanvas,  
    KStatusWindow * pStatus);  
  
virtual ~KFrame();  
};
```

The WndProc virtual method is for main window message handling. It calls OnCreate during WM_CREATE processing, OnSize during WM_SIZE processing, and OnCommand during WM_COMMAND processing. There is nothing much to paint in the main frame window itself, because its client area is fully covered by a toolbar window, a canvas window, and a status window.

The constructor KFrame::KFrame(...) is interesting. We would like the class to be generic and reusable. So the KToolbar, KCanvas, and KStatusWindow instances are created outside of the KFrame class. Only their pointers are passed to KFrame class's constructor, together with a definition for toolbar buttons. Note that you can derive a class from KCanvas class and pass its pointer as a KCanvas object pointer. The constructor's implementation is very simple: Just store the parameters for later use by OnCreate and other methods.

The OnCreate method is the only routine with some code here. It needs to call the right method to create toolbar, canvas, and status windows, in the right size and position.

LRESULT KFrame::OnCreate(void)

```
{  
    RECT rect;  
  
    // toolbar window is at the top of client area  
    if ( m_pToolbar )  
    {  
        m_pToolbar->Create(m_hWnd, m_hInst,  
            ID_TOOLBAR, m_pButtons, m_nButtons);  
        GetWindowRect(m_pToolbar->m_hWnd, & rect);  
        m_nToolbarHeight = rect.bottom—rect.top;  
    }  
    else  
        m_nToolbarHeight = 0;  
  
    // status window is at the bottom of client area  
    if ( m_pStatus )  
    {  
        m_pStatus->Create(m_hWnd, ID_STATUSWINDOW);  
        GetWindowRect(m_pStatus->m_hWnd, & rect);  
  
        m_nStatusHeight = rect.bottom—rect.top;  
    }  
    else  
        m_nStatusHeight = 0;  
  
    // create a canvas window above the status window  
    if ( m_pCanvas )  
    {  
        GetClientRect(m_hWnd, & rect);  
  
        m_pCanvas->SetStatus(m_hInst, m_pStatus);  
  
        m_pCanvas->CreateEx(0, _T("Canvas Class"), NULL,  
            WS_VISIBLE | WS_CHILD,  
            0, m_nToolbarHeight, rect.right,  
            rect.bottom—m_nToolbarHeight—m_nStatusHeight,  
            m_hWnd, NULL, m_hInst);  
    }  
  
    return 0;  
}
```

The code checks the pointers to all child-window objects, calling their window creation method only when a valid pointer is passed. This allows each child window to be optional, and the main program still functions properly. The window management in the OS makes sure the toolbar is at the top of the client area, and the status window is at the bottom of it. The CreateEx call on the canvas window takes this into consideration to adjust the canvas window's position and height.

The default KFrame::OnSize method implementation makes sure the three windows are still in the correct position

and size when the main window resizes. The OnCommand method is the initial place where menu commands will be sent. Its default implementation passes the message to KCanvas::OnCommand.

Test Program

The real test for our generic frame window classes is how easy and flexible programs using them could be. We only want to show interesting parts of programs, not to repeat similar code fragments over and over again.

Here is a simple test program that uses all four classes mentioned above. It has a title bar, a tool bar with two buttons and tooltip, a canvas window, and a status window. Compared with an MFC program, there are lots of things we don't have here: macros, global variables, heap allocations, and any system DLLs we don't use explicitly.

```
const TBBUTTON tbButtons[] =
{
    { STD_FILENEW,    IDM_FILE_NEW, TBSTATE_ENABLED,
        TBSTYLE_BUTTON, { 0, 0 }, IDS_FILENO, 0 },
    { STD_HELP,      IDM_APP_ABOUT, TBSTATE_ENABLED,
        TBSTYLE_BUTTON, { 0, 0 }, IDS_HELPABOUT, 0 }
};

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR lpCmd,
                    int nShow)
{
    KToolbar    toolbar;
    KCanvas     canvas;
    KStatusWindow status;

    KFrame frame(hInst, tbButtons, 2, & toolbar, & canvas,
                 & status);

    frame.CreateEx(0, _T("ClassName"), _T("Program Name"),
                  WS_OVERLAPPEDWINDOW,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  CW_USEDEFAULT, CW_USEDEFAULT,
                  NULL,
                  LoadMenu(hInst, MAKEINTRESOURCE(IDR_MAIN)),
                  hInst);

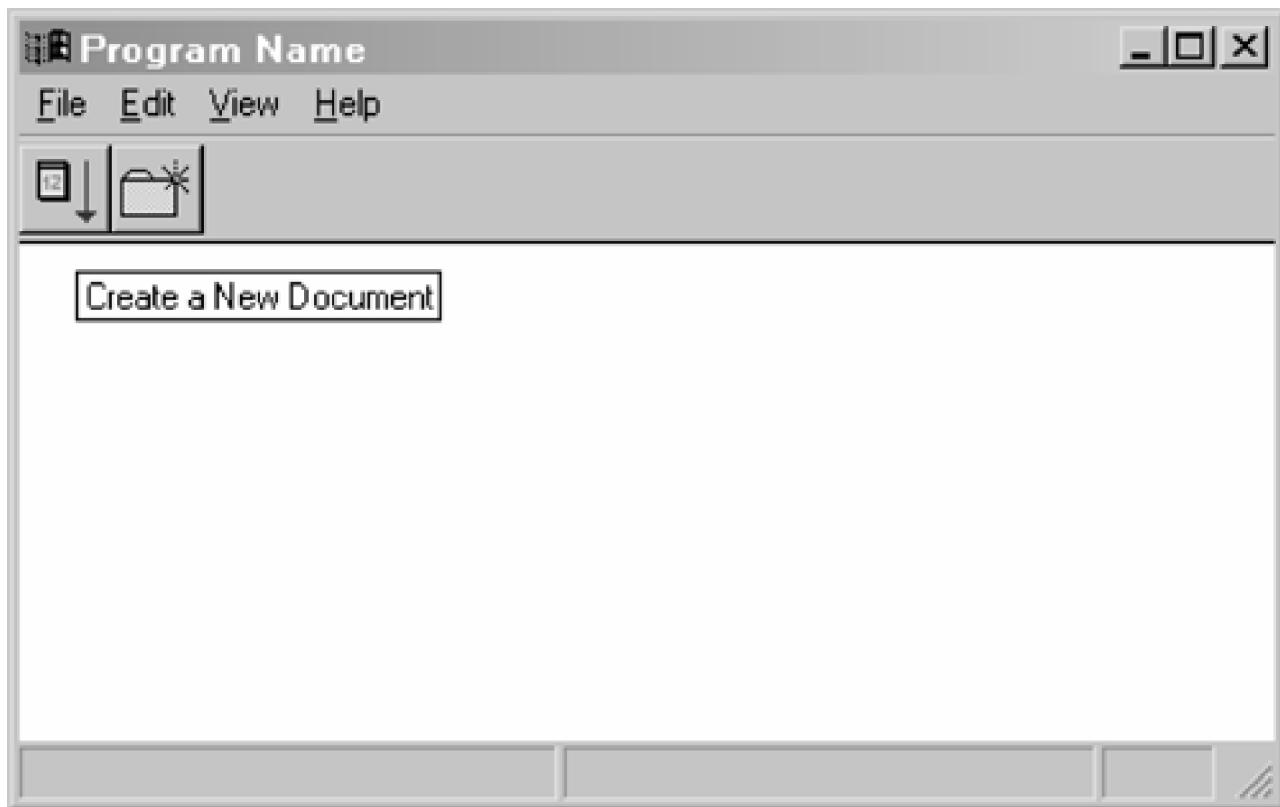
    frame.ShowWindow(nShow);
    frame.UpdateWindow();

    frame.MessageLoop();

    return 0;
}
```

[Figure 5-10](#) shows the screen display of our test program.

Figure 5-10. A simple program using the generic frame window classes.



If you think the TBBUTTON is used incorrectly here, you may still be using an incorrect document. Win32 TBBUTTON structure has seven fields. MSDN and other documents do not show the fifth field: **BYTE bReserved[2]**. The C/C++ compiler is forgiving until you start to use the last two fields, which the code uses to store tooltip string resource identifiers.

[< BACK](#) [NEXT >](#)

5.5 SAMPLE PROGRAM: PAINTING AND DEVICE CONTEXT

Graphics drawing in the Windows environment happens in an event-driven manner. An application is always supposed to be able to reproduce its entire screen display, because the display is shared by multiple windows belonging to different applications. When drawing is needed in a window, the WM_PAINT message is sent to its window procedure. The WM_PAINT message is the key to most painting happening in Windows programs, yet it's very hard to conceptualize. The window manager automatically generates a WM_PAINT message when a window is visible, there are no other more urgent messages to process, and there is an update region to repaint.

A Window's Update Region

A window's visible region is determined by the window's bounding rectangle, its region set by SetWindowRgn, and its relationship with other windows on the desktop. The WM_PAINT message is not actually put into a thread's message queue and handled equally with other messages. Instead, when painting is needed, a bit is set which triggers the window scheduler to call window's message procedure when no other messages are available in the message queue. Another way to force painting in a window is to call UpdateWindow directly.

A window's update region starts with an empty region, and continues to update when the following functions are called.

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

```
BOOL InvalidateRgn(HWND hWnd, HRGN hRgn, BOOL bErase);  
BOOL ValidateRgn(HWND hWnd, HRGN hRgn);
```

InvalidateRect/InvalidateRgn adds a rectangle or region to a window's update region. If NULL is given, the whole client area of a window is added. ValidateRect/ValidateRgn does the opposite: It removes a rectangle or a region from a window's update region. If NULL is given, the whole client area is removed. The bErase parameter tells window manager whether the background erasing message WM_ERASEBKGND should be generated when BeginPaint is called.

A window's update region could also change when the window resizes or scrolls, or when a window on top of it is removed, moved, or resized such that a certain area needs to be repainted.

When a window resizes, the WM_SIZE message is generated; the window manager checks the CS_HREDRAW and CS_VREDRAW flag in window class's style flag (WNDCLASSEEx.style), not window's style flag. If CS_HREDRAW or CS_VRE DRAW flag is on, when a window's width or height changes, its whole client area is invalidated; otherwise, only the newly added window area is invalidated. Any window size change forces window to repaint immediately.

When a user drags a window's border to resize the window, normally the window manager simulates the window size change without actually changing the window's size, until the mouse button is released. In the newer Windows operating systems like Windows 95, 98, and 2000, the control panel display applet keeps a flag to overwrite this

behavior. Under the “Effects” tab, if the “Show window contents while dragging” check box is checked, the resizing message is generated multiple times during dragging. This could cause a serious user interface responsiveness problem if window repainting is slow.

Scrolling a window or its associated device context also causes a window to repaint. When a window or its client area is scrolled, pixels move up or down, left or right, and certain areas become unpainted. Those areas are also added to the window's update region.

Two functions are provided to query a window's current update region:

```
int GetUpdateRgn(HWND hWnd, HRGN hRgn, BOOL bErase);
BOOL GetUpdateRect(HWND hWnd, LPRECT lpRect, BOOL bErase);
```

GetUpdateRgn returns a window's update region through preexisting region handle hRgn; that is, hRgn must be a valid region object handle before the call, and after the call it will contain the window's update region.

GetUpdateRect just returns the bounding box of a window's update region. The last parameter bErase is for controlling whether a WM_ERASEBKGND message should be sent if the update region is not empty.

WM_PAINT Message

When the WM_PAINT message reaches a window's message procedure, an application normally calls BeginPaint. BeginPaint retrieves a device context, initializing its system region with the intersection of the window's visible region and update region. The update region is also validated (reset to empty region) before BeginPaint returns, so another cycle of update region collection can begin.

Besides returning an HDC, BeginPaint also fills a PAINTSTRUCT structure as follows:

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

The hdc field holds the same HDC returned by BeginPaint; it may be used by EndPaint to release the device context.

If the bErase flag is TRUE, the application should erase the background of the window, because all attempts to erase the background have failed. If background erasing is needed, as controlled by the bErase parameter in InvalidateRect and InvalidateRgn, the BeginPaint implementation sends a WM_ERASEBKGND to the window procedure, which should either handle the message or pass it to DefWindowProc. The default window procedure uses the background brush handle specified in the WNDCLASSEX.hbrBackground field to paint the background. But if no valid brush is specified, the background erasing is considered unfinished business to be handled by application.

The rcPaint rectangle is the bounding box of the device context's current system region; that is, the region needs repainting.

There are several ways the WM_PAINT message can be handled after BeginPaint is called. If you have more than a dummy program, you should consider optimizing the handling of WM_PAINT.

- In the simplest implementation, a window procedure draws everything it wants to display in the window, and lets GDI figure out how the drawing should be clipped. If the drawing involves sophisticated calculation and a large amount of drawing calls, performance will be a serious problem.
- An adequate implementation should check the rcPaint rectangle and repaint only those objects that intersect with it. Performance could be improved significantly when only a small region needs to be repainted, especially when the window contents is shown during dragging of the window's border.
- A more sophisticated implementation could use the system region directly. The rcPaint is a bounding box of the system region, which may not always be rectangular in shape. The system region could be much smaller than the area covered by the rcPaint rectangle. Querying the system region directly allows finer control of painting performance.
- If painting takes a long time, an application should consider using incremental window updating techniques. For example, a web browser needs an indefinite amount of time to download a large bitmap. A browser's WM_PAINT handling needs to display what's available on the local machine quickly, return control, and then update the window when new data is available. During that time, the user can scroll the window to read the information that's displayed, or even quit the viewing.

A device context's system region has been hidden from programmers for a long time. The newer versions of Windows header files document a function GetRandomRgn that allows you to access it. Although the function has long been exported from GDI32.DLL, it used to be undocumented.

```
int GetRandomRgn(HDC hDC, HRGN hrgn, INT iNum);
```

The only documented value for iNum is SYSRGN, although you can pass undocumented indexes to it to access other regions associated with a DC (which will be covered in [Chapter 7](#)). GetRandomRgn(hDC, hRgn, SYSRGN) copies a device context's system region data to the region data behind hRgn, which must be a valid region handle before the call. Once the region is retrieved, it can be decomposed to individual rectangles using GetRegionData. If this is too confusing, just take it as it is and wait until we explain the full details in [Chapter 6](#).

Before returning from WM_PAINT handling, the window procedure must call EndPaint, which releases resources associated with a device context if needed, or returns the device context to the common device context pool.

Visualize Window Painting Messages

In a normal implementation of WM_PAINT, the update region is repainted to form a new seamless picture with what's already on the screen.

But as a developer, we want to visualize the WM_PAINT messages—that is, to see when the messages are generated, what area is covered by the system region, and whether the device context handle is reused or is newly

created every time. We would also like to see how other related messages like WM_NCCALCSIZE, WM_NCPAINT, WM_ERASEKBNG, and WM_SIZE are generated and handled.

[Listing 5-1](#) shows a program WinPaint, which is designed to help us understand the WM_PAINT message. It is based on the generic window framework built in [Section 5.4](#).

Listing 5-1 WinPaint Program: Visualize WM_PAINT Message

```
// WinPaint.cpp
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <assert.h>
#include <tchar.h>

#include "..\..\include\win.h"
#include "..\..\include\Canvas.h"
#include "..\..\include\Status.h"
#include "..\..\include\FrameWnd.h"
#include "..\..\include\LogWindow.h"
#include "Resource.h"

class KMyCanvas : public KCanvas
{
    virtual void OnDraw(HDC hDC, const RECT * rcPaint);
    virtual LRESULT WndProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam);

    int      m_nRepaint;
    int      m_Red, m_Green, m_Blue;
    HRGN    m_hRegion;
    KLogWindow m_Log;
    DWORD   m_Redraw;

public:
    BOOL OnCommand(WPARAM wParam, LPARAM lParam);

    KMyCanvas()
    {
        m_nRepaint = 0;
        m_hRegion = CreateRectRgn(0, 0, 1, 1);

        m_Red   = 0x4F;
        m_Green = 0x8F;
        m_Blue  = 0xCF;
        m_Redraw = 0;
```

}

};

LRESULT KMyCanvas::OnCommand(WPARAM wParam, LPARAM lParam)

{

switch (LOWORD(wParam))

{

case IDM_VIEW_HREDRAW:

case IDM_VIEW_VREDRAW:

{

HMENU hMenu = GetMenu(GetParent(m_hWnd));

MENUTEMEINFO mii;

memset(& mii, 0, sizeof(mii));

mii.cbSize = sizeof(mii);

mii.fMask = MIIM_STATE;

if (GetMenuState(hMenu, LOWORD(wParam),

MF_BYCOMMAND) & MF_CHECKED)

mii.fState = MF_UNCHECKED;

else

mii.fState = MF_CHECKED;

SetMenuItemInfo(hMenu,LOWORD(wParam),FALSE,& mii);

if (LOWORD(wParam)==IDM_VIEW_HREDRAW)

m_Redraw ^= WVR_HREDRAW;

else

m_Redraw ^= WVR_VREDRAW;

}

return TRUE;

case IDM_FILE_EXIT:

DestroyWindow(GetParent(m_hWnd));

return TRUE;

}

return FALSE; // not processed

}

LRESULT KMyCanvas::WndProc(HWND hWnd, UINT uMsg,

WPARAM wParam, LPARAM lParam)

{

LRESULT lr;

switch(uMsg)

{

case WM_CREATE:

m_hWnd = hWnd;

m_Log.Create(m_hInst, "WinPaint");

m_Log.Log("WM_CREATE\r\n");

```
return 0;

case WM_NCCALCSIZE:
    m_Log.Log("WM_NCCALCSIZE\r\n");
    lr = DefWindowProc(hWnd, uMsg, wParam, lParam);
    m_Log.Log("WM_NCCALCSIZE returns %x\r\n", lr);

    if ( wParam )
    {
        lr &= ~ (WVR_HREDRAW | WVR_VREDRAW);
        lr |= m_Redraw;
    }
    break;

case WM_NCPAINT:
    m_Log.Log("WM_NCPAINT HRGN %0x\r\n", (HRGN) wParam);
    lr = DefWindowProc(hWnd, uMsg, wParam, lParam);
    m_Log.Log("WN_NCPAINT returns\r\n");
    break;

case WM_ERASEBKGND:
    m_Log.Log("WM_ERASEBKGND HDC %0x\r\n", (HDC) wParam);
    lr = DefWindowProc(hWnd, uMsg, wParam, lParam);
    m_Log.Log("WM_ERASEBKGND returns\r\n");
    break;

case WM_SIZE:
    m_Log.Log("WM_SIZE type %d, width %d, height %d\r\n",
              wParam, LOWORD(lParam), HIWORD(lParam));
    lr = DefWindowProc(hWnd, uMsg, wParam, lParam);
    m_Log.Log("WM_SIZE returns\r\n");
    break;

case WM_PAINT:
{
    PAINTSTRUCT ps;

    m_Log.Log("WM_PAINT\r\n");
    m_Log.Log("BeginPaint\r\n");
    HDC hDC = BeginPaint(m_hWnd, &ps);
    m_Log.Log("BeginPaint returns HDC %8x\r\n", hDC);

    OnDraw(hDC, &ps.rcPaint);

    m_Log.Log("EndPaint\r\n");
    EndPaint(m_hWnd, &ps);
    m_Log.Log("EndPaint returns "
              "GetObjectType(%08x)=%d\r\n",
              hDC, GetObjectType(hDC));
    m_Log.Log("WM_PAINT returns\r\n");
}
```

```
    }

    return 0;

default:
    lr = DefWindowProc(hWnd, uMsg, wParam, lParam);
}
return lr;
}

void KMyCanvas::OnDraw(HDC hDC, const RECT * rcPaint)
{
    RECT rect;

    GetClientRect(m_hWnd, &rect);
    GetRandomRgn(hDC, m_hRegion, SYSRGN);

    POINT Origin;
    GetDCOrgEx(hDC, &Origin);

    if ( ((unsigned) hDC) & 0xFFFF0000 )
        OffsetRgn(m_hRegion,—Origin.x,—Origin.y);

    m_nRepaint++;

    TCHAR mess[64];

    wsprintf(mess, _T("HDC 0x%X, Org(%d, %d)"), hDC,
             Origin.x, Origin.y);
    if ( m_pStatus )
        m_pStatus->SetText(pane_1, mess);

    switch ( m_nRepaint % 3 )
    {
        case 0: m_Red = (m_Red + 0x31) & 0xFF; break;
        case 1: m_Green= (m_Green + 0x31) & 0xFF; break;
        case 2: m_Blue = (m_Blue + 0x31) & 0xFF; break;
    }

    SetTextAlign(hDC, TA_TOP | TA_CENTER);

    int size = GetRegionData(m_hRegion, 0, NULL);
    int rectcount = 0;

    if ( size )
    {
        RGNDATA * pRegion = (RGNDATA *) new char[size];
        GetRegionData(m_hRegion, size, pRegion);

        const RECT * pRect = (const RECT *) &pRegion->Buffer;
        rectcount = pRegion->rdh.nCount;
```

```
TEXTMETRIC tm;
GetTextMetrics(hDC, & tm);
int lineHeight = tm.tmHeight + tm.tmExternalLeading;
for (unsigned i=0; i<pRegion->rdh.nCount; i++)
{
    int x = (pRect[i].left + pRect[i].right)/2;
    int y = (pRect[i].top + pRect[i].bottom)/2;

    wsprintf(mess, "WM_PAINT %d, rect %d",
        m_nRepaint, i+1);
    ::TextOut(hDC, x, y-lineHeight, mess,
        _tcslen(mess));

    wsprintf(mess, "(%d, %d, %d, %d)", pRect[i].left,
        pRect[i].top, pRect[i].right, pRect[i].bottom);
    ::TextOut(hDC, x, y, mess, _tcslen(mess));

}

delete [] (char *) pRegion;
}

wsprintf(mess, _T("WM_PAINT message %d, %d rects in sysrgn"),
    m_nRepaint, rectcount);
if ( m_pStatus )
    m_pStatus->SetText(pane_2, mess);

HBRUSH hBrush = CreateSolidBrush(RGB(m_Red, m_Green, m_Blue));
FrameRgn(hDC, m_hRegion, hBrush, 4, 4);
FrameRgn(hDC, m_hRegion, (HBRUSH)
    GetStockObject(WHITE_BRUSH), 1, 1);

DeleteObject(hBrush);
}

int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, LPSTR, int nShow)
{
    KMyCanvas canvas;
    KStatusWindow status;

    KFrame frame(hInst, NULL, 0, NULL, & canvas, & status);

    frame.CreateEx(0, _T("ClassName"), _T("WinPaint"),
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT,
        CW_USEDEFAULT, CW_USEDEFAULT,
        NULL, LoadMenu(hInst, MAKEINTRESOURCE(IDR_MAIN)), hInst);
    frame.ShowWindow(nShow);
```

```
frame.UpdateWindow();  
  
frame.MessageLoop();  
  
return 0;  
}
```

I guess we have some serious explaining to do. The long include-list is a good sign that we are reusing classes developed before. The code uses the KWindow, KCanvas, KToolbar, KFrame, and a new KLogWindow class. The KLogWindow manages a popup multiline "EDIT" window where logging information is kept. Implementation for these and other reusable classes are grouped to form a library and linked into programs using them.

The KMyCanvas class is derived from the KCanvas class. The message procedure, painting and command handlers are overridden. The new command handler On Command handles two menu commands, toggling vertical and horizontal redraw flags. We mentioned that there are two flags, CS_HREDRAW and CS_VREDRAW, in WNDCLASSEX, which determine whether the whole client area should be redrawn when the window resizes. KMyCanvas::OnCommand lets you toggle an internal flag kept in m_Redraw, which will be honored by our WM_NCCALCSIZE handling.

The new window message handler handles several messages related to window painting and drawing; they are WM_NCCALCSIZE, WM_NCPAINT, WM_ERASE BKBNG, WM_SIZE, and finally WM_PAINT. The code here calls the default window procedure DefWindowProc, except for WM_PAINT, which goes to the OnDraw method. But instead of simply passing the calls, the code logs data before and after processing the message. For WM_PAINT handling, log data are sent before and after BeginPaint and EndPaint.

The WM_NCCALCSIZE message gives a window a chance to calculate its client area size. Its processing does have a useful function. When the wParam is TRUE, the window procedure should return WVR_HREDRAW and/or WVR_VREDRAW flags, if resizing means redrawing the whole client area. So basically, this message really links the CS_VREDRAW and CS_HREDRAW flags to the window manager. The code here patches results returned by DefWindowProc to honor user choice through toggles on the main menu. You can observe the effects of those flags without recompiling this test program.

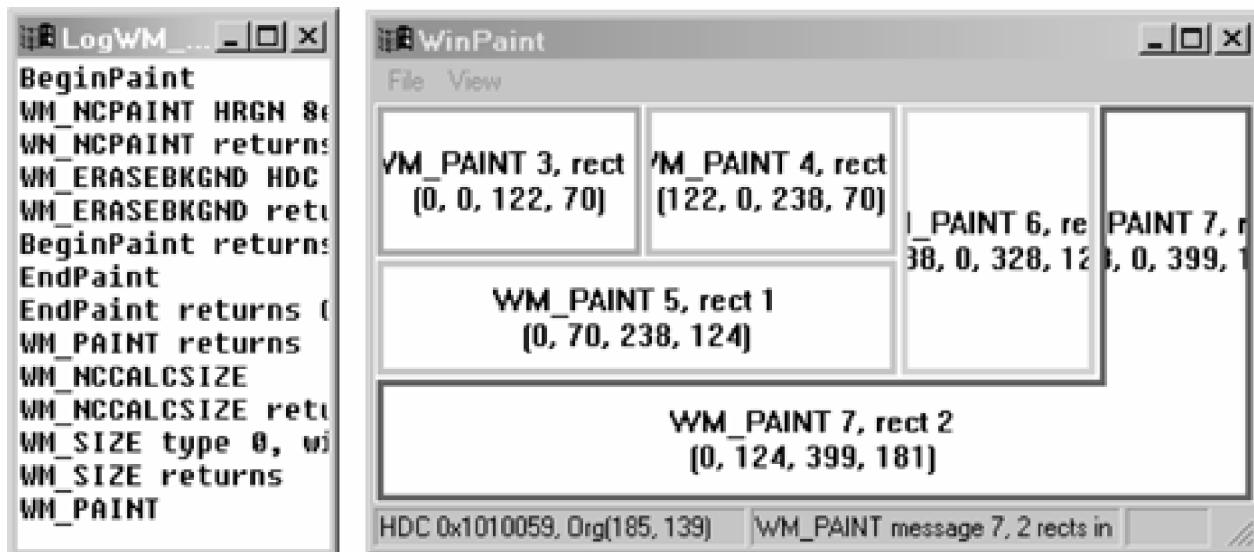
The KMyCanvas::OnDraw implementation is designed to visualize a WM_PAINT message. The routine starts by querying client area dimensions, a device context's system region, and a device context origin. The system region has two different interpretations. Under Windows NT/2000, the system region is in the screen coordinate, or physical device coordinate, space; under Windows 95/98, the system region is in the client area coordinate, or device coordinate, space. The code checks if it's running under Windows NT/2000 and, if so, adjusts the coordinate to the client area coordinate using OffsetRgn. Because we know that only Windows NT/2000 uses 32-bit GDI handles, the code tests the high word of HDC as a simple operating-system check.

The code then shows a device context handle and an origin in the first pane of the status window and calculates what color should be used to paint the system region. The color selection changes one of the red, green, and blue components for each WM_PAINT message processing. Now we're ready to decode the system region, which may be an empty region, a single-rectangle region, or made up of hundreds of rectangles.

The code calls GetRegionData twice, first to get the size of region data and second to retrieve the actual data. Again, just take it as it is; we will provide more details in [Chapter 6](#). For each rectangle within the region, the code displays its sequential number and coordinates in the center of the rectangle. After the rectangles are processed, the code displays a sequential WM_PAINT message number and the number of rectangles in the second pane of the status window. Finally, the system region is framed using a one-pixel-wide white border and a three-pixel-wide color border.

Now you can run the program, play with it, and learn how WM_PAINT messages are generated and how big an area they cover. [Figure 5-11](#) shows a sample run of the program, when the window is resized along both axes alternatively.

Figure 5-11. Sequence of WM_PAINT messages when a window resizes.



The first WM_PAINT message repaints a default-size window. We then resize the window to a smaller window, which generates the second WM_PAINT message with no rectangles in its system region. The window then gets minimized and restored, generating the third message to paint the small yet complete client area, shown as the first rectangle in [Figure 5-11](#). Resizing the window in a single direction each time it generates a WM_PAINT message with a single-rectangle system region. But resizing the window in both directions generates a WM_PAINT message with a dual-rectangle system region, shown as rectangle 1 and 2 for WM_PAINT number 7. If you open a menu and then close it, no new WM_PAINT message is generated, because the menu display saves and restores background images. But if you move a window over it, or you move part of the window off screen and bring it back, you will surely see painting messages. If you check the CS_HREDRAW and CS_VREDRAW flags under the view menu, resizing the window repaints the whole client area, instead of just new territory. If you enable "Show window contents during dragging" in the control panel display applet, dragging a window border generates frequent painting messages.

The log window on the left also generates interesting results. Here is the log for a single window resizing, indented for better reading.

```
WM_NCCALCSIZE  
WM_NCCALCSIZE returns 0  
WM_SIZE type 0, width 581, height 206  
WM_SIZE returns  
WM_PAINT  
BeginPaint  
WM_NCPAINT Hrgn 9e040469  
WN_NCPAINT returns  
WM_ERASEBKGND HDC 3b0105ae  
WM_ERASEBKGND returns
```

BeginPaint returns HDC 3b0105ae

EndPaint

EndPaint returns GetObjectType(3b0105ae)=0

WM_PAINT returns

After a user finishes dragging a window's border to find its new dimensions, a WM_NCCALCSIZE message is generated, which is followed by WM_SIZE and a WM_PAINT message. During WM_PAINT processing, BeginPaint generates a WM_NCPAINT message to repaint nonclient areas, and a WM_ERASEBKGND message to erase the background. The device context handle returned by BeginPaint is also passed to a WM_ERASEBKGND message. One interesting thing is that on Windows NT/2000, after EndPaint returns, the device context handle returned by BeginPaint is invalid (GetObjectType returns 0), yet after a few rounds, the same HDC appears again. This seems to prove that the graphics engine does keep a global cache of a few device context handles.

[< BACK](#) [NEXT >](#)

5.6 SUMMARY

This chapter has discussed the centerpiece of windows graphics programming, the device context. We covered a major class of graphics devices, including the video display card, to give us a concrete example of a graphics device. Methods to enumerate display devices and different display modes and to query their capabilities were discussed. Different types of device contexts and various ways to create a device context, especially to create a device contexts associated with a window, have been discussed. We also looked at how drawing in a window is managed using the window's update region. C++ classes have been built to write SDI programs to demonstrate Windows graphics programming concepts around the WM_PAINT message.

What's covered in this chapter is merely the background of device contexts and their connection with window management. The use of a device context for graphics drawing will be covered topic by topic in subsequent chapters.

Further Reading

Using a device context to encapsulate graphics attributes and abstract graphics devices is not unique to Windows GDI API. XWindow System uses a similar construct called the graphics context. *Xlib Programming Manual*, by Adrian Nye, is a good reference on the Xlib part of XWindow Systems.

If you're interested in the internal data structure of a device context, refer to [Chapter 3](#) of this book, where a detailed description is given. [Chapter 3](#) also shows how a device context is linked with graphics device drivers with a function table, which implements the DDI interface. [Chapter 4](#) of this book provides a way to monitor the Windows NT/2000 DDI interface.

Sample Programs

Compared with [Chapters 3](#) and [4](#), sample programs in this chapter are plain normal Windows programs. Yet, these programs are still trying to demonstrate what's normally not so apparent about device contexts and their relationship with drawing in a window environment (see [Table 5-7](#)).

Table 5-7. Sample Programs for Chapter 5

Directory	Description
Samples\Chapt_05\Device	Program to list various display devices, different display modes, device capabilities, and device context attributes.
Samples\Chapt_05\Ellipse	Program to illustrate rectangle and nonrectangle windows.
Samples\Chapt_05\FrameWindow	Sample program to test our SDI frame window-related classes.
Samples\Chapt_05\WinPaint	Program to visualize window paint messages, the window's system region, and CS_VREDRAW, CS_HREDRAW flags.

[< BACK](#) [NEXT >](#)

Chapter 6. Coordinate Spaces and Transformation

An application keeps a data structure with all objects it wants to represent. For example, a desktop publishing application may store paragraphs of text, images, line art, page heading, page footer, etc. A garden design application stores objects representing plants, fences, sidewalks, decks, lawns, etc. This kind of data structure is called a *model*.

What we are interested in is a subset of the model, the *geometric model*, which describes objects' dimensions, location, shape, texture, color, and other properties. The dimensions and locations of objects are normally given in physical units, inches or meters, as appropriate to the domain. The shapes of objects can be described in terms of primitives, such as rectangles, circles, polygons, etc.

Different applications may choose to model their world using different coordinate systems. Desktop publishing applications traditionally model their page layout using the point as the basic unit, roughly 1/72 of an inch. A computer-aided design package may choose 0.1 mm as a basic measuring unit, because it may be the smallest measurable unit supported by machines.

When a user tries to use an application to design a page layout, or an embroidery logo motif, or a backyard design, the application needs to provide ways to let the user change the display ratio and the position of the objects being displayed. It will be very inefficient for an application to modify all the location and size information it stores before such a request can be met. More often than not, objects exhibit a certain similarity, such that only one of them needs to be described in every detail and the remainder can be constructed through reflection, rotation, translation, or a combination of them. For example, a piece of clip art that represents a rose can be repeated multiple times to form a rose garden.

For practical reasons, then, the Win32 GDI supports several layers of coordinate systems, which are customizable by setting the transformation matrix and other attributes in a device context.

A coordinate space, as we use it in GDI, is actually a two-dimensional Cartesian coordinate system, with two axes that provide a means of specifying the location of each point on a plane. The Windows NT/2000 implementation of the Win32 API supports four layers of coordinate space:

- *World coordinate space*: supports *affine* transformation to page coordinate space. World space coordinates are limited to 2^{32} units high by 2^{32} units wide, because the Win32 API uses a 32-bit value to represent coordinates.
- *Page coordinate space*: supports limited transformation to device coordinate space. Page coordinate space is limited to 2^{32} units high by 2^{32} units wide.
- *Device coordinate space*: addresses individual pixels with a device context; supports

mapping into a rectangular region within a physical device coordinate space. Device coordinate space is limited to 2^{27} units high by 2^{27} units wide, because the WinNT/2000 graphics engine uses signed fixed-point numbers with a 4-bit fraction value internally.

- *Physical device coordinate space:* coordinate space formed by every pixel on the physical device drawing surface. A physical device coordinate space is limited to 2^{27} units high by 2^{27} units wide.

Note

Implementation of coordinate spaces and transformations is a totally different picture for Windows 95/98. The Windows 95/98 GDI relies heavily on the 16-bit GDI implementation inherited from Windows 3.1, which does not allow world transformation and truncates all coordinates to 16-bit values. To be more specific, on Windows 95/98, although coordinates passed to 32-bit GDI calls are 32-bit, they are truncated to 16-bit when GDI32.DLL thunked down to 16-bit GDI implementation.

Let's examine those coordinate spaces from the bottom up—from the physical device coordinate space to the world coordinate space.

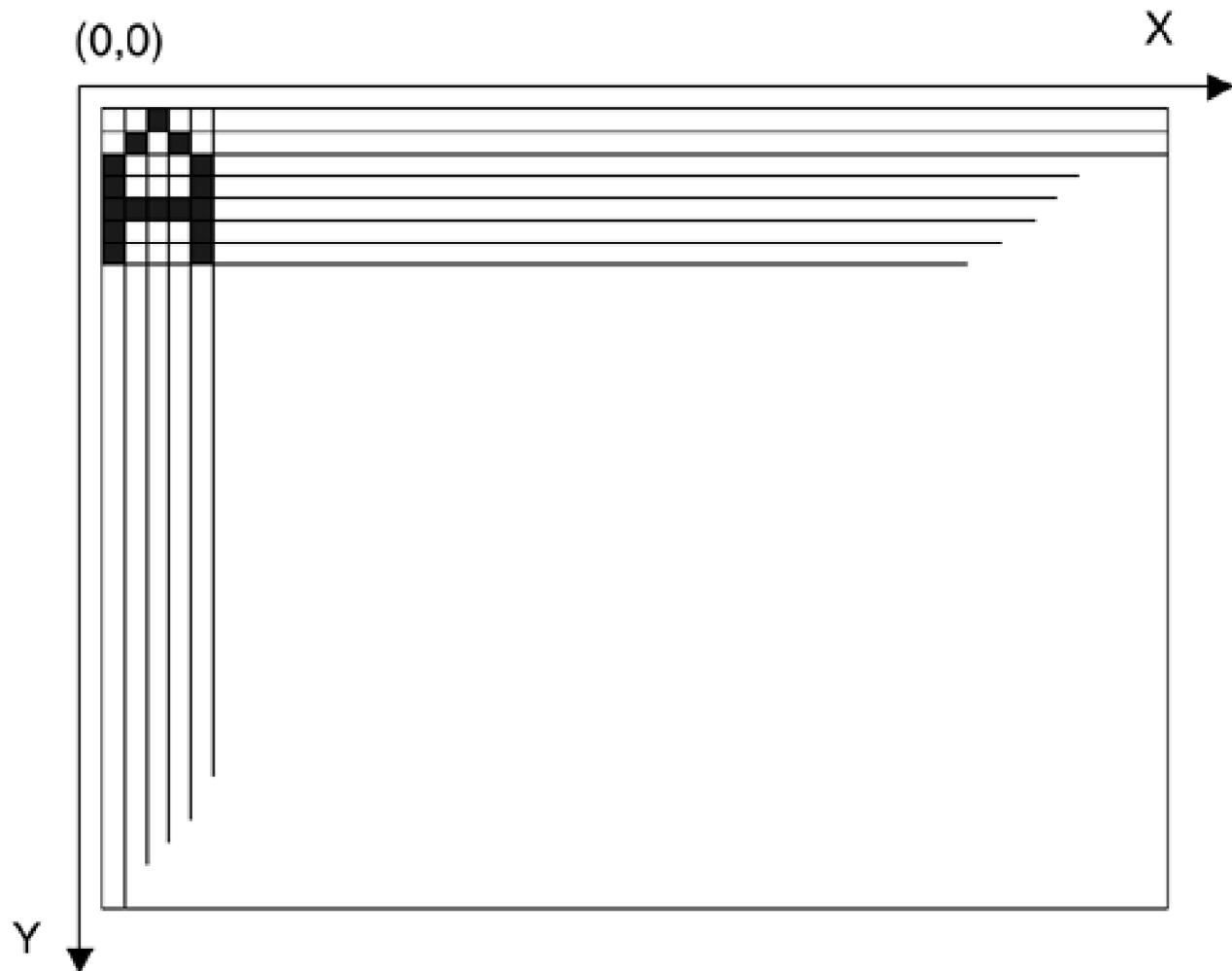
[< BACK](#) [NEXT >](#)

6.1 THE PHYSICAL DEVICE COORDINATE SPACE

The physical device coordinate space is the coordinate space used by the graphics device driver, which is formed by an array of pixels of fixed width and height. The coordinate for the top-left-most pixel is (0,0). The x-axis goes from left to right; the y-axis goes from top to bottom.

The Windows NT/2000 graphics engine uses signed fixed-point numbers to represent coordinates, having a 4-bit fraction part and a 28-bit signed integer part. Points with negative coordinates or coordinates beyond the height or width of the device surface are considered clipped. So the maximum physical device size is 2^{27} by 2^{27} , or 0.88 mile by 0.88 mile in 2400 dots-per-inch (dpi) resolution. [Figure 6-1](#) illustrates the physical device coordinate space.

Figure 6-1. Physical device coordinate space.



Physical device coordinates are used in the DDI interface between the graphics engine and graphics device drivers. Note the so-called physical device coordinate space may not be the final coordinate space that pixels are going to lie on. It's only final as far as the Windows graphics system is concerned. A device driver or the graphics device has the freedom to add more transformation to the drawing primitives. For example, a PostScript driver translates physical device coordinates it receives into floating-point numbers in points (1/72 inch). The generated Postscript data can

then be printed on different printers with various resolutions.

Strangely enough, there is no easy way to get a physical device's dimension if you have a device context handle. GetDeviceCaps(hDC, HORZRES) and GetDeviceCaps(hDC, VERTRES) normally work fine, but for memory and metafile device contexts, they return the dimensions of their reference device context. For memory device context, physical drawing surface is the bitmap selected into it. You can use GetObjectType to make sure it's OBJ_MEMDC, then use GetObject to retrieve a copy of BITMAP structure or DIBSECTION structure for the bitmap selected in the memory device context, which holds the dimensions of the selected bitmap. A metafile device context is unbound during its generation. Its dimensions may grow as drawing primitives are recorded. The header for an enhanced metafile (ENHMETAHEADER structure) holds its dimension information both in device units and physical dimension.

Not every pixel on a physical device surface is drawable. Hard-copy devices like printers have mechanical limitations on putting dots near the edges of a page. The application needs to query the printable area of a page using GetDeviceCaps.

For screen display, physical device coordinates are also called *screen coordinates*, which are used quite often in window management. For example, GetWindowRect returns a window's bounding rectangle in screen coordinates; coordinates in window messages like WM_NCMOUSEMOVE are also in screen coordinates.

[< BACK](#) [NEXT >](#)

6.2 THE DEVICE COORDINATE SPACE

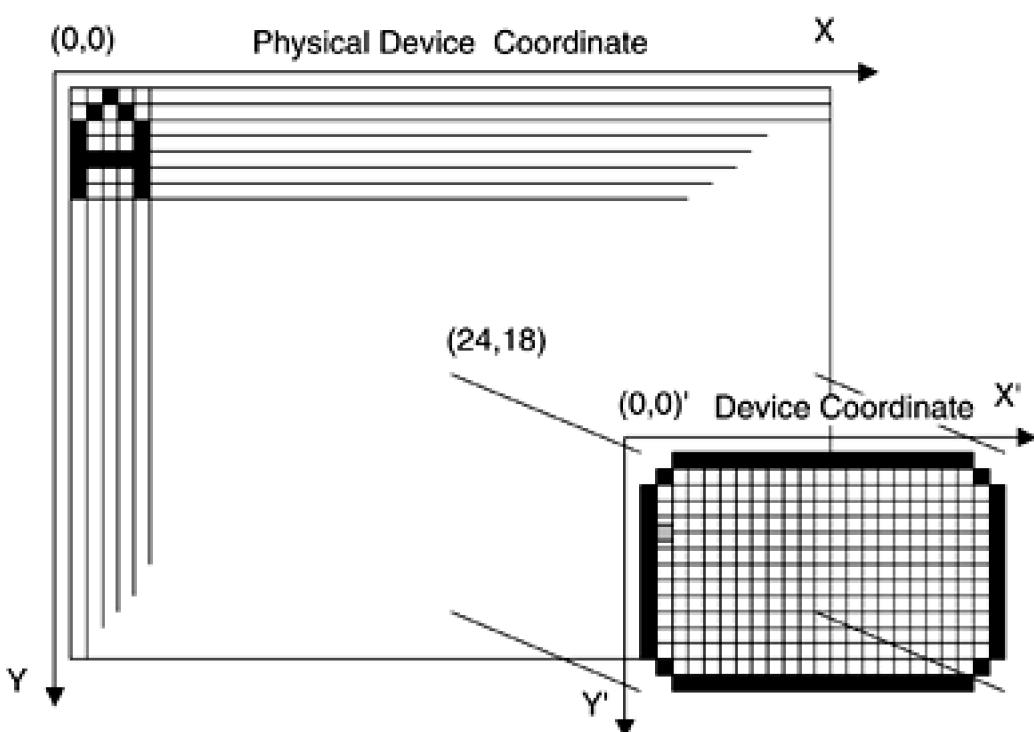
The device coordinate space is used by device contexts in the Win32 GDI API. In this general form, a device coordinate space is a subspace of its corresponding physical device coordinate space.

For a device context created using CreateDC, CreateIC, and CreateCompatibleDC, a device coordinate space and a physical device coordinate space are identical. For a device context associated with a window—that is, return values from GetDC, GetWindowDC, GetDCEx, or BeginPaint—a device coordinate space is defined by the window's rectangle or its client area.

Just like a physical device coordinate space, a device coordinate space's top-left corner has coordinate (0, 0); the x-axis extends from left to right; the y-axis extends from top to bottom. Given a device context handle, the relative position of a device coordinate space within its physical device coordinate space can be retrieved by GetDCOrgEx. To find out the dimensions of a device coordinate space, you need to know whether it corresponds to the client area of a window or the whole window. Then you can use either GetWindowRect or GetClientRect on the window handle returned by WindowFromDC.

[Figure 6-2](#) illustrates a device coordinate space and its mapping to a physical device coordinate space. The physical device coordinate space is the big grid in the background; the device coordinate space is a small rectangle in the front whose top-left corner is mapped to (24, 18) on the physical device coordinate space.

Figure 6-2. Device coordinate vs. physical device coordinate.



Two factors restrict the area an application can write to in a device coordinate space: system region and meta/ clipping regions. As we discussed in [Chapter 5](#), a device context's system region is the intersection of a window's visible region and its update region managed by the system window management module, while meta/ clipping regions are managed by the application. The intersection of a device context's system region and its meta/ clipping regions determines the pixels an application can access through a device context.

The origin, dimensions, and system region of a device context are automatically updated by the system; GDI has no control over them. So the mapping from a device coordinate space to its physical device space is a simple translation (shift of origin). It normally is not an application program's concern when you're dealing with a single-device context.

For interactive graphics where mouse messages are used to define, move, or edit graphics objects, or when a hit-test is needed, coordinates need to be translated between a physical device coordinate space (screen coordinates) and a device coordinate space (window coordinates or client area coordinates). The Win32 API provides several functions to do that.

```
BOOL ClientToScreen(HWND hWnd, LPPOINT lpPoint);
BOOL ScreenToClient(HWND hWnd, LPPOINT lpPoint);
int MapWindowPoints(HWND hWndFrom, HWND hWndTo, LPPOINT lpPoints,
    UINT cPoints);
```

`ClientToScreen` translates a point (POINT) in client area coordinates to screen coordinates; `ScreenToClient` does the opposite. `MapWindowPoints` translates an array of points from coordinates relative to one window to coordinates relative to another window.

Device coordinates are widely used in the Win32 API. In the GDI domain, a clipping region is expressed in device coordinates instead of page or world coordinates. This could cause lots of confusion and problems. We will cover details about clipping in [Chapter 7](#). In the window management domain, the counterpart to device coordinates is client area coordinates. They are used in location parameters to CreateWindow and SetWindowPos and in mouse message positions like WM_MOUSEMOVE and WM_LBUTTONDOWNDBLCLK.

[< BACK](#) [NEXT >](#)

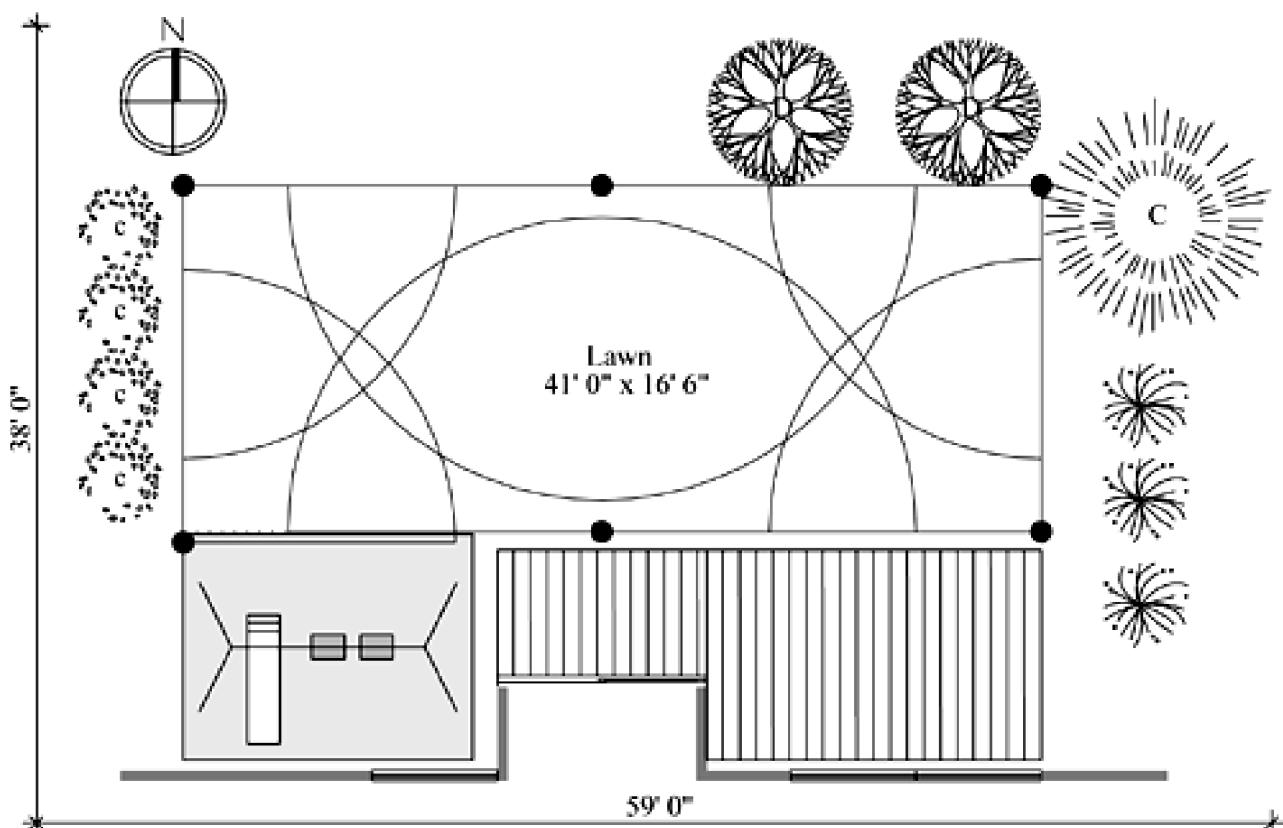
6.3 THE PAGE COORDINATE SPACE AND MAPPING MODES

Both physical device coordinate space and device coordinate space are confined to a device-dependent fixed array of pixels. The size of a window on a high-resolution display is normally different from what's on a low-resolution display; the thickness of a 3-pixel-wide line is different on printers with different resolution. To allow graphics programming to be more device independent, the Windows GDI allows an application to set up logical coordinate spaces that are closer to its geometric model, easier to use, and much more device independent.

The page coordinate space is one of the two logical coordinate spaces supported by the Win32 GDI. Actually, it's the only logical coordinate space supported in 16-bit Windows OSs, and even in Windows 95/98, WinCE implementations of the Win32 API. Only Windows NT/2000 supports the second logical coordinate space: world coordinate space. For this historical reason, Windows documentation, and even function names, normally refer to the page coordinate space as the logical coordinate space.

The page coordinate space allows an application to build its geometry model using unbounded 32-bit integer coordinates, arbitrary direction of axes, and arbitrary physical units on axes. For example, a landscape design software may use ? inch as its basic measure unit (or centimeter for the metric countries), origin in the bottom-left corner of a lot, x-axis extending from left to right, and y-axis extending from bottom to top [Figure 6-3](#) shows the logical coordinate space used in a landscape design. If your lawn is 41 feet by 16 feet 6 inches, its dimensions will be stored as (328, 132), as illustrated in [Figure 6-3](#).

Figure 6-3. Logical coordinate space used in a landscape design.



There are three questions as to how to produce a view of a geometric model on a graphics device: How much of the model should be drawn, where should it be displayed on the device surface, and how big should it be on it? This allows displaying a different portion of a model, at various scales, and at selectable locations on the surface.

To map page coordinate space to device coordinate space, the Win32 API uses two concepts: window and viewport, which are commonly used computer graphics. A window is any rectangle area on the page coordinate space—for example, the area covered by lawn in [Figure 6-3](#). A viewport is a rectangular area on the device coordinate space. So a window determines how much of the geometric model should be shown, the location of viewport determines where it should be drawn in the device surface, and the dimension ratio between the two determines how coordinates should be scaled.

To be more precise, a window is defined by four variables in page coordinates:

WOrgx	Window origin, x coordinate
WOrgy	Window origin, y coordinate
WExtx	Window horizontal extent
WExty	Window vertical extent

A view window is defined by four variables in device coordinates:

VOrgx	Viewport origin, x coordinate
VOrgy	Viewport origin, y coordinate
VExtx	Viewport horizontal extent
VExty	Viewport vertical extent

A point (x, y) in a page coordinate space can be mapped to a point (x', y') in a device coordinate space using the following formulas:

$$\begin{aligned}x' &= (x - WOrgx) * VExtx / WExtx + VOrgx \\y' &= (y - WOrgy) * VExty / WExty + VOrgy\end{aligned}$$

The formulas simply find the difference between point (x, y) and window origin, scale it to the viewport space using extent ratio, and then add the viewport origin. Converting a point in a device coordinate space back to a page coordinate space is equally simple.

$$\begin{aligned}x &= (x' - VOrgx) * WExtx / VExtx + WOrgx \\y &= (y' - VOrgy) * WExty / VExty + WOrgy\end{aligned}$$

Now let's take a look at what kind of mappings are allowed between these two coordinate spaces:

- **Identity.** Set window and viewport both to $(0, 0, 1, 1)$; we have $x' = x$ and $y' = y$. So page coordinate space is identical to device coordinate space.
- **Translation.** Set window to $(0, 0, 1, 1)$ and viewport to $(dx, dy, 1, 1)$; we have $x' = x + dx$ and $y' = y + dy$. Every

point in page space is moved by (dx, dy) when mapped into device space.

- **Scale.** Set window to $(0, 0, 1, 1)$ and viewport to $(0, 0, mx, my)$; we have $x' = x * mx$ and $y' = y * my$. Every point in page space is scaled by (mx, my) when mapped into a device space. The scale can be any fraction number, both upscale and downscale, and can be independent in x, y directions.
- **Reflection.** Set window $(0, 0, width, height)$ and viewport to $(width, height, -width, -height)$; we have $x' = width - x, y' = height - y$. An image in the page coordinate space can be flipped both horizontally and vertically when mapped to the device coordinate space. Reflection allows a page coordinate space to set up its axes differently from the fixed axes directions in a device coordinate space.
- **Combination.** Any combination of the above mappings is allowed.

The Win32 API provides the following functions to set up a page coordinate space through defining a window and a viewport:

```
BOOL SetWindowOrgEx(HDC hDC, int X, int Y, LPPOINT pPoint);
BOOL SetWindowExtEx(HDC hDC, int X, int Y, LPSIZE pSize);
BOOL SetViewportOrgEx(HDC hDC, int X, int Y, LPPOINT pPoint);
BOOL SetViewportExtEx(HDC hDC, int X, int Y, LPSIZE pSize);
```

The last parameter in these four functions is a pointer to a POINT or SIZE structure, which will be filled in with the device context's original setting if provided. Corresponding functions are provided to query a device context's window and viewport setup. Check for GetWindowOrgEx, GetWindowExtEx, GetViewportOrgEx, and GetViewportExtEx in Win32 document.

It does take some math to figure it out, doesn't it? To follow the fast-food trend, the Win32 API provides quite a few semiprefabricated page coordinate spaces, called *mapping modes*. A mapping mode normally has a preset window extent and viewport extent, which determine the unit of a page coordinate space and the scaling to a device coordinate space. But an application is free to change the window origin and viewport origin to allow displaying different portions of its geometric model on different portions of a screen. The mapping mode of a device context can be changed by calling:

```
int SetMapMode(HDC hDC, int fnMapMode);
```

MM_TEXT Mapping Mode

The simplest mapping mode is named MM_TEXT, set by `SetMapMode(hDC, MM_TEXT)`, which is the default mapping mode a fresh device context would be in. In the MM_TEXT mapping mode, both window extent and viewport extent are fixed at $(1, 1)$; window origin and viewport origin have the default value $(0, 0)$. So the default page coordinate space in a device context is the same as its device coordinate space.

An application can change window origin and viewport origin in MM_TEXT mode, so the general formulas for mapping a page coordinate to a device coordinate would be:

$$x' = x - WOrgx + VOrgx$$

$$y' = y - WOrgy + VOrgy$$

The axes setup in MM_TEXT mode is handy for text display in the English writing system, which goes from left to right and top to bottom. That could be why it's called MM_TEXT mapping mode in the first place. Simple graphics applications may also use it for screen display. If you want to use MM_TEXT mapping mode for printing, you need to handle lots of coordinate calculation and scaling yourself, to make sure a documentation can be printed in the same size on printers with different resolutions. MM_TEXT mapping mode does not allow changing the display ratio because the ratio between window and viewport is fixed.

MM_LOENGLISH, MM_HIENGLISH Mapping Modes

The physical units used in MM_LOENGLISH and MM_HIENGLISH mapping modes are based on the inch, the traditional English measure unit. In MM_LOENGLISH mode, one unit in a page coordinate space is 1/100 of an inch, while in MM_HIENGLISH mode, one unit is 1/1000 of an inch. In MM_LOENGLISH mode, one inch is 100 units, half an inch is 50 units, a quarter of an inch is 25 units, and an eighth of an inch can't be represented without loss of precision. In MM_HIENGLISH mode, one inch is 1000 units, half an inch is 500 units, a quarter of an inch is 250 units, and an eighth of an inch is 125 units.

The direction of the y axis is changed in both these two modes to follow the traditional Cartesian coordinate system; that is, the y-axis goes bottom up, as shown in [Figure 6-3](#). This is different from the MM_TEXT mapping mode, the device coordinate space, and the physical coordinate space.

For an application, setting up MM_LOENGLISH or MM_HIENGLISH mapping mode is easy; just call SetMapMode(hdc, MM_LOENGLISH) or SetMapMode(hdc, MM_HIENGLISH). GDI will change both window and viewport extent accordingly. This is a rough implementation of SetMapMode's handling of those two modes:

```
BOOL SetMapMode(HDC hDC, int fnMapMode)
{
    // change DC's mapmode to fnMapMode
    int mul;
    int div;

    switch ( fnMapMode )
    {
        case MM_HIENGLISH: mul = 10000; div = 254; break;
        case MM_LOENGLISH: mul = 1000; div = 254; break;
        ...
        default: return FALSE;
    }

    SetWindowExtEx(hDC,
        GetDeviceCaps(hDC, HORZSIZE) * mul / div ,
        GetDeviceCaps(hDC, VERTSIZE) * mul / div
        NULL);

    SetViewportExtEx(hDC,
        GetDeviceCaps(hDC, HORZRES),
        - GetDeviceCaps(hDC, VERTRES), NULL);
```

...

```
    return TRUE;  
}
```

Setting up English mapping modes in a device context uses a physical device's pixel and physical dimension. Note that HORZRES actually means the width of a physical device surface.

For the English mapping modes, a viewport's width is a device surface's width, and a viewport's height is the negation of a device surface's height. So the x-axis is the same as a device coordinate space, while the y-axis is reflected. Window width is calculated by multiplying device surface physical size by the number of units per 1/10 inch, and then dividing the result by 254. Note that a device's physical size is in millimeters (an inch is approximately 25.4 millimeters).

For an 1152-by-864-pixel screen display, the reported physical display size is 320 mm by 240 mm. So SetMapMode(hdc, MM_HIENGLISH) will set window extent to be (12,598, 9449) and viewport extent to be (1152, -864).

You may wonder why the logical resolution, as returned by GetDeviceCaps(hdc, LOGPIXELSX) and GetDeviceCaps(hdc, LOGPIXELSY), is not used instead. If logical resolution were used, a page coordinate space would be set up differently. For example, for the same 1152-by-864-pixel display mode, the display driver claims logical resolution of (96, 96). Device surface's physical size should be considered as 12 inches by 9 inches, so window extent should be (12,000, 9,600) in MM_HIENGLISH mode.

The word "logical" implies that the values need not be accurate; it's only a nominal number. A display driver normally supports different screen frame buffer sizes, and a display card can be linked to monitors of different sizes. People normally prefer that a document to be displayed clear enough and big enough for them to read and edit on screen. They don't really care if a word processor displays a letter-size page exactly as 8.5 inches on their monitor. A display driver in normal resolution reports itself as 96 dpi in logical resolution, or 120 dpi for high-resolution modes. Printing devices need to be really accurate, because people expect to see 6 lines of text in a document having 1/6-inch line spacing, and they expect output from an accounting package to fit exactly into pre printed forms. For hard-copy devices, logical resolution is the actual resolution.

On Windows NT/2000, GDI is relying more on the real size of a graphics device reported by a display device driver to set up mapping mode. A display driver queries the video port driver for video information, including physical screen size. It's possible for the display driver to report an accurate size, if it can get information from a monitor. But up to now, we've not seen a display driver reporting anything other than (320, 240) mm, the dimensions of a 17-inch monitor.

If you're using any mapping mode relying on the physical size of a device, you should not use logical resolution in your application to avoid possible inconsistencies.

Another thing to notice is the SetMapMode does not change window and viewport origin. The original value is kept intact, and an application is free to change it.

In terms of resolution, MM_HIENGLISH is 1000 dpi and MM_LOENGLISH is 100 dpi.

MM_LOMETRIC and MM_HIMETRIC Mapping Modes

The metric mapping modes MM_LOMETRIC and MM_HIMETRIC are similar to the English mapping modes. MM_LOMETRIC uses 0.1 millimeter as its basic unit, and MM_HIMETRIC uses 0.01 millimeter. Just as in the English mapping modes, positive x is to the right, and positivey is up.

To add support for metric mapping modes to the code shown above, just add two lines of code:

```
case MM_HIMETRIC: mul = 100; div = 1; break;  
case MM_LOMETRIC: mul = 10; div = 1; break;
```

In terms of resolution, MM_HIMETRIC is 2540 dpi and MM_LOMETRIC is 254 dpi.

MM_TWIPS Mapping Mode

The metric mapping modes serve countries using the metric system, and the English mapping modes serve countries that are still using the English system, but the printing industry needs another measuring system. Traditionally, the printing industry uses points to measure the size of variable types. A point is roughly 0.013835 inch, or 1/72.228 of an inch. Computer typesetting systems now normalize one point to be exactly 1/72 of an inch, or 0.13889 inch, a 0.4% increase from its original size. Windows applications use points to measure font size. For example, a normal text paragraph uses 10-point font and 12-point line spacing. Postscript language uses points as the basic measurement unit, where all coordinates and dimensions are expressed in points using floating-point numbers.

The English and metric mapping modes would do an adequate job in text formatting, but not accurate or precise enough, so the Win32 API provides an extra mapping mode for this class of application: MM_TWIPS mapping mode. Each logical unit in MM_TWIPS mapping modes is 1/20 of a point, or 1/1440 inch (also called a *twip*). Other than the unique scale, MM_TWIPS mapping mode is the same as the other physical-unit-based mapping modes.

Add the following line to the code shown above to support MM_TWIPS mapping mode:

```
case MM_TWIPS: mul = 14400; div = 254; break;
```

Needless to say, MM_TWIPS mapping mode is 1440 dpi, which should be good enough to allow accurate distribution of errors in text justification, condensation, or expansion.

MM_ISOTROPIC Mapping Modes

“Isotropic” is a physics jargon meaning “having equal physical properties along all axes.” In Win32 GDI, an MM_ISOTROPIC mapping mode is any mapping mode having the same window-to-extent ratio on the two axes, without considering axis directions, or more precisely

$$\text{abs}(W_{\text{Ext}x} / V_{\text{Ext}x}) = \text{abs}(W_{\text{Ext}y} / V_{\text{Ext}y})$$

or $\text{abs}(W_{\text{Ext}x} * V_{\text{Ext}y}) = \text{abs}(W_{\text{Ext}y} * V_{\text{Ext}x})$

To use an isotropic mapping mode, first you need to call SetMapMode(hDC, MM_ISOTROPIC). GDI is found to

borrow window and viewport settings from the MM_LOMETRIC mapping mode. After that, you should call SetWindowExtEx first, and then SetViewportExtEx; GDI will do some magic to make both axes have the same ratio in magnitude.

The actual implementation used on Windows NT/2000 seems to be questionable. From our test cases, it seems that in MM_ISOTROPIC mode, SetWindowExtEx and SetViewportExtEx are first executed as normal; after that, GDI does some normalization to meet the isotropic requirement. Among WExtX, VExtX, WExtY, and VExtY, GDI finds the variable with the largest magnitude and calculates its new magnitude using the other three. [Table 6-1](#) shows some test results.

In our test case, we call SetWindowExtEx followed by SetViewportExtEx. We see clearly that GDI is trying to approximate the ratio by changing only the number with the largest magnitude, a process that can lead to error. The final result, (3, 5) vs. (2, 5), is far from isotropic. For this case, one possible solution is to set window extent to (9, 15) and viewport extent to (15, 25), making the mapping truly isotropic.

Table 6-1. Setup of MM_ISOTROPIC Mapping Mode

API Call (shortened)	Window Extent	Viewport Extent	Comments
SetMapMode(MM_ISOTROPIC)	(3200, 2400)	(1152, -864)	Using MM_LOMETRIC extents, may not be expected.
SetWindowExtEx(3,5)	(3, 5)	(518, -864)	1152 was the largest, replaced with $864 \times 3/5$, 0.07% difference.
SetViewportExtEx(5, 3)	(3, 5)	(2, 3)	5 was the largest, replaced with $3 \times 3/5$,

GDI uses integers to represent window and viewport extents. When trying to adjust them to make a mapping isotropic, sometimes approximation needs to be made, which could generate almost isotropic but not truly isotropic mappings as shown in [Table 6-1](#). We would suggest forgetting about the little benefit provided by MM_ISOTROPIC mode and using MM_ANISOTROPIC mode directly. The former is not worth the trouble.

MM_ANISOTROPIC Mapping Mode

The word “anisotropic” means “not having the same equal physical properties along all axes.” But the MM_ANISOTROPIC mapping mode actually allows any valid window and viewport extents, isotropic or anisotropic.

All the mapping modes we have mentioned so far restrict the way window and viewport extents can be set up. MM_TEXT, MM_LOGENGLISH, MM_HI ENGLISH, MM_LOMETRIC, MM_HIMETRIC, and MM_TWIPS modes have fixed window and viewport extents, which can't be changed by applications. MM_ISOTROPIC mapping modes allow applications to change the extents, but GDI intervenes to force the mapping ratio on both axes to be the same or close enough. MM_ANISOTROPIC is the only mapping mode allowing an application to set up window and viewport extents freely.

Calling SetMapMode(hDC, MM_ANISOTROPIC) changes a device context's mapping mode to be anisotropic, without touching other attributes. After that an application can call SetWindowExtEx and SetViewportExtEx in any order to finish the setup.

MM_ANISOTROPIC mapping mode can be used to implement all the other mapping modes, plus mapping modes of your own design. A Windows application normally allows users to choose the zoom factor under which a document

will be displayed, 500%, 200%, all the way to 25% and 10%. 100% would display one pixel in the document as one pixel on the screen, or one inch in the document as one inch on the screen; 500% would be a 5:1 zoom, and 25% would be a 1:4 zoom. If an image editor uses the pixel as its logical space unit, the code below sets up an isotropic mapping mode for $m:n$ zoom display:

```
SetMapMode(hDC, MM_ANISOTROPIC);
SetExtents(hDC, n, n, m, m);
```

Here SetExtents is a routine responsible for removing common factors in all the parameters and setting the extents.

```
int gcd(int x, int y) // greatest common divisor
{
    while (x!=y)
        if ( x > y) x -= y; else y -= x;

    return x;
}

BOOL SetExtents(HDC hDC, int wx, int wy, int vx, int vy)
{
    int gx = gcd(abs(wx), abs(vx));
    int gy = gcd(abs(wy), abs(vy));

    SetWindowExtEx (hDC, wx/gx, wy/gy, NULL);
    return SetViewportExtEx(hDC, vx/gx, vy/gy, NULL);
}
```

If a word processor is using the twip (1/20 of a printer's point, or 1/1440 inch) as its logical space unit, the code here sets up an $m:n$ zoom:

```
SetMapMode(hDC, MM_ANISOTROPIC)
SetExtents(hDC, n * 1440, n * 1440,
           m * GetDeviceCaps(hDC, LOGPIXELSX),
           m * GetDeviceCaps(hDC, LOGPIXELSY));
```

The code above relies on logical DPI returned by the device driver. If you choose to rely on the device's physical size to calculate its DPI, here is the code:

```
SetMapMode(hDC, MM_ANISOTROPIC);
SetExtents(hDC, n * 1440, n * 1440,
           m * GetDeviceCaps(hDC, HORZRES) * 254
             / GetDeviceCaps(hDC, HORZSIZE) / 10,
           m * GetDeviceCaps(hDC, VERTRES) * 254
             / GetDeviceCaps(hDC, VERTSIZE) / 10);
```

In the code shown above, both extents use positive numbers, so the axes in page coordinate space follow the axes in device coordinate space. Negative signs can be added in viewport extent to change the direction of axes.

Besides these fixed zoom ratios, it's also quite common for Windows applications to provide runtime-calculated zoom ratios that can fit part of a document to the whole device surface. For example, *page width zoom* fits the width of a page to the device surface width, *whole page zoom* fits a full page to a device surface, and *dual page zoom* fits two pages to a device surface. [Listing 6-1](#) shows a generic routine for this kind of problem:

Listing 6-1 Fit col by row Pages into a Window

```
BOOL FitPages(hDC hDC, int pagewidth, int pageheight,
    int dcwidth, int dcheight,
    int col, int row, int margin, int gap)
{
    // calculate final dimension for col by row pages
    int width = margin*2 + pagewidth *col + gap*(col-1);
    int height = margin*2 + pageheight*row + gap*(row-1);

    if (width <=0) width = 1; // avoid zero
    if (height <=0) height = 1; // avoid zero

    // find the smaller ratio, use it to adjust the other
    if ( dcheight * width > dcwidth * height )
    {
        dcheight = dcwidth;
        height = width;
    }
    else
    {
        dcwidth = dcheight;
        width = height;
    }

    return SetExtents(hDC, width, height, dcwidth, dcheight);
}
```

The routine handles the generic case of fitting col-by-row pages, each pagewidth by pageheight in size, into a device coordinate space dcwidth by dcheight in size. The whole document can have margins on four sides and gaps between the pages. The code first calculates the final dimension of col-by-row pages, considering all the factors. It then compares the viewport-to-window ratio around the two axes, using the smaller ratio to adjust the viewport extent along the axis with larger extent.

Let's look at a few examples to illustrate this routine. Consider a simple situation: each page in the document is 850 by 1100 units, the device surface is 1024 by 768 pixels, and there is zero margin, zero gap.

Page-width zoom will be considered one column by zero row, or FitPages(hDC, 850, 1100, 1024, 768, 1, 0, 0, 0). A one-by-zero page is 850 by 1 (adjusted to avoid dividing by zero) units in size, so the ratio on the y-axis (768:1) is higher than the ratio on the x-axis (1024:850). The code adjusts viewport height to be 1024 and window height to be

850. The final window extent is (850, 850); viewport extent is (1024, 1024). The width of the page is fit into the width of a device coordinate space.

Now let's jump to fit two pages to a device surface. A two-by-one page is 1700 by 1100 units in size, so the viewport-to-window ratio on the y-axis (768:1700) is smaller than the ratio on the x-axis (1024:1100). The code adjusts viewport width to be 768 and window width to be 1100. Now we are mapping (1100, 1100) units to (768, 768) pixels, which will be simplified to (275, 275) to (192, 192) by SetExtents.

Setting up window and viewport extents takes care of how many units in the page coordinate space are mapped to how many units in the device coordinate space. This is different from other graphics API, where the window and the viewport also serve as the clipping region. In the Windows GDI, the window and the viewport is a way to set up the page coordinate space to the device coordinate space mapping; clipping is handled independently in the device coordinate space. The only thing important here is the ratio of the extents, not the value itself. For example, the MM_TEXT mapping uses (1, 1) in both window and viewport extents, and (0, 0) as both origins. This does not mean that only one pixel can be mapped.

Window and Viewport Origins

Once window and viewport extents are set up, an application needs to set up the window and viewport origins. Once set up properly, GDI will map the window origin in a page coordinate space to the viewport origin in a device coordinate space; other positions will be mapped accordingly.

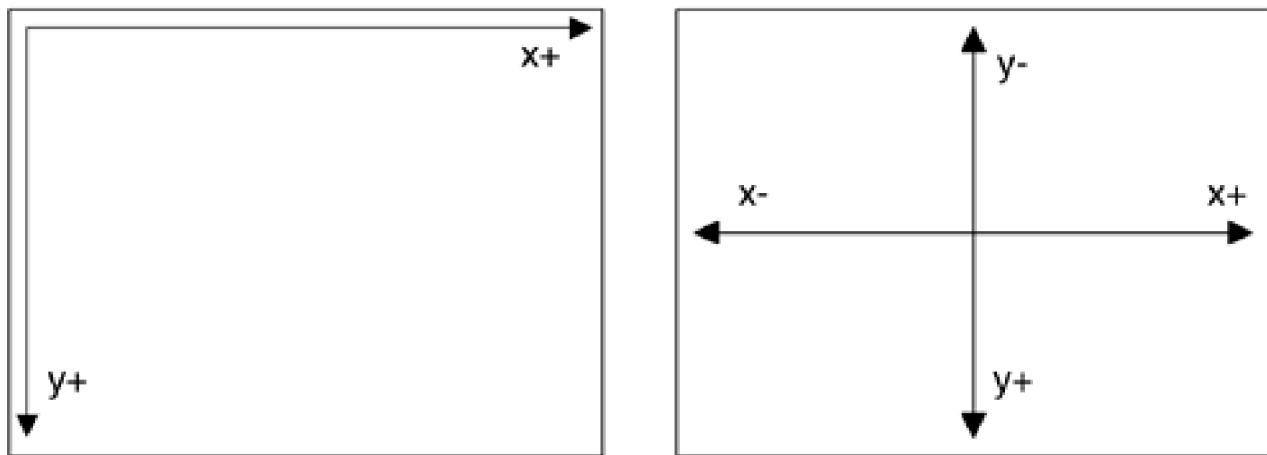
The default values for both window origin and viewport origin are (0, 0); setting up mapping modes and extents does not change origins. A programmer needs to understand the direction of x-and y-axes implied by the window and viewport setup and to determine if window and viewport origins need to be changed.

For MM_TEXT and the default MM_ANISOTROPIC mapping modes, the x-axis goes from left to right, and the y-axis goes top down. So in a page coordinate space, only the first quadrant having positive x and y values is mapped to the device coordinate space, while the rest is not visible. If you want to map the origin of a page coordinate space to the center of a device coordinate space, use this:

```
SetWindowOrgEx(hDC, 0, 0, NULL);
SetViewportOrgEx(hDC, dcHeight/2, dcWidth/2, NULL);
```

Figure 6-4 illustrates the change.

Figure 6-4. Change the MM_TEXT mapping mode origin.



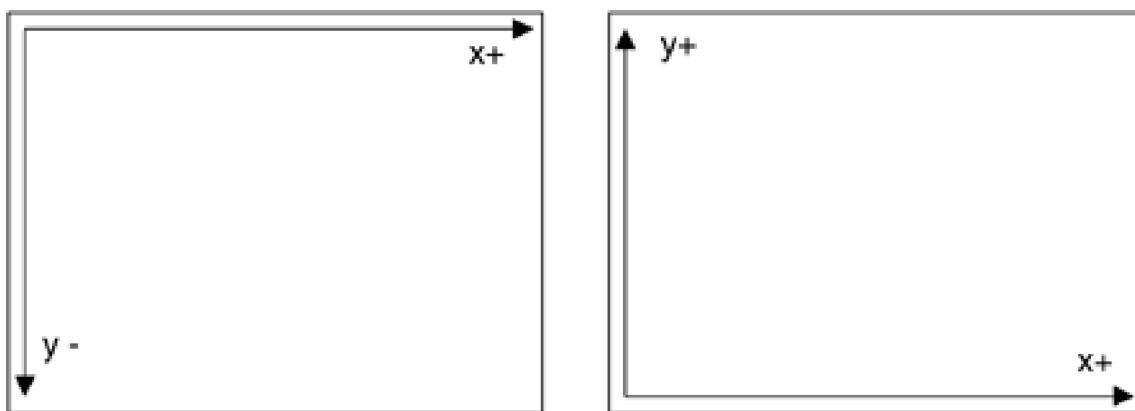
Note this is not the only way to achieve this mapping. You can also keep the viewport origin as (0, 0), and change the window origin to $(-\text{dcHeight} * \text{WExtx}/\text{VExtx}/2, -\text{dcWidth} * \text{WExty}/\text{VExty}/2)$. Mathematically the two methods are the same, but the second method may have rounding-off errors because of the fraction computation.

For all the other mapping modes in their default setting, the direction of the y-axis is different from MM_TEXT mapping mode; it follows the traditional Cartesian space: positive y is up. Now only the second quadrant, having positive x and negative y, is mapped to the device address space. Mapping the window origin to the center of the device space could still use the same code shown above. But if you only want to map the first quadrant to the device space, here is the code:

```
SetWindowOrgEx(hdc, 0, 0, NULL);  
SetViewportOrgEx(hdc, dcHeight, 0, NULL);
```

Postscript language uses the same y-axis orientation. [Figure 6-5](#) illustrates the change.

Figure 6-5. Change English, metric, and twip mapping modes origin.



For MM_ISOTROPIC and MM_ANISOTROPIC mapping modes, an application is free to define axis directions. Care should be taken that GDI is supposed to honor the setting to implement the drawing primitive accordingly. There is one exception: GDI always draws text strings in one direction except when a device context is in the advanced graphics mode. Even if you set up the x-axis to go from right to left, text strings are still displayed from left to right, without each glyph being reflected. This could be very annoying if an application is using mapping mode to flip pages along the y-axis. The solution to this problem is either simulating text reflection using a memory device

context or using the world transformation in a world coordinate space.

Other Window and Viewport Functions

The Win32 API provides a few miscellaneous functions to help in managing current window and viewport setting. GetMapMode returns the current mapping mode. OffsetWindowOrgEx and OffsetViewportOrgEx move the window or viewport origin. They will be useful to perform incremental changes to window and viewport, for example, in response to a window's scroll messages. ScaleWindowExtEx and Scale ViewportExtEx scale window or viewport extent by a fractional number, which could be handy in processing zoom requests.

[< BACK](#) [NEXT >](#)

6.4 THE WORLD COORDINATE SPACE

To professional graphics programming, the Win32 window-to-viewport mapping, and the use of various mapping modes, is a compromise—a compromise deep-rooted in the original Win16 GDI API design, when the CPU was running at 8-MHz clock cycle and memory was expensive. This forced GDI design and implementation to be simple and efficient. Here are some of the shortcomings of the page coordinate space design.

- **Fractional mapping from window to viewport.** Both window and viewport are represented using integers. So the mapping from window to viewport is fractional-number mapping, instead of real-number mapping. Fractional numbers are easy to compute using an integer multiplication, followed by an integer division. But the numbers to choose from are limited, and it's easier to lose precision. We have seen that the MM_ISOTROPIC mode can generate different mapping ratios on two axes.
- **Incomplete implementation.** Text drawing does not implement the reflection along-axis semantics. That is, if an application sets the *x*-axis to go from right to left, the text string is still displayed from left to right.
- **Limited transformation.** The window-to-viewport mapping allows only translation, scale, and reflection. Rotation and shearing are not supported, which are very hard to simulate without direct GDI support.

As a solution to these problems, Windows NT/2000 implements a new logical coordinate space: the world coordinate space. The world coordinate space still uses 32-bit integers as coordinates, unlike other graphics systems like Postscript, which use floating point numbers. Points in a world coordinate space are mapped to a page coordinate space using a more generic transformation.

Affine Transformations

There could be different kinds of transformation from one coordinate space to another. A fish-eye transformation simulates what can be seen through a fish-eye camera lens; perspective projection maps objects in 3D space onto 2D surface. The class of transformation supported by Windows NT/2000 is two-dimensional *affine* transformation. An affine transformation maps parallel lines to parallel lines, and finite points to finite points.

A 2D affine transformation is defined by six numbers, forming a 2×3 matrix. The Win32 API uses the XFORM structure to define such a matrix.

```
typedef struct _XFORM {
    FLOAT eM11;
    FLOAT eM12;
    FLOAT eM21;
    FLOAT eM22;
    FLOAT eDx;
    FLOAT eDy;
} XFORM;
```

An affine transformation defined above transforms point (x, y) to (x', y') , where:

```
x' = eM11 * x + eM21 * y + eDx;  
y' = eM12 * x + eM22 * y + eDy;
```

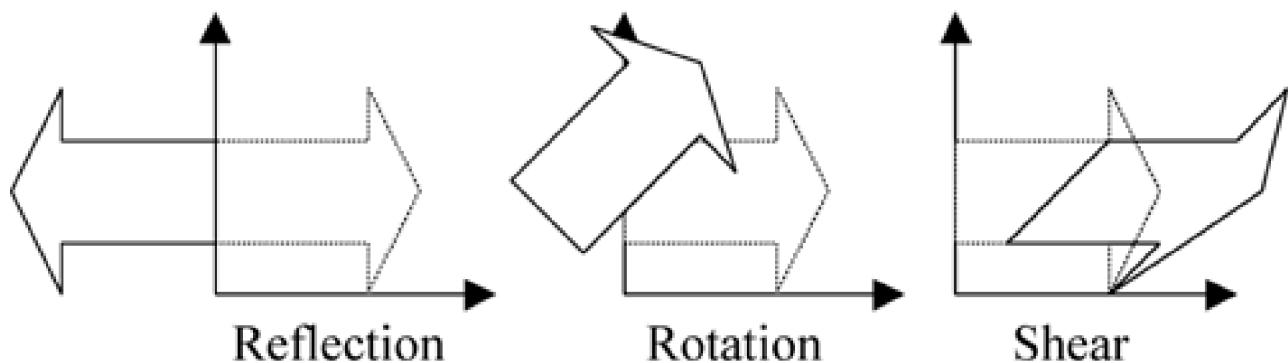
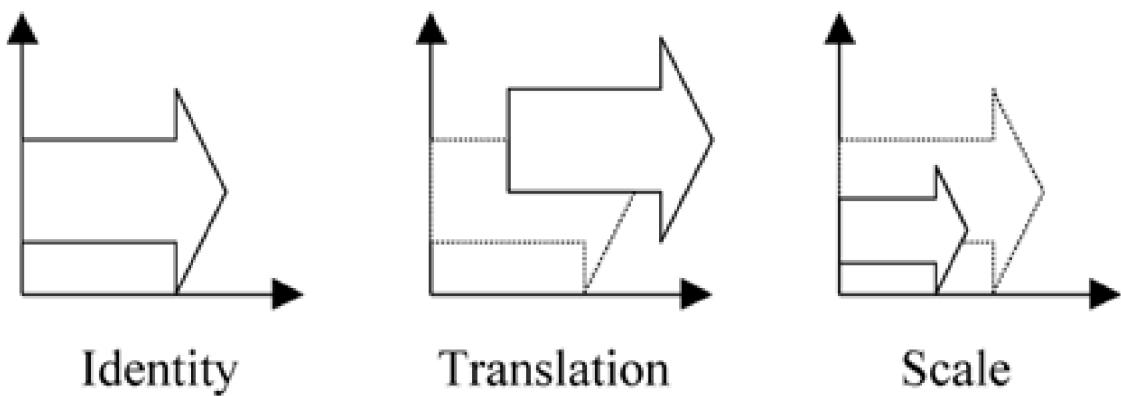
This may look similar to the window-to-viewport mapping formulae, but actually it's more powerful.

Window-to-viewport mappings used by a page coordinate space can be seen as a special kind of affine transformation, where $eM21$ and $eM12$ are both zeros. This is what an affine transformation can do:

- **Identity:** Set matrix to $\{1, 0, 0, 1, 0, 0\}$ $x'=x$, and $y'=y$.
- **Translation:** Set matrix to $\{1, 0, 0, 1, dx, dy\}$, $x'=x+dx$ and $y'=y+dy$.
- **Scale:** Set matrix to $\{mx, 0, 0, my, 0, 0\}$, $x'=mx*x$ and $y'=my*y$.
- **Reflection:** Set matrix to $\{-1, 0, 0, -1, 0, 0\}$ $x'=-x$ and $y'=-y$.
- **Rotation:** Set matrix to $\{\cos(\theta), \sin(\theta), -\sin(\theta), \cos(\theta), 0, 0\}$ $x'=\cos(\theta)*x - \sin(\theta)*y$, $y'=\sin(\theta)*x + \cos(\theta)*y$. This rotates (x, y) by angle θ counterclockwise about the origin.
- **Shearing:** Set matrix to $\{1, s, 0, 1, 0, 0\}$, $x'=x+s*y$ and $y'=y$. This translates each x coordinate by an amount proportional to y .
- **Combination:** Multiple affine transformations can be combined using matrix multiplication to form a new affine transformation.

[Figure 6-6](#) illustrates six simple transformations.

Figure 6-6. Simple 2D affine transformations.



We have seen identity, translation, scale, and reflection with a page coordinate to device coordinate mapping, although now we are using floating-point numbers freely. *Rotate* is something new and very useful here. To rotate the coordinate space about its origin (0, 0), the formula is:

$$\begin{aligned}x' &= \cos(\theta) * x - \sin(\theta) * y \\y' &= \sin(\theta) * x + \cos(\theta) * y\end{aligned}$$

The generic problem of rotating about an arbitrary point (x_0, y_0) can be achieved in three steps: First translate the coordinate space by $(-x_0, -y_0)$, then rotate, and finally translate the coordinate space back by (x_0, y_0) . The combined transformation is:

$$\begin{aligned}x' &= \cos(\theta) * (x-x_0) - \sin(\theta) * (y-y_0) + x_0 \\y' &= \sin(\theta) * (x-x_0) + \cos(\theta) * (y-y_0) + y_0\end{aligned}$$

Shearing, meaning lateral deformation, adds to one coordinate a value proportional to the other coordinate. It can also be applied to both coordinates. So the general form of shearing is:

$$\begin{aligned}x' &= x + h * y \\y' &= g * x + y\end{aligned}$$

Affine transformations have several interesting properties that make them popular in computer graphics. Understanding these properties helps us understand how application-defined geometric models are going to be transformed under these transformations. It also gives us some insight into how the graphics engine implements them internally.

- Affine transformation preserves lines. A straight line will be mapped to a straight line, a triangle to a triangle, and a polygon to a polygon. To implement affine transformations for lines and polygons, the graphics engine only needs to transform their vertexes, then draw the lines or polygons using the transformed vertexes.
- An affine transformation preserves parallelisms. Parallel lines are mapped to parallel lines. So parallelograms map to parallelograms, but rectangles may not always be mapped to rectangles, and squares may not always be mapped to squares.
- An affine transformation preserves ellipses. An ellipse is always mapped to an ellipse under an affine transformation, but a circle is not always mapped to a circle. Note that in a logical coordinate space, the GDI API only allows definition of orthogonal circles and ellipses, which have axes parallel to the x-, y-axes of the coordinate space. Affine transformations allow you to draw any ellipse on a device surface.
- An affine transformation preserves Bezier curves. Just like lines and polygons, GDI only needs to apply affine transformation to vertexes defining a Bezier curve, then link them to form a transformed Bezier curve in a device coordinate space.
- An affine transformation is uniquely defined by three noncollinear points p, q, r in a logical coordinate space, and three noncollinear points p', q', r' in a device coordinate space. That is, there is a unique affine transformation that maps p, q , and r to $p', q',$ and r' , respectively. Note that each mapping that GDI supports from a page coordinate space to a device coordinate space is uniquely defined by two points in each space.

These properties of affine transformations are reflected in the Windows NT/2000's implementation of GDI drawing primitives. We mentioned that a circle or ellipse is mapped to an ellipse under an affine transformation, which may not be orthogonal to axes. An arbitrary ellipse is very hard to draw effectively in computer graphics. What is the graphics engine's solution? It breaks every ellipse into four Bezier curves, which can be easily mapped and drawn in a device coordinate space as Bezier curves. Because an affine transformation is defined using floating-point numbers, instead of fractions as in window-to-viewport mappings, Windows NT/2000 graphics engine uses fixed-point numbers internally to make a calculation more accurate.

Besides looking at individual transformations, affine transformations can also be studied as a whole. Mathematically, an affine transformation is a function $t: R^* R \rightarrow R^* R$ of the form: $t(x) = Ax + b$, where A is an invertible 2×2 matrix and b is a point in $R^* R$. The set of all affine transformations of $R^* R$ is denoted by $A(2)$. It can be shown that the set of affine transformations $A(2)$ forms a group under the operations of function composition. The term "group" here is an abstract-algebra-system concept, which is defined as any set of operations on a domain having the following properties: closure, identity, inverse, and associativity. To be more specific, for the set of affine transformations, we have:

- **Closure.** The composition of any two affine transformations is an affine transformation. For $t_1(x) = A_1 * x + b_1$, and $t_2(x) = A_2 * x + b_2$, the composite $(t_1 * t_2)(x) = (A_1 * A_2) * x + (A_1 * b_2 + b_1)$.
- **Identity.** There exists an identity affine transformation $i(x)$ such that for any affine transformation $t(x)$, we have $(i * t)(x) = t(x)$, and $(t * i)(x) = t(x)$. Actually, $i(x) = I * x + 0$, where I is the 2×2 identity matrix.
- **Inverse.** The inverse of an affine transformation is an affine transformation. For the inverse transformation of $t(x) = A * x + b$ is $(1/A) * x - (1/A) * b$.
- **Associativity.** The composition of affine transformations is associative; that is, for any affine

transformations t_1 , t_2 , and t_3 , we have $(t_1 * t_2) * t_3 = t_1 * (t_2 * t_3)$.

These seemingly abstract mathematical concepts are very useful to computer graphics. Quite often, an application needs to define multiple stages of transformation to be applied to drawing objects. The closure property ensures that multiple transformations can be combined into one transformation. This is used internally by the graphics engine to combine the world coordinate space to page coordinate space transformation, and the page coordinate space to device coordinate space mapping into one single transformation. Remember here that a page coordinate space to a device coordinate space is a special class of affine transformations. The inverse property makes it easier to manage the reverse mapping from a page or device coordinate space back to a world coordinate space. This is useful in mapping coordinates received from mouse events to coordinates in a world coordinate space. The graphics engine can use the same transformation code with an inverse transformation.

The WIN32 API for World Transformations

So much for the geometry mathematics on affine transformations; now let's look at the Win32 GDI API support for a world coordinate space and world transformations.

A default device context is operating in what's called *compatible graphics mode*, to be compatible with 16-bit GDI semantics. In compatible mode, a world coordinate space is not supported, and the only logical coordinate space is the page coordinate space. If an application wants to enable a world coordinate space, it needs to switch the device context's graphics mode by calling `SetGraphicsMode(hDC, GM_ADVANCED)`, which is implemented only on Windows NT/2000. After that, a device context supports two layers of logical coordinate space: a world coordinate space and a page coordinate space, and a transformation matrix between the two. The current graphics mode can be queried by `GetGraphicsMode(hDC)`. To switch back to compatible mode, reset the transformation matrix to the identity matrix and call `SetGraphicsMode(hDC, GM_COMPATIBLE)`. Or you can just use `SaveDC` and `RestoreDC`.

Setting a device context to `GM_ADVANCED` mode does more than enable a world coordinate space. It also has significant implications in the GDI drawing primitive implementation. [Table 6-2](#) summarizes the differences between two graphics modes.

Table 6-2. Differences in GDI Graphics Modes

Area	GM_COMPATIBLE	GM_ADVANCED
Logical coordinate space	Page coordinate space	World coordinate space and page coordinate space
Platform supported	Windows 95/98, NT/2000	Windows NT/2000
Font direction	Text is always displayed left to right and right-side up, even when axes are reflected in mapping modes. The only way to change text direction is to use escapement, logical font orientation, or a memory device context.	Text strings conform to transformation and mapping.
Font scale	Only height of TrueType font is scaled.	TrueType and vector fonts are scaled properly. GDI tries to produce the best quality for raster fonts, but may not be smooth.
Rectangle	Right and bottom edges are excluded.	Right and bottom edges are inclusive.
Arc direction	Arcs follow direction set by SetArc-Direction.	Arcs are always drawn counter clockwise in logical space.
Arc mapping, transformation	Arc does not follow axis reflection.	Arc conforms to transformation and mappings.

In a device context, the default transformation between a world coordinate space and a page coordinate space is an identity matrix. To modify the current transformation, the following functions can be used:

```
BOOL SetWorldTransform(HDC hDC, CONST XFORM * lpXform);
BOOL ModifyWorldTransform(HDC hDC,
    CONST XFORM * lpXform, DWORD iMode);
```

SetWorldTransform simply overwrites the transformation attribute in a device context with a new transformation as defined by an XFORM structure. Note that not every six FLOAT numbers define a valid affine transformation. The formal definition for an affine transformation requires the 2-by-2 matrix formed by eM11, eM12, eM21, and eM22 to be invertible; that is, $eM11 \cdot eM22 \neq eM12 \cdot eM21$. For example, the following code will fail:

```
XFORM xm = { 1, 2, 1, 2, 3, 4};
SetGraphicsMode(hDC, GM_ADVANCED);
BOOL result = SetWorldTransform(hDC, & xm);
DWORD error = GetLastError();
```

The transformation defined by xm is $x' = x + y + 3$, and $y' = 2x + 2y + 4$, so we have $y' = 2x' + 1$. All the points in the world coordinate system are mapped into one single line $y = 2x + 1$ in the page coordinate system. This is neither an invertible mapping nor a one-to-one mapping, so it's not a valid transformation. In this case, SetWorldTransform returns FALSE as expected, but unbelievably GetLastError returns 0 (ERROR_SUCCESS), even on Windows NT/2000. GDI is not doing a good job explaining the reason for errors. SetWorldTransform also fails when the device context is in GM_COMPATIBLE graphics mode, or the program is running on Windows 95/98.

ModifyWorldTransform provides three options using the iMode parameter. When iMode is MTW_IDENTITY,

transformation is reset to the identity transform. If it's MTW_LEFTMULTIPLY, current transformation is multiplied on the left by *lpXform; if it's MTW_RIGHTMULTIPLY, current transformation is multiplied on the right by *lpXform. Here, multiplying means the composition of two transformations to form a new transformation. The reason GDI has to distinguish left and right multiplication is that multiplication of transformations is not always commutative. Unlike integer arithmetic, for transformations, $a * b$ does not always equal $b * a$.

To query the current transformation, use GetWorldTransform. This concludes Win32 API's support for the wonderful world of transformation. Beyond that, you could rely on your old, analytical geometry book, computer graphics book, or read on.

Using World Transformation

A world coordinate space and a world transformation using two-dimensional affine transformations are very powerful computer graphics techniques. But the support for them in the Win32 GDI API is very limited. To solve even a simple practical problem, you have to scratch your head. For example, how do you rotate an object around an arbitrary point (x, y) , how do you figure out transformations that can map a rectangular image onto faces of a three-dimensional cube?

We are now going to develop a C++ class, KAffine, which has quite a few useful methods. [Listing 6-2](#) shows the declaration of KAffine class.

Listing 6-2 KAffine Class Declaration: Affine Transformation

```
class KAffine
{
public:
    XFORM m_xm;

    KAffine()
    {
        Reset();
    }

    void Reset();
    BOOL SetTransform(const XFORM & xm);
    BOOL Combine(const XFORM & b);

    BOOL Invert(void);
    BOOL Translate(FLOAT dx, FLOAT dy);
    BOOL Scale(FLOAT sx, FLOAT dy);
    BOOL Rotate(FLOAT angle, FLOAT x0=0, FLOAT y0=0);

    BOOL MapTri(FLOAT px0, FLOAT py0, FLOAT qx0, FLOAT qy0,
                FLOAT rx0, FLOAT ry0);

    BOOL MapTri(FLOAT px0, FLOAT py0, FLOAT qx0, FLOAT qy0,
                FLOAT rx0, FLOAT ry0, FLOAT px1, FLOAT py1,
                FLOAT qx1, FLOAT qy1, FLOAT rx1, FLOAT ry1);
```

};

The class manages an affine transformation as represented by the WIN32 XFORM structure, so the only data member m_xm is an XFORM structure. The first three methods repeat what's provided by the WIN32 API: Reset sets an identity transform, SetTransform copies from an XFORM structure, and Combine does transformation composition.

The KAffine::Invert method replaces the current transformation with its inverse transformation. The three methods after that are actually PostScript operators. The translate method adds a translation to the current transformation. The scale method changes the unit length on the two axes. The rotate method adds a rotation around an arbitrary point (x0, y0) to the current transformation. All three methods can be seen as creating a simple transformation that performs the translation, scaling, or rotation, and then combines it with the current transformation.

The two MapTri methods are defined to achieve one goal: Find an affine transformation that maps three noncollinear points p0, q0, and r0 to three noncollinear points p1, q1, and r1. The task is divided into two steps. We first need to find two transformations that map points (0, 0), (1, 0), and (0, 1) to the two triplets. This is a much easier task than the original one; it's performed by the first MapTri method. We then invert the first transformation and combine it with the second to form the final transformation. The trick can be visualized as map p0, q0, and r0 to (0, 0), (1, 0), and (0, 1) first, and then map the latter to p1, q1, and r1.

If you don't mind some mathematics, [Listing 6-3](#) shows the KAffine class implementation, with some error checking.

Listing 6-3 KAffine Class Implementation: Affine Transformations

```
#define STRICT
#define WIN32_LEAN_AND_MEAN

#include <windows.h>
#include <math.h>
#include "Affine.h"

// Reset to identity transform
void KAffine::Reset()
{
    m_xm.eM11 = 1;
    m_xm.eM12 = 0;
    m_xm.eM21 = 0;
    m_xm.eM22 = 1;
    m_xm.eDx = 0;
    m_xm.eDy = 0;
}

// Copy transform if valid
BOOL KAffine::SetTransform(const XFORM & xm)
{
    if ( xm.eM11 * xm.eM22 == xm.eM12 * xm.eM21 )
        return FALSE;

    m_xm = xm;
```

```
    return TRUE;
}

// transform = transform * b
BOOL KAffine::Combine(const XFORM & b)
{
    if ( b.eM11 * b.eM22 == b.eM12 * b.eM21 )
        return FALSE;
    XFORM a = m_xm;

    // 11 12 11 12
    // 21 22 21 22
    m_xm.eM11 = a.eM11 * b.eM11 + a.eM12 * b.eM21;
    m_xm.eM12 = a.eM11 * b.eM12 + a.eM12 * b.eM22;
    m_xm.eM21 = a.eM21 * b.eM11 + a.eM22 * b.eM21;
    m_xm.eM22 = a.eM21 * b.eM12 + a.eM22 * b.eM22;
    m_xm.eDx = a.eDx * b.eM11 + a.eDy * b.eM21 + b.eDx;
    m_xm.eDy = a.eDx * b.eM12 + a.eDy * b.eM22 + b.eDy;

    return TRUE;
}

// transform = 1 / transform
// M = A * x + B
// Inv(M) = Inv(A) * x - Inv(A) * B
BOOL KAffine::Invert(void)
{
    FLOAT det = m_xm.eM11 * m_xm.eM22 - m_xm.eM21 * m_xm.eM12;

    if ( det==0 )
        return FALSE;

    XFORM old = m_xm;

    m_xm.eM11 = old.eM22 / det;
    m_xm.eM12 = -old.eM12 / det;
    m_xm.eM21 = -old.eM21 / det;
    m_xm.eM22 = old.eM11 / det;

    m_xm.eDx = - ( m_xm.eM11 * old.eDx + m_xm.eM21 * old.eDy );
    m_xm.eDy = - ( m_xm.eM12 * old.eDx + m_xm.eM22 * old.eDy );

    return TRUE;
}

BOOL KAffine::Translate(FLOAT dx, FLOAT dy)
{
    m_xm.eDx += dx;
    m_xm.eDy += dy;
```

```
return TRUE;
}

BOOL KAffine::Scale(FLOAT sx, FLOAT sy)
{
    if ( (sx==0) || (sy==0) )
        return FALSE;

    m_xm.eM11 *= sx;
    m_xm.eM12 *= sx;
    m_xm.eM21 *= sy;
    m_xm.eM22 *= sy;
    m_xm.eDx *= sx;
    m_xm.eDy *= sy;

    return TRUE;
}

BOOL KAffine::Rotate(FLOAT angle, FLOAT x0, FLOAT y0)
{
    XFORM xm;

    Translate(-x0, -y0); // make (x0,y0) the origin
    double rad = angle * (3.14159265359 / 180);

    xm.eM11 = (FLOAT) cos(rad);
    xm.eM12 = (FLOAT) sin(rad);
    xm.eM21 = -xm.eM12;
    xm.eM22 = xm.eM11;
    xm.eDx = 0;
    xm.eDy = 0;

    Combine(xm); // rotate
    Translate(x0, y0); // move origin back

    return TRUE;
}

// Find maps from (0,0) (1,0) (0,1) to p, q, and r
BOOL KAffine::MapTri(FLOAT px0, FLOAT py0, FLOAT qx0,
                      FLOAT qy0, FLOAT rx0, FLOAT ry0)
{
    // px0 = dx, qx0 = m11 + dx, rx0 = m21 + dx
    // py0 = dy, qy0 = m12 + dy, ry0 = m22 + dy
    m_xm.eM11 = qx0 - px0;
    m_xm.eM12 = qy0 - py0;
    m_xm.eM21 = rx0 - px0;
    m_xm.eM22 = ry0 - py0;
```

```
m_xm.eDx = px0;
m_xm.eDy = py0;

return m_xm.eM11 * m_xm.eM22 != m_xm.eM12 * m_xm.eM21;
}

// Find map from p0, q0, and r0 to p1, p1 and r1
BOOL KAffine::MapTri(FLOAT px0, FLOAT py0, FLOAT qx0,
                      FLOAT qy0, FLOAT rx0, FLOAT ry0,
                      FLOAT px1, FLOAT py1, FLOAT qx1,
                      FLOAT qy1, FLOAT rx1, FLOAT ry1)
{
    if ( ! MapTri(px0, py0, qx0, qy0, rx0, ry0) )
        return FALSE;

    Invert(); // map p0, q0, r0 to (0,0),(1,0),(0,1)

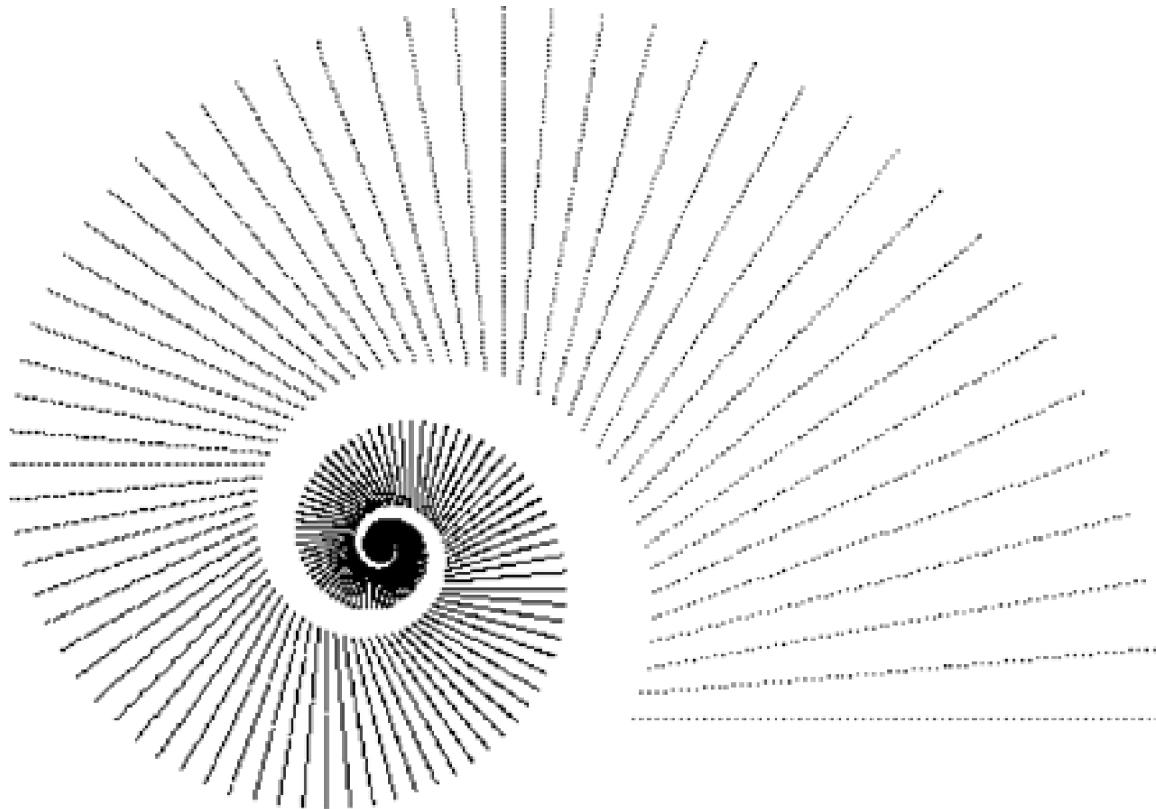
    KAffine map1;

    if (! map1.MapTri(px1, py1, qx1, qy1, rx1, ry1) )
        return FALSE;

    return Combine(map1.m_xm); // then to p1,r1,q1
}
```

Affine transformations are widely used in other graphics systems like Postscript. With this basic KAffine class, it's much easier to convert pretty Postscript examples into GDI calls. [Figure 6-7](#) shows a simple example which draws simple dotted lines repetitively under a gradually changing transformation to form an interesting pattern.

Figure 6-7. Dotted line under a world transformation.



```
void Transform_DottedLine(HDC hDC, int width, int height)
{
    KAffine af;

    // Cartesian coordinate system with origin in center
    SetMapMode(hDC, MM_ANISOTROPIC);
    SetViewportExtEx(hDC, 1, -1, NULL);
    SetViewportOrgEx(hDC, width/2, height/2, NULL);

    SetGraphicsMode(hDC, GM_ADVANCED);

    for (int i=0; i <=72*5; i++)
    {
        // dotted line (50,0) -> (248, 0)
        for (int x=0; x <=200; x+=3)
            SetPixel(hDC, x+50, 0, 0);

        af.Translate(5, 5);           // move away
        af.Scale((FLOAT) 0.98, (FLOAT) 0.98); // gradually smaller
        af.Rotate(5);                // 5 degree
        SetWorldTransform(hDC, & af.m_xm);
    }
}
```

Would you agree that the calls to Translate, Scale, and Rotate are quite similar to what's in Postscript—that is, “5 pixel 5 pixel translate 0.98 0.98 scale 5 rotate”?

Now let's try something more interesting—draw a three-rectangle array of pixels of different colors and map them to three faces of a cube using parallel projection. This is illustrated by [Figure 6-8](#) and the following code:

```
void Face(HDC hDC, COLORREF color)
{
    for (int x=0; x<100; x++)
        for (int y=0; y<100; y++)
            SetPixel(hDC, x, y, color);
}

void Draw_Cube(HDC hDC, int width, int height)
{
    KAffine af;

    SetMapMode(hDC, MM_ANISOTROPIC);
    SetViewportExtEx(hDC, 1, -1, NULL);
    SetViewportOrgEx(hDC, width/2, height/2, NULL);

    SetGraphicsMode(hDC, GM_ADVANCED);

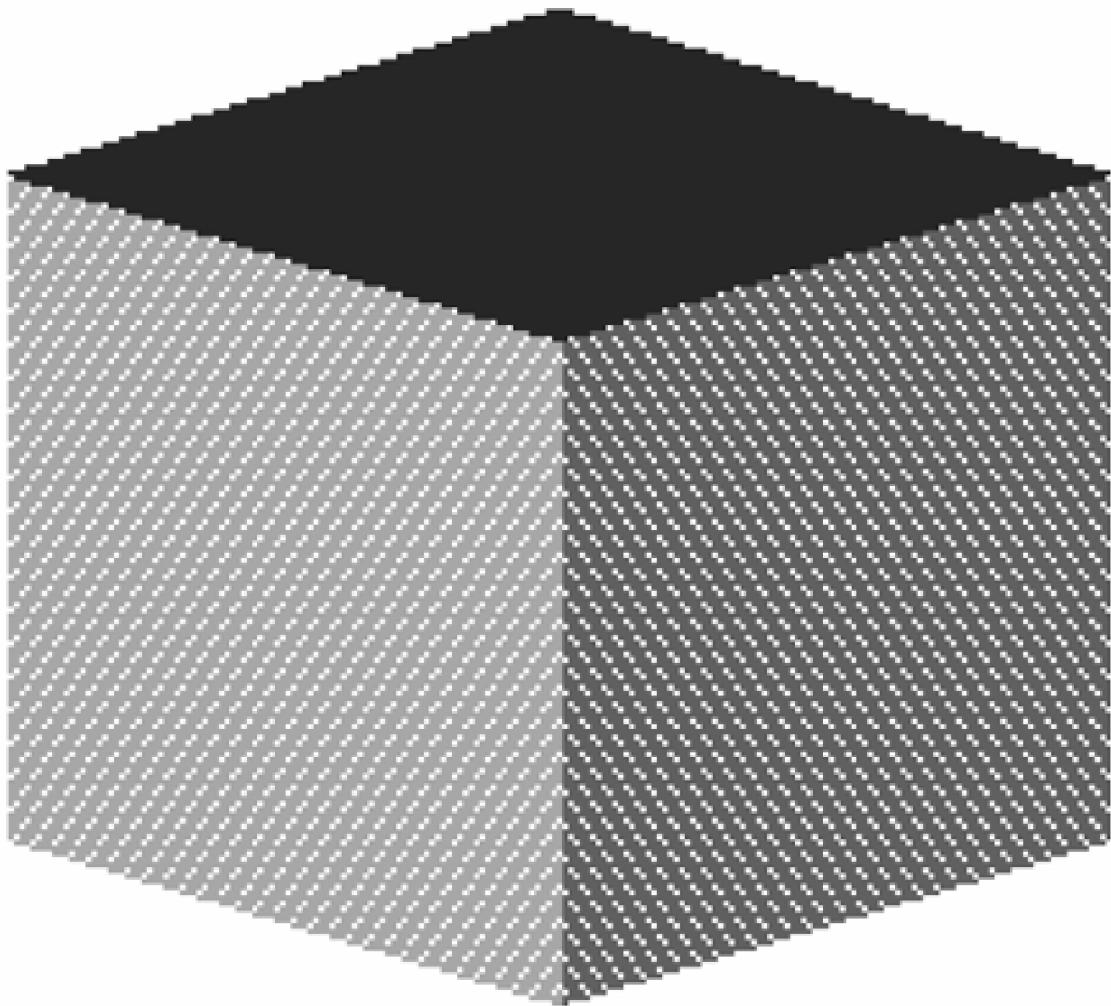
    const FLOAT dx = 100, dy = 30, h = 120;

    af.MapTri(0,0, 100,0, 0,100, 0,0, dx,dy, 0,h);
    SetWorldTransform(hDC, & af.m_xm);
    Face(hDC, RGB(0xFF, 0, 0));

    af.MapTri(0,0, 100,0, 0,100, 0,0, -dx,dy, 0,h);
    SetWorldTransform(hDC, & af.m_xm);
    Face(hDC, RGB(0, 0xFF, 0));

    af.MapTri(0,0, 100,0, 0,100, 0,h, dx,h + dy, -dx,h + dy);
    SetWorldTransform(hDC, & af.m_xm);
    Face(hDC, RGB(0, 0, 0xFF));
}
```

Figure 6-8. Color cube using a transformation.



The routine Face draws a rectangular array of pixels in a world coordinate space, without worrying about their final destination. The main routine uses the KAffine ::MapTri method to map the rectangle area drawn by Face to three surfaces of a cube in a page coordinate space. The page coordinate space is set up according to Cartesian coordinate space with the y-axis going up. With the world transformation, the same drawing code can be reused to draw into different places and different shapes, if you can calculate the right transformation. On each face, you're certainly not restricted in GDI pixel drawing commands. Although what's shown here is only basic pixel draw, you can change it to lines, ellipses, bitmaps, and even texts.

6.5 USING COORDINATE SPACES

We've discussed in quite some length the four coordinate spaces supported by the Windows graphics system—that is, world coordinate space, page coordinate space, device coordinate space, and physical device coordinate space. A world transformation maps a world coordinate space to a page coordinate space. A window-to-viewport mapping maps a page coordinate space to device coordinate space. A device coordinate space is simply a rectangular region within the physical device coordinate space, so the mapping between them is a simple translation.

Both world coordinate space and page coordinate space are referred to as logical coordinate spaces in Win32 terminology. The mappings between these four coordinate spaces are independent from each other, in that when you change one mapping, others are not changed. Yet collectively they all contribute to the mapping from the application's geometric model to locations on a physical device surface. GDI provides functions for the application to define a world transformation and window-to-viewport mapping. The mapping from a device coordinate space and a physical device coordinate space is managed by the operating system.

In a default device context, the world transformation is disabled, because the device context is in Win16 GDI compatible mode. An application needs to change graphics mode to GM_ADVANCED mode to enable the world transformation. So the default mapping from world to page coordinate spaces is an identity mapping. The default page-coordinate-space to device-coordinate-space mapping is also an identity mapping—that is, the MM_TEXT mapping mode with window and viewport both start at (0, 0).

An application can also query the mapping between the four coordinate spaces.

`GetWorldTransformation` returns the mapping between world and page coordinate spaces.

`GetWindowOrgEx`, `GetWindowExtEx`, `GetViewportOrgEx`, and `GetViewportExtEx` return the mapping between page and device coordinate spaces. `GetDCOrgEx` returns the translation between device and physical device coordinate space.

Normally the mapping from the topmost logical coordinate space to the physical device coordinate space is managed by the graphics system. But applications do receive data in different coordinate spaces from time to time, and they need to manage the mapping among them. Win32 provides several functions:

```
BOOL LPtoDP(HDC hDC, LPPOINT lpPoints, int nCount);
BOOL DPtoLP(HDC hDC, LPPOINT lpPoints, int nCount);
BOOL ClientToScreen(HWND hWnd, LPPOINT lpPoint);
BOOL ScreenToClient(HWND hWnd, LPPOINT lpPoint);
```

Function `LPtoDP` maps an array of `POINT` structures from a world coordinate space to a device coordinate space; `DPtoLP` does the opposite. A device context's graphics mode, world transformation, mapping mode, and window-to-viewport mapping determine the mappings. The

functions accept a counted array of POINT structures, not only a single POINT. So it's safe to pass a RECT structure, or an array of RECT structures, to these two functions, because a RECT structure has the same layout as two POINT structures, one for the top-left corner, one for the bottom-right corner. Here is an example, which maps two corners of a client rectangle to a world coordinate space:

```
RECT rect;
GetClientRect(WindowFromDC(hDC), & rect);
DPtoLP(hDC, (POINT *) & rect, sizeof(RECT)/sizeof(POINT));
```

For some reason, an application may want to take control of the logical to device coordinate space mapping instead of using LPtoDP and DPtoLP. For example, an application may want mapping without a device context, or it may not like how rounding is handled by GDI, or GDI's performance overhead may be too big, or it may want to take advantage of improved hardware floating-point performance. GDI does not provide a simple way to get the combined transformation between a world coordinate space and a device coordinate space. Here is a new KAffine class method that calculates the combined transformation.

```
// get the combined world to device coordinate space mapping
BOOL KAffine::GetDPtoLP(HDC hDC)
{
    if ( !GetWorldTransform(hDC, & m_xm) )
        return FALSE;

    POINT origin;
    GetWindowOrgEx(hDC, & origin);
    Translate(- (FLOAT) origin.x, - (FLOAT) origin.y);

    SIZE sizew, sizev;
    GetWindowExtEx (hDC, & sizew);
    GetViewportExtEx(hDC, & sizev);

    Scale((FLOAT) sizew.cx/sizev.cx, (FLOAT) sizew.cy/sizev.cy);

    GetViewportOrgEx(hDC, & origin);
    Translate((FLOAT) origin.x, (FLOAT) origin.y);

    return TRUE;
}
```

As the code shows, the mapping from logical to device space is the world transformation, followed by a translation to the window origin, a scale for the change in extents, and another translation to the viewport origin. The mapping from device to world space is just the inverse of the previous transformation.

Normally, a device context corresponds to a window's client area. If this is the case, you can use ClientToScreen to translate from a device coordinate space (client area) to a physical device coordinate space (screen), or use ScreenToClient for the reverse translation. Or else you can do your own addition or subtraction using a device context's origin.

GDI Implementation: Mapping and Transformation

An engineer should always be concerned about whether the system is doing the right thing in the right way with the right performance. Understanding how features are implemented helps to build confidence or to design alternatives if confidence is not high. Now we'll discuss what we know about Windows NT/2000's implementation of world transformation and window-to-viewport mapping.

Windows NT/2000 graphics engine lives in the kernel address space. Using floating point was not considered safe or it may be too slow for the older-generation processors. So the graphics engine does a simulation of floating point. Each floating-point number is converted to a FLOATOBJ structure, with a 32-bit exponent and a 32-bit signed significand. The XFORM structure is represented using a MATRIX structure, which has six FLOATOBJ fields for eM11, eM12, eM21, eM22, eDx, and eDy; two integer fields for integer portions of eDx and eDy; and a flag. Within a device context, three MATRIX structures are kept. The first one is for world-to-page mapping; clearly this is what's behind the SetWorldTransform and the GetWorldTransform API. The other two are for world-to-device mapping and device-to-world mapping. You can be assured that LPtoDP and DPtoLP goes through only a single transformation matrix. A device context also has numerous flags like the following:

`WORLD_TO_PAGE_IDENTITY,`
`PAGE_TO_DEVICE_IDENTITY,`
`PAGE_TO_DEVICE_SCALE_IDENTITY,`
`XFORM_UNITY,`
`XFORM_NO_TRANSLATION`

These flags suggest that the whole transformation or part of it may be skipped in simple cases.

For window-to-viewport mapping, the window/viewport origin and extent are stored in the original integer form in a device context. They will certainly be merged into the world-to-device and device-to-world matrix when in advanced graphics mode.

Both the window-to-viewport mapping and world transformation are kept in user address space, so that they can be accessed quickly. For point mapping like LPtoDP and DPtoLP, the simple cases are handled in user mode GDI32.DLL, while the more sophisticated cases generate system service calls that are handled by the kernel mode graphics engine.

Another thing to remember is that Windows NT/2000 graphics engine uses a 28.4 fixed-point number for a device coordinate space and a physical device space, in order to more precisely locate positions.

For details on a device context internal data structure, refer to [Chapter 3](#) of this book.

6.6 SAMPLE PROGRAM: SCROLLING AND ZOOM

Programs we have developed so far draw into the whole client area of a window and only the client area of a window. This is too restrictive even for simple tasks like text editing, not to mention word processing or CAD/CAM applications. These programs tend to draw into an imaginary canvas much bigger than a window's client area. Only a portion of the canvas is shown to the user at a time. Scroll bars are provided to show the user the relative size and position of the current screen display compared with the whole canvas. Scroll bars can also be used to change the areas to be displayed. Besides scrolling, professional applications also provide multiple levels of zoom, allowing the user to have a global view of the whole canvas or to zoom into tiny little details.

In this section we will derive a KScrollCanvas class from the KCanvas class developed in [Chapter 5](#). The KScrollCanvas class supports application-defined canvas size, scroll bars, zoom in and zoom out. It is implemented using page-coordinate-space to device-coordinate-space mapping. We will also show a simple sample program based on the new class. [Listing 6-4](#) is the class declaration.

Listing 6-4 KScrollCanvas Class Declaration: Scrolling and Zoom Support

```
class KScrollCanvas : public KCanvas
{
    virtual LRESULT WndProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam);

public:
    int m_width, m_height;
    int m_linedx, m_linedy;
    int m_zoommul, m_zoomdiv;

    virtual void OnZoom(int x, int y, int mul, int div);
    virtual void OnTimer(WPARAM wParam, LPARAM lParam);
    virtual void OnMouseMove(WPARAM wParam, LPARAM lParam);
    virtual void OnCreate(void);
    virtual void OnDestroy(void);

    KScrollCanvas(void)
    {
        m_width = 0; m_height = 0;
        m_linedx = 0; m_linedy = 0;
        m_zoommul = 1; m_zoomdiv = 1;
    }
    void SetSize(int width, int height, int linedx, int linedy)
    {
        m_width = width;
        m_height = height;
    }
}
```

```
m_linedx = linedx;
m_linedy = linedy;
}

void SetScrollBar(int side, int maxsize, int pagesize);
void OnScroll(int nBar, int nScrollCode, int nPos);
};
```

The KScrollCanvas class is derived from the KCanvas class. It overrides the WndProc message handler to take over more message handling, as can be seen from a few new virtual functions that handle window creation, destruction, mouse movement, scroll bar, and timer messages. The new member variables m_width and m_height keep the dimension of canvas size; m_linedx and m_linedy control the amount of line scrolling; and m_zoommul and m_zoomdiv are the fractional zoom ratio.

[Listing 6-5](#) shows part of the class implementation. The OnZoom method implements a simple yet effective mouse-based zoom-in/zoom-out mechanism. When you left-mouse-click on a point in a window, the window zooms in, and the point clicked becomes the center of the display if possible; when you right-mouse-click on a point, the window zooms out, and the point clicked becomes the center if possible. The actual routine is designed to be generic; it accepts four parameters, the center after the zoom and the change in zoom ratio. The code updates the zoom ratio, adds scroll bar positions to the point given, and translates it to the position after the zoom. It then updates canvas size and scroll bars; it calculates the new position for scroll bars to move the click point to the center if possible; and finally it requests a total repaint.

Listing 6-5 KScrollCanvas Implementation: Zoom and Message Handling

```
void KScrollCanvas::OnZoom(int x, int y, int mul, int div)
{
    m_zommul *= mul;
    m_zoomdiv *= div;

    int factor = gcd(m_zommul, m_zoomdiv);

    m_zommul /= factor;
    m_zoomdiv /= factor;

    // add scrollbar offset, map to position after zoom
    x = ( x + GetScrollPos(m_hWnd, SB_HORZ) ) * mul / div;
    y = ( y + GetScrollPos(m_hWnd, SB_VERT) ) * mul / div;
    // update canvas
    m_width = m_width * mul / div;
    m_height = m_height * mul / div;

    RECT rect;

    GetClientRect(m_hWnd, &rect);

    // reset scrollbars
    SetScrollBar(SB_HORZ, m_width, rect.right);
```

```
SetScrollBar(SB_VERT, m_height, rect.bottom);

// x in center the center of window if needed
x -= rect.right/2;
if ( x < 0 )
    x = 0;
if ( x > m_width - rect.right )
    x = m_width - rect.right;

SetScrollPos(m_hWnd, SB_HORZ, x, FALSE);

y -= rect.bottom/2;
if ( y < 0 ) y = 0;
if ( y > m_height - rect.bottom )
    y = m_height - rect.bottom;

SetScrollPos(m_hWnd, SB_VERT, y, FALSE);

// repaint
InvalidateRect(m_hWnd, NULL, TRUE);
::UpdateWindow(m_hWnd);
}

LRESULT KScrollCanvas::WndProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    switch( uMsg )
    {
        case WM_CREATE:
            m_hWnd = hWnd;
            OnCreate();
            return 0;

        case WM_SIZE:
            SetScrollBar(SB_VERT, m_width, HIWORD(lParam));
            SetScrollBar(SB_HORZ, m_height, LOWORD(lParam));

            return 0;

        case WM_PAINT:
        {
            PAINTSTRUCT ps;

            HDC hDC = BeginPaint(m_hWnd, &ps);

            SetWindowOrgEx(hDC, 0, 0, NULL);
            SetViewportOrgEx(hDC,
                - GetScrollPos(hWnd, SB_HORZ),
                - GetScrollPos(hWnd, SB_VERT), NULL);
        }
    }
}
```

```
    OnDraw(hDC, & ps.rcPaint);

    EndPaint(m_hWnd, &ps);
}

return 0;

case WM_RBUTTONDOWN:
OnZoom( LOWORD(lParam), HIWORD(lParam), 1, 2);
return 0;

case WM_LBUTTONDOWN:
OnZoom( LOWORD(lParam), HIWORD(lParam), 2, 1);
return 0;

case WM_HSCROLL:
OnScroll(SB_HORZ, LOWORD(wParam), HIWORD(wParam));
return 0;

case WM_VSCROLL:
OnScroll(SB_VERT, LOWORD(wParam), HIWORD(wParam));
return 0;

case WM_TIMER:
OnTimer(wParam, lParam);
return 0;

case WM_MOUSEMOVE:
OnMouseMove(wParam, lParam);
return 0;

case WM_DESTROY:
OnDestroy();
return 0;

default:
return CCanvas::WndProc(hWnd, uMsg, wParam, lParam);
}
```

The KScrollView::WndProc routine controls how messages to the window are handled. For a few messages, the processing just calls the corresponding virtual functions to give derived classes a chance to handle those messages without rewriting a message procedure. For the WM_SIZE message, scroll bar gets updated to reflect the change in window size. For the WM_PAINT message, it reads scroll bar positions and uses the values to set the viewport origin to a position off the screen on the top-left corner. Note that when the scroll bar positions are not zero, part of the virtual canvas is off the screen on the top and left sides. So the origin of viewport is moved to an off-screen position given by (-vertical scroll bar position, -horizontal scroll bar position). After the viewport origin is set properly, the OnDraw method does not need to care about scrolling. The WM_LBUTTONDOWN message zooms in the window at the mouse-click position; the WM_RBUTTONDOWN message zooms out the window at the

mouse-click position. The scrollbar generates the WM_VSCROLL and WM_HSCROLL messages, which are handled by the same routine OnScroll. OnScroll, which is not shown here, calculates the new scroll position according to the scroll code specified in LOWORD(wParam), normalizes it to be within the valid range, updates scroll bar position, and then scrolls the window by calling ScrollWindow. ScrollWindow is a Win32 function that scrolls the contents of a window by shifting the display buffer. The unpainted region uncovered by it is added to the window's update region, which will be handled by a later WM_PAINT message.

Game Board Based on KScrollView Class

As a test case for the KScrollView class, we wrote a simple program that displays a Go board. Go is the Japanese name of an exciting ancient Chinese game called WeiQi, which means “the game of enclosure.”

[Listing 6-6](#) shows the class declaration for KWeiQiBoard class. The only virtual methods implemented in this derived class are OnDraw and OnCommand, which handle problem-specific painting and main menu commands. The rest is for displaying the game board.

Listing 6-6 KWeiQiBoard Class Declaration: Game Board Based On KScrollView Class

```
class KWeiQiBoard : public KScrollView
{
    virtual void OnDraw(HDC hDC, const RECT * rcPaint);
    virtual BOOL OnCommand(WPARAM wParam, LPARAM lParam);

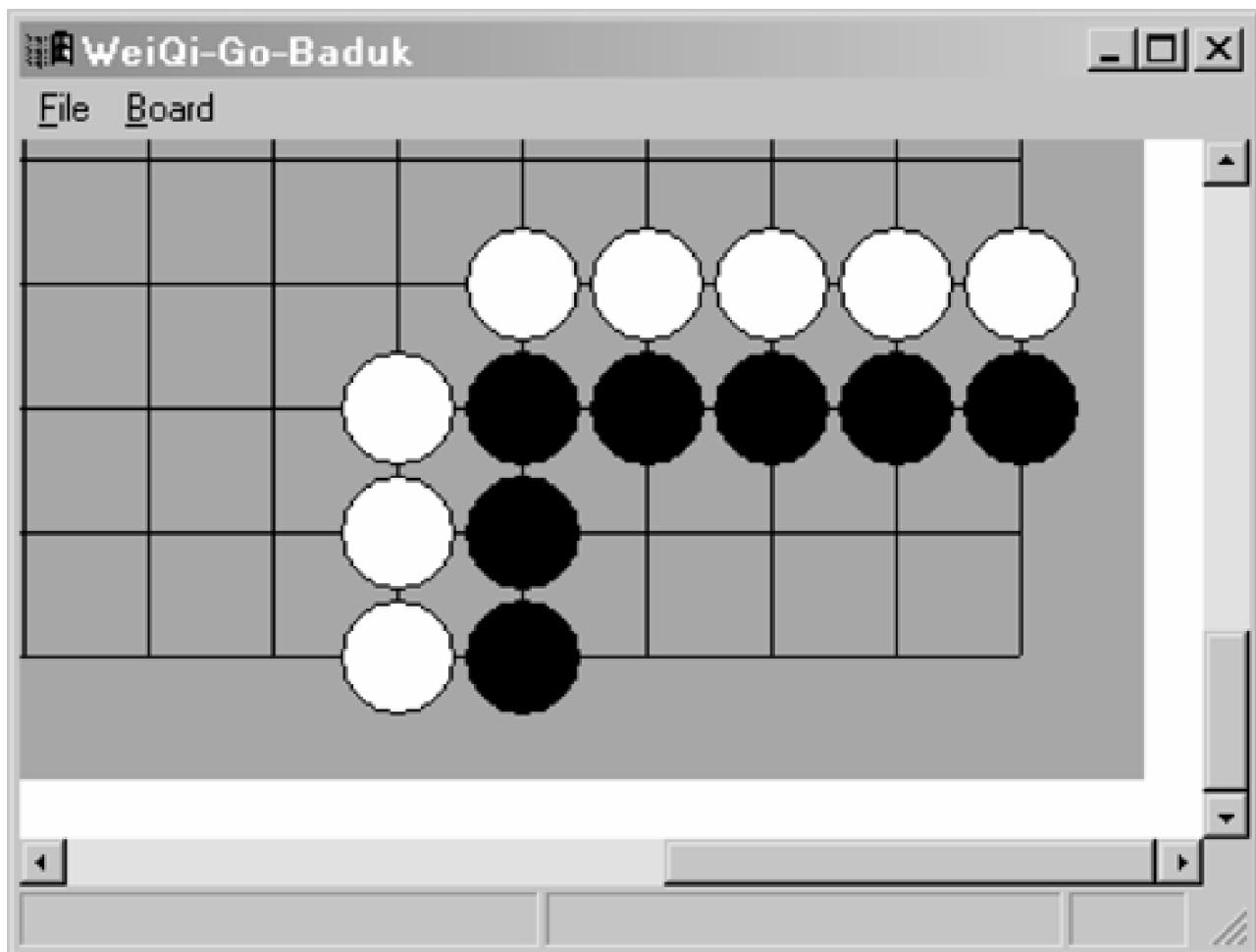
    int m_grids;
    int m_unitsize;
    char * m_stones;
    int margin(void) const; // margin on the four sides
    int pos(int n) const; // position of grid n

public:
    KWeiQiBoard()
    {
        m_grids = 19; // 19 x 19 board
        m_unitsize = 20; // 20 pixels each grid
        m_stones = "B20212223241404W3031323334251505"; // sample

        SetSize(pos(m_grids-1)+margin(), pos(m_grids-1)+margin(),
               m_unitsize, m_unitsize);
    }
};
```

[Figure 6-9](#) shows a zoomed-in, scrolled game board to examine a game situation at the bottom-right corner of the game.

Figure 6-9. A game board displayed using the KScrollView Class.



[< BACK](#) [NEXT >](#)

6.7 SUMMARY

This chapter discusses the four coordinate spaces supported by the Win32 API, namely the world coordinate space, the page coordinate space, the device coordinate space, and the physical device coordinate space. We discuss world transformation which maps coordinates in world coordinate space to page coordinate space, different mapping modes which map coordinates from page coordinate space to device coordinate space, and the relationship between device coordinate space and physical device coordinate space. A detailed description of an affine transformation is given—for example, properties of affine transformations and manipulation of affine transformations including composition and reverse.

To illustrate the knowledge covered here, KScrollView class which allows zoom-in/zoom-out and scrolling is developed. The chapter ends with a simple game board based on the KScrollView class.

Further Reading

Postscript defines two coordinate spaces: user space and device space. An affine transformation is defined to map points between the two. *Postscript Language Reference*, 3d ed., published by Adobe Systems Incorporated, is the standard reference on Postscript. *PostScript by Example*, by Henry McGilton and Mary Gampione, provides many interesting yet small Postscript code fragments which illustrate the features of Postscript. Transformations are widely used.

OpenGL, a premium render for 3D photo-realistic scenes, defines a very sophisticated coordinate system and transformations between them. *OpenGL Programming Guide*, 3d ed., by OpenGL Architecture Review Board, is the official guide to OpenGL.

If you want to know more about affine transformations and other geometrical transformation, *Geometry* by David A. Brannan et al. has very detailed descriptions. [Chapter 17](#) of this book uses mapping modes to implement device-independent, multi-column, multi-page layouts, which support different zoom ratios.

Sample Programs

This chapter has only two sample programs, as listed in [Table 6-3](#).

Table 6-3. Sample Programs for Chapter 6

Directory	Description
\Samples\Chapt_0b\CoordinateSpace	Program to explore coordinate spaces and affine transformations.
\Samples\Chapt_0b\WeiQi	Test program for the KScrollView class which supports scrolling, zoom-in, and zoom-out.

[< BACK](#) [NEXT >](#)

Chapter 7. Pixels

Once you have a device context, the hardest thing to understand in Win32 GDI is drawing a single pixel. We don't mean simply to be able to draw a single pixel, but rather to completely understand all the factors that could affect the drawing of a single pixel. After that, lines, curves, filled areas, bitmaps, and text will follow without much trouble.

[Chapter 6](#) covered coordinate systems, mapping modes, and transformations; this chapter will cover GDI objects, handles, clipping, color, and finally pixel drawing.

7.1 GDI OBJECTS, HANDLES, AND HANDLE TABLE

The Win32 API uses dozens of different kinds of objects—file objects, synchronization objects, GDI objects—although the API is not based on any modern object-oriented language. In fact, the Windows 1.0 API was released in 1985, when C++ had only 500 users, long before object-oriented languages became popular during the 1990s. An object in an object-oriented language is defined as an instance of a class, which defines the object's data structure and behavior through class member variables and class member functions. Special member functions called constructors initialize new objects for the class, which will finally be deinitialized by another special member function called a destructor. So an object, in an object-oriented language, has a data structure and a set of class member functions that operate on it. The definition of a class determines how many member variables and functions are publicly accessible, protected, or private. This serves the important purpose of information hiding.

Although the Win32 API is not an object-oriented API, it faces the same problems that object-oriented programming languages are trying to solve—namely, information hiding and abstract data type.

The Win32 API tries to hide objects more than an object-oriented language does. For a Win32 API object, the application program does not know the size of the object or where it's allocated. The Win32 API provides special functions to create objects of different kinds, which is different from the use of constructors in the C++ language. Creating an object in C++ is a two-step process: First, storage for the object needs to be allocated; second, a constructor is called to initialize the object. So a user application using the object knows exactly how much space the object occupies and where it's located. A Win32 function to create a Win32 object hides away the object's size and location. Furthermore, the application does not get back a pointer to an object; it merely gets back a handle to the object. A handle in the Win32 API is a value that can be mapped to a Win32 object in a one-to-one mapping, known only to the operating system.

Abstract data type is achieved in C++ through abstract class, concrete class, and virtual functions. An abstract class defines the generic behavior of a class through virtual functions. Several concrete classes could implement the same abstract class by providing implementation of its virtual functions. Objects of the concrete classes can be treated as objects of the abstract class, with function calls dispatched through a virtual function table.

Win32 uses quite a few abstract object types, if you look carefully. The file object type is an abstract object type, which has several concrete object types. CreateFile can be used to create files, pipes, mailslots, communication ports, consoles, directories, and devices, all returning the same type of handle. If you trace into WriteFile, you will find the final operation is handled by different routines in different parts of the OS and even in vendor-provided device drivers, much as virtual functions are used in C++. In the GDI domain, device context can be seen as an abstract object type. Creating or retrieving a printer device context, display device context, memory device context, or meta file device context—all return the same type of device context handle. But clearly, generic drawing calls through the handle are handled by different routines provided by GDI or graphics device drivers through a function pointer table in the physical device structure.

The Win32 API does have good reason to claim it's object based. Now let's take a detailed look at how Windows NT/2000 stores GDI objects, and how GDI object handles are mapped to objects. This serves as a concrete example of how the Win32 GDI could be implemented, which of course is not the only possible implementation.

GDI Object Storage

A C++ object is normally stored on a stack, a heap, or wherever defined by an application through a customized new operator. Basically, it is normally in user mode address space, unless we are talking about a kernel mode device driver.

The Windows NT/2000 graphics system is divided into three parts: the user mode client DLL GDI32.DLL, the kernel mode graphics engine WIN32K.SYS, and various device drivers which are also normally in kernel mode. The mixed user and kernel mode implementation makes GDI object storage design a little more fun.

A GDI object stored entirely in user mode address space can be accessed only when the process creating it is active, because different processes have different user mode address spaces. Once the process is switched, the address which stores the object is no longer valid. If this approach is used, GDI operations in the kernel mode graphics engine can be carried out only in the context of the calling process. This clearly interferes too much with the smooth task switching required for high-performance multitasking.

On the other hand, a GDI object stored fully in kernel mode address space can be accessed only in kernel mode address space. Simple operations like setting an attribute in a device context will need a system service call to switch the processor into kernel mode, execute a few lines of assembly code, and then switch back to user mode.

In Windows NT/2000, a GDI object normally has two portions of storage—a user mode object and a kernel mode object. The user mode object provides quick access, while the kernel mode object provides process-independent information storage. Some kernel mode objects, like the kernel mode device context, have copies of their user mode objects, which are synchronized through some mechanism.

For most GDI objects, a fixed-size data structure is kept by GDI to maintain GDI's internal representation of it. For these GDI objects, memory usage is not a big concern. For GDI region objects and device-dependent bitmap objects, the size of the data kept by GDI can grow to a significant amount. The kernel mode data structure for the GDI data structure is stored in a so-called paged pool. According to the Windows NT/2000 design, the kernel paged pool has a 192-MB size limitation on a single-user system. Given that GDI's internal memory pool is limited and shared by all applications on a system, the application should take care not to create large amounts of GDI regions and device-dependent bitmaps.

The GDI Object Table

GDI objects are stored in a systemwide fixed-size object table, or call it an object handle table. We have several things to explain here. First, the GDI object table is a fixed-size table, not a dynamically growing array as you might have expected. This means simplicity and efficiency, but also limitation. As of now, Windows NT/2000 allows 16,384 GDI handles. Second, the GDI object table is shared by all the processes and system DLLs in the system. Your desktop, browser, word processor, and DirectX game all compete for the same pool of GDI handles. In Windows 2000, DirectX uses a separate object handle table.

The GDI handle table is stored in kernel mode address space so that the graphics engine can easily access it. Through some magic, a read-only view of the table is created in user mode address space for every process that uses GDI.

NOTE

On a terminal server, each session has its own copy of the Windows graphics engine and the window manager (WIN32K.SYS), so there are multiple GDI object tables on the system.

Each entry in the GDI handle table is a 16-byte structure, as shown here:

```
typedef struct
{
    void *      pKernel;
    unsigned short nPid;
    unsigned short nCount;
    unsigned short nUnique;
    unsigned short nType;
    void *      pUser;
} GdiTableEntry;
```

So a GDI object has a pointer to its kernel mode object, a pointer to its user mode object, a process id, a count of some kind, a uniqueness value, and a type identifier. What a nice, elegant design!

The nPid field stores the process identifier of the process creating the object [value returned by `GetCurrentProcessId()`]. Every GDI object created by a user process is tagged with the process's id, to prevent the cross-process usage of GDI handles. When you pass a GDI object handle to GDI, it can easily validate whether the object is created by the current process. Using a GDI object created by a foreign process results in failure, except for stock objects. Stock objects, like black pen, white brush, and system font, are GDI objects that are preselected into a new device context. It would be wasteful for each process to create its own copy of all the stock objects. Instead, stock objects are created once in the system and have 0 as their process id. Your task manager will tell you that zero happens to be the process id for the system idle process. GDI object handles with zero as their Pid can be used in all processes.

The nCount could serve as some sort of select count, or reference count, to prevent the deletion of objects in use, or to assure that certain objects can be selected only once. Unfortunately, the nCount field is used only in a limited way. The user application still needs to take care of when an object can be deleted.

The uUnique field may be the more interesting field here. GDI objects are stored in the same table. It could happen that an object is created, used, and then deleted, so the slot occupied by it is freed. Later on, another object is created in the same slot. Now some buggy code still holds a handle to the first object and dares to use it; what's going to happen? On Windows NT/2000, the buggy call is almost guaranteed to fail, because of this uniqueness field. The nUnique field is divided into an 8-bit recycle count and an 8-bit type code. The recycle count starts with zero, incremented every time a GDI object is created in this slot. There is another scheme in the GDI handle manager that ensures that slots in the table are used evenly. So the old handle can pass the uniqueness check only if the number of objects created on the same slot is a multiple of 256 and if the object types are the same.

The nType field is the internal type GDI keeps for each GDI object, which can be translated to the GDI object type returned by `GetObjectType`.

The GDI Object Handle

My dictionary defines a handle to be the part of a door, window, etc., by which it can be opened. MSDN defines a handle to be a variable that identifies an object, an indirect reference to an operating-system resource. A

programmer nowadays should be able to live with this kind of abstract definition and write wonderful programs with it. But someone chasing a thunking or subclassing defect between 16-bit and 32-bit code on Windows NT/2000, or someone writing a low-level tool, would prefer a more concrete breakdown of every bit of every handle in the Win32 API.

A GDI object handle in Windows NT/2000 is made up of two 16-bit values, its uniqueness value and its index to the GDI handle table.

What's the significance of this design? Easy mapping from handle to GDI object, excellent protection, and limited GDI handles.

To map a GDI handle to an entry in the GDI object table, just mask out the lower 16 bits of the handle, multiply it by 16, and add the starting address of the GDI object table. The GDI object table starting address is kept in global variables in both WIN32K.SYS and GDI32.DLL.

For protection, we've talked about the effectiveness of the uniqueness value. The uniqueness value is also part of a GDI object handle. The uniqueness value in a GDI handle must match the uniqueness value stored in a GDI object table entry before GDI allows accessing of the handle. Once GDI gets the GDI object table entry, the process identifier stored in it provides a double check. An invalid handle, dead handle, or a handle from a foreign process can be easily singled out.

In the current implementation, the number of GDI object handles on a system (or per session on a Windows terminal server) is limited to 16,384 objects. This can easily be expanded to 65,536 handles, because within a GDI handle, the index part takes 16 bits instead of 12.

User applications should not be designed to rely on knowledge of the GDI handle format and how the GDI object table is organized. This changes from one operating system to another and may change again in future releases. But Windows NT/2000 does provide more protections and restrictions on GDI handles, which also allows better debugging of code. Make sure that you don't share GDI handles across a process boundary and that you test your application on all current operating systems.

For more information on the internal GDI data structure behind the GDI handles, refer to [Chapter 3](#) of this book.

The GDI Object API

GDI objects are divided into several major categories. Logical brushes, logical pens, logical fonts, logical palettes, regions, device-dependent bitmaps, DIB sections, enhanced metafiles, and device contexts are common types of GDI objects. Each normally has several dedicated routines to create a new GDI object of that type. Once a GDI object is created, GDI returns a GDI object handle to the application. A number of GDI functions operate on generic GDI object handles.

```
HGDIOBJ SelectObject(HDC hDC, HGDIOBJ hgdiobj);
BOOL DeleteObject(HGDIOBJ hObject);
DWORD GetObjectType(HGDIOBJ h);
int GetObject(HGDIOBJ hgdiObj, int cbBuffer, LPVOID lpvObject);
```

Several types of GDI objects can be attributes of a device context: brushes, pens, fonts, palettes, device-dependent bitmaps, and DIB sections. SelectObject is used to attach one of them to a device context object. The function

returns the previous GDI handle of the same type as a return value. To deselect a GDI object from a device context, call SelectObject again with the value returned from the first selection. The logical palette has a special routine for selecting it into a device context, SelectPalette.

When a GDI object is no longer required, it needs to be deleted using Delete Object. Before deleting a GDI object, the application needs to make sure it's not selected into any device context.

Windows 95/98/Me and Windows NT/2000 have slightly different implementations of the GDI object deletion. On Windows 95/98/Me, DeleteObject does not delete an object if it's still selected into a device context. This can cause potential GDI object leakage. On Windows NT/2000, DeleteObject deletes a GDI object even if it is still selected into a device context. Further drawing using the already deleted GDI object will fail, which makes it easier for programmers to notice the problem.

If a program is developed and tests fine on a Windows NT/2000 machine, but fails on a Windows 95/98/Me machine, one possible reason is that it has an improper GDI object deletion problem. The class KGDIObject is a simple wrapper around GDI object selection, deselection, and deletion.

```
class KGDIObject
{
    HGDIOBJ m_hOld;
    HDC    m_hDC;

public:
    HGDIOBJ m_hObj;

    KGDIObject(HDC hDC, HGDIOBJ hObj)
    {
        m_hDC = hDC;
        m_hObj = hObj;
        m_hOld = SelectObject(hDC, hObj);

        assert(m_hDC);
        assert(m_hObj);
        assert(m_hOld);
    }

    ~KGDIObject()
    {
        HGDIOBJ h = SelectObject(m_hDC, m_hOld);
        assert(h==j);

        DeleteObject(m_hObj);
        // assert(GetObjectType(m_hObj)==0);
    }
};
```

The KGDIObject has three member variables: for the GDI object handle to be selected, for a device context handle,

and for a handle to the original object in that device context. Its constructor handles the selection of a GDI object into a device context. Three assertions are added to make sure that the parameters are right and that selection succeeds. The destructor of the class deselects the GDI object and then deletes it. The first assertion in it makes sure that deselection occurs as expected. The second assertion makes sure the GDI object is really deleted. The second assertion is commented out, because GDI sometimes caches delete the objects to speed up the creation of new objects of the same types.

NOTE

MFC wraps GDI's SelectObject function to return a pointer to an instance of MFC's GDI object class CGdi Object. But the result could be a temporary pointer if MFC is unable to map a GDI object handle to an MFC object pointer. If an application is relying on the temporary pointer to deselect an object when it's not used, it could already be changed by another SelectObject call—a possible cause for a GDI object leak.

Here is a simple example of using the KGDIObject class:

```
void OnDraw(HDC hDC)
{
    KGDIObject blue(hDC, CreateSolidBrush(RGB(0, 0, 0xFF));
    Rectangle(hDC, 0, 0, 100, 100);
}
```

As a last line of defense for a healthy system, when a process finishes, all GDI objects created by that process are automatically deleted, apparently using the process identifier attached to each GDI object. Still, applications should be very careful in deleting GDI objects properly, instead of relying on the system to clean up the mess when processes terminate. Constant GDI resource leakage could quickly stop the whole system from functioning properly.

Given a GDI object handle, GetObjectType returns an integer number representing the type of GDI object it is. For example, OBJ_DC will be returned for a device context object, OBJ_BRUSH will be returned for a logical brush object, OBJ_BITMAP will be returned for either a device-dependent bitmap or a DIB section. If zero is returned, the handle passed in is either an invalid GDI object handle, or a handle belonging to another process.

GetObject can be used to query for the original structure used in creating a GDI object for several types of GDI objects. It fills in a BITMAP structure for a device-dependent bitmap, a DIBSECTION structure for a DIB section, an EXTLOGOPEN structure for an extended pen, a LOGPEN structure for a pen, a LOGBRUSH structure for a brush, a LOGFONT structure for a font, and a WORD for a palette. To call GetObject, pass a GDI object handle, the expected size of the structure, and a pointer to a data block large enough to hold the structure. If the caller is not sure about the number of bytes needed to hold the structure, passing a NULL pointer to Get Object lets it return the number of bytes needed.

GetObject can be used to distinguish between device-dependent bitmaps and DIB sections, which make no difference to GetObjectType. Just call GetObject with a DIBSECTION structure; if the call succeeds, the GDI object is a DIB section.

Details of each type of GDI object will be discussed as their services are needed.

GDI Object Leakage Detection

When symptoms of GDI object leakage appear—for example, when painting can't be completed or the system user interface display is not drawn properly—it's quite hard to find which application is causing the problem, or exactly what is leaked, and where the leak occurs.

The only powerful tool to pinpoint GDI object leakage and other types of memory leakage and resource leakage seems to be Numega's BoundsChecker. Bounds Checker works by overwriting almost all Win32 API functions, recording, validating, and analyzing parameters and function return results. For example, if all GDI object-creation calls and object-deletion calls are logged, a tool like BoundsChecker can compare object creation and object deletion before process termination and find all GDI object leakage together with the caller's exact location. [Chapter 4](#) of this book describes several tools for spying on GDI API calls, system service calls, and DDI calls. It's possible to extend these tools to detect GDI object leakage.

Based on the understanding of GDI internal data structure, it's not hard to write a program to monitor the usage of GDI objects for all the processes running on a Windows NT/2000 system. Such a monitoring program can indicate GDI resources in the system, although it does not have enough information to pinpoint the exact location of leaking. [Figure 7-1](#) illustrates the “GDIObj” program developed for this chapter. The program enumerates objects in the GDI object table on a periodic basis, sorts them according to their process identifiers and object types, and displays in a list view.

Figure 7-1. GDI objects monitoring program.

PID	Process	Total	DC	Region	Bitmap	Palette	Font	Brush	Other
0	unknown	707	10	38	357	175	16	45	66
912	OSA.EXE	25	3	0	5	0	0	17	0
18	unknown	5	5	0	0	0	0	0	0
212	services.exe	4	2	0	1	0	0	1	0
164	unknown	22	3	4	1	0	12	2	0
184	winlogon.exe	14	2	1	4	1	5	1	0
1948	MSDEV.EXE	466	48	5	269	5	73	66	0
2336	IEXPLORE.EXE	256	25	8	160	3	28	32	0
224	lsass.exe	4	2	0	1	0	0	1	0
392	svchost.exe	4	2	0	1	0	0	1	0
440	spoolsv.exe	8	2	1	1	0	0	4	0
480	svchost.exe	4	2	0	1	0	0	1	0
536	MSTask.exe	4	2	0	1	0	0	1	0
852	Explorer.exe	126	20	23	33	5	20	25	0
944	internat.exe	15	6	1	5	0	0	3	0
908	tgsched.exe	4	2	0	1	0	0	1	0
928	zm32nt.exe	5	2	0	1	0	1	1	0

As shown in [Figure 7-1](#), MS Developer Studio is using 466 GDI objects, 2.8% of the total available GDI object handles. If the number of GDI handles used by an application keeps increasing over time, it's a clear sign of GDI object leakage. For a monitored process, "GDIObj" displays the total number of GDI objects and breaks it down into several major categories. If there is a leak, it gives you an indication of the types of GDI objects actually being leaked.

[Figure 7-1](#) shows that the first process (with PID 0) has 66 unclassified objects. These objects are mainly physical font objects used by GDI. It also shows that a Win32 process with a graphics user interface uses at least four GDI objects: two device contexts, one bitmap, and one brush. These objects may be created during the initialization of either the USER32.DLL or GDI32.DLL. This implementation limits the number of processes you can run on a single system to 4,096, because there are only 16,384 handles available.

Windows 2000 provides a new function for applications to query GDI and USER resource usage.

```
DWORD GetGuiResources(HANDLE hProcess, DWORD uiFlags);
```

Function GetGuiResources can be used to query the number of GDI objects or USER objects used by an application. The first parameter to GetGuiResources is the process handle for the process you're interested in. The second parameter can either be GR_GDIOBJECTS or GR_USEROBJECTS. The function returns the current number of objects used.

Function GetGuiResources offer an official way to tell if a piece of code has possible resource leakage. For example, you can call it before and after a drawing routine and compare the function return values to see if it creates any new objects. Although the difference in the return results does not always mean resource leakage, because of possible object caching by application and GDI, continuous increase in object usage is a clear indication of object leakage.

Here is a simple wrapper class and its sample usage in handling the WM_PAINT message:

```
class KGUIResource
{
    int m_gdi, m_user;

public:
    KGUIResource()
    {
        m_gdi = GetGuiResources(GetCurrentProcess(), GR_GDIOBJECTS);
        m_user = GetGuiResources(GetCurrentProcess(), GR_USEROBJECTS);
    }

    ~KGUIResource()
    {
        int gdi = GetGuiResources(GetCurrentProcess(), GR_GDIOBJECTS);
        int user = GetGuiResources(GetCurrentProcess(), GR_USEROBJECTS);

        if ( (m_gdi==gdi) && (m_user==user) )
```

```
return;

char temp[64];
wsprintf(temp, "ResourceDifference: gdi(%d->%d) user(%d->%d)\n",
    m_gdi, gdi, m_user, user);
OutputDebugString(temp);
}

};

case WM_PAINT:
{
    KGUIResource res;

    PAINTSTRUCT ps;
    HDC hDC = BeginPaint(m_hWnd, &ps);

    OnDraw(hDC, &ps.rcPaint);

    EndPaint(m_hWnd, &ps);
}
```

[< BACK](#) [NEXT >](#)

7.2 CLIPPING

In ordinary English, clipping means cutting something with a tool to make it shorter, or trimming something. In computer graphics, clipping is part of a drawing algorithm to selectively cut, ignore, or disable certain parts of a graphics drawing. For example, when you are reading a document in your word processor in a high zoom ratio, only part of the page is shown in the window; the rest is clipped. Or if you use a photographic application to put an oval-shaped frame on a picture, what's outside of the oval shape is clipped.

Traditional computer graphics defines clipping in logical address space. For example, the term "window" in computer graphics really means a window, in the sense that only drawing primitives falling within the window get displayed; anything outside gets clipped. Special algorithms need to be designed to find the intersections of drawing primitives and the window so as to draw only the nonclipped parts.

Windows GDI supports clipping as one of its basic features. For example, an application is free to draw a line from the origin to the pseudoinfinite point allowed in 32-bit coordinate space (maxint, maxint). The line will stop as soon as it reaches the edge of the window's client area or device surface. This design simplifies algorithm design tremendously.

Clipping Pipeline

Conceptually, there are several layers of clipping for every drawing call, which forms a clipping pipeline. The documentation provided by Microsoft is quite vague and sometimes confusing on this topic; so are quite a few books based on this information. Here is an example which you may have read: "If you obtain a device context handle from the BeginPaint function, the DC contains a predefined rectangular clipping region that corresponds to the invalid rectangle that requires repainting." There are three mistakes in this description.

Here is our understanding of the layers of clipping as supported in Microsoft Windows:

- **Window rectangle.** Each window has a rectangular box, which is defined initially in the CreateWindow/CreateWindowEx call and can later be resized. Anything outside of this window rectangle is clipped.
- **Window region.** An application can change the window's region to an arbitrary region using SetWindowRegion. Anything outside the window region is clipped.
- **Visibility.** Windows may overlap each other; a window may have child windows or sibling windows. Any part that is obscured by other windows is clipped. Areas covered by child windows and sibling windows may be clipped, too, according to the WS_CLIPCHILDREN and WS_CLIPSIBLINGS flags.
- **Client area.** If a device context corresponds to a window's client area, anything outside the client area is clipped.
- **Update region.** Win32 provides API to mark the region that needs repainting, which is called `update region` for a window. An update region can be queried using GetUpdateRegion. For a device context returned by BeginPaint, anything outside the update region is clipped.

- **System region.** The combined region generated considering the above factors is called the *system region*. A device context's system region can be queried using GetRandomRgn(hDC, hRgn, SYSRGN).
- **Meta region.** A meta region is the first level of clipping controlled by the application for a device context. Anything outside a meta region is clipped.
- **Clip region.** A clip region is the second level of clipping controlled by the application for a device context. Anything outside a clip region is clipped.

Now let's rephrase the BeginPaint clipping region description. If you obtain a device context handle from the BeginPaint function, the DC contains a predefined system region that may not be rectangular; it is generated from the window's update region considering other factors. The device context's meta and clip regions always start as NULL regions; they allow the application to control the overall clipping.

The system region is controlled by the operating system. It gets updated automatically when a window resizes, or moves. The meta and clip regions are controlled by the application. A pixel is drawn only if it is within the intersection of the system region, meta region, and clip region. In a certain sense, the term *clip* in *clip region* is misleading, because the region determines the points to retain, not those to clip away. Anything outside gets clipped.

Simple Regions

A region is a set of points in a coordinate space. The set could be empty, could contain points forming a rectangle, a circle, or an arbitrary shape, or it could cover the entire coordinate surface. Win32 provides a rich set of APIs that deal with a region as a type of GDI object. We will cover some simple stuff here to allow us to proceed, leaving the more complicated features until later in the book.

In Win32, a region is an object managed by GDI, just like a device context, logical pen, logical brush, etc. When an application calls a region-creation function, a region object is created, initialized by the right data, and then has its handle returned to the application. A region handle, of type HRGN, is later passed to GDI to use the region. The easiest way to create a region is:

```
HRGN CreateRectRgn(int nLeftRect, int nTopRect,  
                    int nRightRect, int nBottomRect);
```

This creates a rectangular region that contains all the points within the rectangle from (nLeftRect, nTopRect) to (nRightRect, nBottomRect), except those on the bottom and right edges. Note that the parameters do not need to be well formed; GDI will reorder them to form a well-formed rectangle. Here are a few examples:

```
HRGN hRgn1 = CreateRectRgn(0, 0, 0, 0);  
HRGN hRgn2 = CreateRectRgn(0, 0, 1, 1);  
HRGN hRgn3 = CreateRectRgn(-0x7FFFFFFF, -0x7FFFFFFF,  
                           0x7FFFFFFF, 0x7FFFFFFF);  
HRGN hRgn4 = CreateRectRgn(1, 1, 0, 0);
```

The first region is empty, the second contains only a single point (0,0), the third may be the largest region supported by Windows NT/2000's 32-bit GDI implementation, and the last is the same as the second region.

The right/bottom-edge exclusion is normally quite confusing. When you create a rectangle region with {0, 0, 1, 1}, GDI stores the region object using a single rectangle {0, 0, 1, 1}, not {0, 0, 0, 0}. The right/bottom edge is effective only during the final stage of drawing calls, or during some query operations. This makes a difference in understanding how operations on regions work. For example, if you scale a region represented using {0, 0, 1, 1} by a factor of n , it becomes {0, 0, n , n } , containing n points. But if you scale a region represented using {0, 0, 0, 0} , would it still be {0, 0, 0, 0} or {0, 0, $n - 1$, $n - 1$ }?

When a region object is no longer needed, call DeleteObject to release the resource associated with it.

Clipping Region

A device context has a clipping-region attribute, which stores an application-defined region object that determines when the region drawing is allowed to proceed. For a device context returned by BeginPaint, GetDC, or CreateDC, the clipping region is NULL. Having NULL as a clipping region is also referred to as having no clipping region.

A NULL clipping region is not the same as an empty region in the set-theoretic sense; it's just the opposite. An empty clipping region—for example, CreateRect Rgn (0, 0, 0, 0)—means that nothing should be displayed; everything should be clipped. A NULL clipping region means that everything (in the system region) should be displayed; nothing is clipped. So an empty clipping region is an empty set of points, while a NULL clipping region can be considered as the complete set of points on the device surface.

The following are the basic functions for dealing with a device context's clipping region:

```
int GetClipRgn(HDC hDC, HRGN hrgn);
int SelectClipRgn(HDC hDC, HRGN hrgn);
int ExtSelectClipRgn(HDC hDC, HRGN hrgn, int fnMode);
int OffsetClipRgn(HDC hDC, int nXOffset, int nYOffset);
int ExcludeClipRect(HDC hDC, int nLeftRect, int nTopRect,
                    int nRightRect, int nBottomRect);
int IntersectClipRect(HDC hDC, int nLeftRect, int nTopRect,
                     int nRightRect, int nBottomRect);
int GetClipBox(HDC hDC, LPRECT lprc);
```

Quite a few clipping-region functions return the complexity of the clipping region as an integer value. Other functions like GetClipRgn return nonzero for success, and zero for failure. Here are the possible complexity values:

NULLREGION Region is empty; that is, the region is an empty set.

SIMPLEREGION The region is made up of a single rectangle.

COMPLEXREGION The region is more than one rectangle.

ERROR An error occurred (previous region unaffected).

NULREGION means the region is empty; for instance, the intersection of two disjoint regions returns the NULREGION flag. Again, this is different from having NULL, or no region. SIMPLEREGION means that the set of points within a region forms a rectangular shape. COMPLEXREGION is for any valid region which can't be represented using a simple rectangle. ERROR normally means that one of the parameters to a GDI function call is invalid or NULL.

If you're familiar with how a pen, a brush, or a font GDI object interacts with a device context, you may find the clipping region API quite strange and hard to understand. You can use GetCurrentObject to query for the current pen, brush, or font associated with a device context, and use SelectObject to replace it with a new GDI object. Once a GDI object is selected into a device context, it's considered to be in use and can't be deleted until it's deselected. The interaction between a region object and a device context is based more on the data defining the region, instead of the region handle. To be more specific, you can't query for the current clipping-region handle for a device context; and after you select a clipping region into a device context, the handle is no longer used by that device context.

To query a device context's current clipping region, an application needs to create a valid region object and pass its handle to GetClipRgn function as its hrgn parameter. So now hrgn references a valid yet dummy region object. If the device context has a clipping region, GetClipRgn frees the region object referenced by hrgn, makes a copy of the device context's clipping region, and causes hrgn to reference the new clipping-region copy. The result returned by GetClipRgn is a flag that says whether the region is an empty region, a single-rectangle region, or a complex region. If the device context does not have a clipping region, a value of 0 (ERROR) is returned, and hrgn is not modified.

For the corresponding call SelectClipRgn, the hrgn could be a valid region object handle. In this case, GDI makes a copy of the region data and associates it with the device context. Note that the handle is not kept by the device context; an application is free to delete it after the call. The parameter could also be NULL, which frees the clipping-region object kept in the device context, and sets the device context to the NULL clipping region.

In some sense, the device context's attributes, such as logical pen, are handle-based attributes. Once a GDI object handle is selected into a device context, it's considered used by the device context and can't be deleted until another object handle of the same type is selected. But a clipping region is a data-based attribute. When a region handle is passed to SelectClipRgn, a copy is made of the region data, not the handle. When an application wants to retrieve the current clipping region, a valid handle must be given, and its corresponding region data will be replaced.

GetClipRgn returns 0 if the device context has no clipping region; the region parameter is not changed in this case. Actually, a fresh device context returned by Begin Paint or CreateDC does not have a clipping region, so GetClipRgn always returns 0. Again, having no clipping region means everything in the system region area gets displayed, instead of nothing getting displayed. An application should always check for GetClipRgn's return result and act properly, instead of relying on the region object handle after the call. For example, the code below checks whether GetClipRgn returns 0; if so, it deletes the region object and sets its handle to NULL. From this NULL handle, the code after that can tell whether the device context has a clipping region.

```
HRGN hRgn = CreateRectRgn(0, 0, 1, 1);
if ( GetClipRgn(hDC, hRgn)==0 )
{
    DeleteObject(hRgn);
    hRgn = NULL;
}
...
...
```

As we have mentioned, a region is a set of points. So it's easy to define operations on regions modeled after set operations. [Table 7-1](#) summarizes the available setlike region operations supported by GDI.

Table 7-1. Binary Region Operations

Operation Mode	Result
RGN_AND	A region representing the overlapping area of region1 and region2.
RGN_COPY	A copy of region1.
RGN_DIFF	A region representing the area covered by region1, but not in region2.
RGN_OR	A region representing the area covered by either region1 or region2.
RGN_XOR	A region representing the area covered by either region1 or region2, but not both.

When a region object is selected as a clipping region or a meta region in a device context, it's supposed to be in the device coordinate space of that device context. This is inconsistent with most of the coordinates passed to the GDI functions, which are normally in the device context's logical coordinate space. The GDI function LPtoDP can be used to convert coordinates from the logical coordinate space to the device coordinate space.

It also helps us to understand empty regions and complete regions. An empty region does not contain any point, so things like CreateRect Rgn(0, 0, 0, 0) are empty regions. A complete set should contain every point in the coordinate space. But if you pass the minimum and maximum 32-bit integer number to CreateRectRgn, it will fail. The maximum region you can create seems to be defined by

```
{-0x7FFFFFFF, -0x7FFFFFFF, 0x7FFFFFFF, 0x7FFFFFFF}
```

This is certainly related to the 28.4 fixed-point number used by the graphics engine for the device coordinate space, which has 28-bit signed-integer parts. But for practical reasons, the pixel dimension of a physical device could be used to generate a "complete" region.

ExtSelectClipRegion offers more control over how the region hrgn should be combined with the existing clipping region in a device context to form the new clipping region, using set operations shown in [Table 7-1](#). Here the first operand is the device context's current clipping region; the second operand is specified by parameter hrgn. For ExtSelectClipRegion, RGN_COPY makes a copy of hrgn to be the current clipping region, the same as SelectClipRegion.

We mentioned that a fresh device context returned by BeginPaint or GetDC does not have a clipping region; then how does ExtSelectClipRegion work? No clipping region is implicitly or explicitly considered a full clipping region. For ExtSelectClip Region on a device context with no clipping region, RGN_AND is the same as RGN_COPY. For RGN_DIFF, RGN_OR, or RGN_XOR, GDI uses the physical device size to create a complete region. For example, let's look at the following code fragment:

```
PAINTSTRUCT ps;
HDC hDC = BeginPaint(hWnd, & ps);
HRGN hRgn = CreateRectRgn(0, 0, 200, 200);
int rslt = ExtSelectClip(hDC, hRgn, RGN_DIFF);
rslt = GetClipRgn(hDC, hRgn);
...

```

The code creates a 200-by-200 rectangle region and tries to change the current clipping region using the RGN_DIFF operation. The device context does not have a clipping region, so the graphics engine uses a rectangle

region defined by {0, 0, 1152, 864} in 1152-by-864 screen mode, and GetClipRgn returns a region with two rectangles {200, 0, 1152, 200} and {0, 200, 1152, 864}.

The next three clipping-region functions are simple. For a device context having a clipping region, OffsetClipRgn translates all the points in the region object by (nXOffset, nYOffset). ExcludeClipRect removes a rectangle from the current clip region, similar to ExtSelectClip with RGN_DIFF. IntersectClipRect intersects the current clip region with a rectangle, similar to ExtSelectClip with RGN_AND. ExcludeClipRect and InsertsectClipRect are easier ways to update the clipping region compared with ExtSelectClip.

GetClipBox returns the smallest bounding box of the intersection of the current system region and clipping region—not any one of them, but their intersection. In the default situation, there is no clipping region, so GetClipBox returns the bounding box of the system region, which is the same as the rcPaint rectangle in PAINT STRUCT filled by BeginPaint. As the clipping region gets built up by SelectClipRgn and ExtSelectClipRgn, the GDI engine maintains its intersection with the system region and its bounding box, which is the result returned by GetClipBox. Within the WM_PAINT message processing, if an application changes the clipping region, the GetClipBox result is more accurate than the rcPaint rectangle.

The Meta Region

The meta region is a feature barely documented in GDI. The meta region is not mentioned as an attribute of a device context. Its default value is not mentioned, nor is its relationship with the system region and the clipping region. We base our discussion here on our investigation.

A meta region is a meta clipping region, or the first-level clipping region, which controls clipping at another level. Now let's rephrase the GDI clipping rule: The effective display region of a device context is the intersection of its system region, meta region, and clipping region; anything outside is clipped.

If an application finds that one layer of an application-defined clipping is too inconvenient, here comes another layer, just as GDI has two layers of logical coordinate spaces. If you find that one layer of an application-defined clipping is good enough, you can forget about the meta region.

Like the clipping region, a fresh device context does not have a meta region. Even when a clipping region is selected, a meta region does not exist until SetMetaRgn is called. Here is the meta region API:

```
int SetMetaRgn(HDC hDC);
int GetMetaRgn(HDC hDC, HRGN hRgn);
```

Unlike any other functions prefixed by “set”, SetMetaRgn accepts only a single parameter: a device context handle. It uses an implicit parameter, the current clipping region. SetMetaRgn replaces the current meta region and its intersection with the current clipping region and resets the device context to have no clipping region. Note that the meta region clips drawing primitives the same way as a clipping region, so immediately after SetMetaRgn, the overall clipping of the device context is not affected functionally. But now that an application moves the old clipping region to the meta clipping layer, it's free to rebuild a new clipping region that will also restrict what's going to be displayed.

GetMetaRgn is nothing unusual; it returns the current meta-region data through an existing region handle, similar to GetClipRgn. A fresh device context does not have a meta region, for which GetMetaRgn returns 0 without updating hRgn.

It's interesting to notice that if an application calls SetMetaRgn multiple times, the device context's meta region can get only smaller, never bigger, and there is no way to reset the meta region to a NULL region once it's set. One workaround is using SaveDC and RestoreDC.

Here is an example of using meta region. The code sets both the meta region and the clipping region to be 100-by-100 rectangles, but offset by (50, 50) so the only drawable area is 50 by 50.

```
HRGN hRgn = CreateRectRgn(0, 0, 100, 100);
SelectClipRgn(hDC, hRgn); // meta: no, clip: 100x100
SetMetaRgn(hDC); // meta: 100x100, clip no
SelectClipRgn(hDC, hRgn); // meta: 100x100, clip: same
OffsetClipRgn(hDC, 50, 50); // meta: 100x100, clip: 100x100
Rectangle(0, 0, 150, 150); // draws a 50x50 rectangle
DeleteObject(hRgn);
```

By now, you may have concluded that meta region is not an essential feature. If an application controls all the drawing in a single layer, the meta region or something more sophisticated can easily be simulated. But if the drawing in an application is implemented in a layered fashion, the meta region allows the clipping to be controlled at two places. For example, if an application is using a third-party DLL to draw a picture, and the drawing code itself uses the GDI clipping region, it can't change the third-party source code to restrict the drawing to a certain area, but it can set up a meta region to achieve the same effect.

A lesser known feature like the meta region in the Win32 API always exists for a practical purpose. The meta region is used by GDI in playing back meta files.

Five Regions in a Device Context

So we have a system region, meta region, and clipping region in the device context. Their intersection is the effective region where a display is allowed. The system region is controlled by the window manager; the meta region and clipping region are controlled by the user application. If for every drawing call GDI needed to calculate their intersection and pass it to the device driver, it would be very inefficient. Guess what—a device context keeps two more regions to improve performance: API region and Rao region.

API region is the intersection of the meta region and the clipping region, clearly named after the fact that it is controlled by GDI API calls. The Rao region is the intersection of the API region and the system region, named after the Microsoft engineer who initially proposed it. When either the meta region or the clipping region changes, the API region and the Rao region are recalculated.

The regions in a device context are stored as pointers to the region kernel objects instead of the GDI region object handles. This explains why SetClipRgn makes a copy of the region instead of using a GDI handle, and GetClipRgn requires a valid region handle as a parameter.

Among the five regions, the system region, meta region, clipping region, and API region are accessible through a single API GetRandomRgn. Have you ever wonder why a function could be named GetRandomRgn? What's the use of a region that is randomly defined? GetRandomRgn gives random access, instead of sequential access, to the four regions stored in a device context. The last parameter of GetRandomRgn is an integer index. Its only documented value is SYSRGN (4). Here are the actual acceptable values:

```
#define CLIPRGN 1 // GetClipRgn
#define METARGN 2 // GetMetaRgn
#define APIRGN 3
#define SYSRGN 4
```

Using GetRandomRgn, an application can query for a device context's clip region, meta region, API region, and system region. The Rao region can be calculated from the API region and the system region.

Visualizing Device Context Regions

In [Section 5.5 of Chapter 5](#) we developed a program to visualize a window paint message and a device context's system region. If you don't think those for-dummy programs are also useful for professionals, we are now going to write a program, "ClipRegion," to allow us to visualize the clipping, meta, and API regions, together with the system region.

First we need a routine to retrieve the four regions and dump some information about them into a text window. Here is the DumpRegions routine:

```
void KMyCanvas::DumpRegions(HDC hDC)
{
    for (int i=1; i<=4; i++) // clip,meta,api,system
    {
        m_bValid[i] = false;
        int rslt = GetRandomRgn(hDC, m_hRandomRgn[i], i);

        switch ( rslt )
        {
            case 1:
                m_bValid[i] = true;
                m_Log.DumpRegion("RandomRgn(%d)", m_hRandomRgn[i],
                    false, i);
                break;

            case -1:
                m_Log.Log("RandomRgn(%d) Error\r\n", i);
                break;

            case 0:
                m_Log.Log("RandomRgn(%d) no region\r\n", i);
                break;

            default:
                m_Log.Log("Unexpected\r\n");
        }
    }
}
```

}

The routine uses GetRandomRgn to access the device context's clipping, meta, API, and system regions, and checks its return value. Each region is dumped into a text window using the KLogWindow class. The region handles are stored in class member variables for later use by the DrawRegions routine.

```
void KMyCanvas::DrawRegions(HDC hDC)
{
    HBRUSH hBrush;

    SetBkMode(hDC, TRANSPARENT);

    if ( m_bValid[1] ) // clipping region
    {
        hBrush = CreateHatchBrush(HS_VERTICAL, RGB(0xFF, 0, 0));
        FillRgn(hDC, m_hRandomRgn[1], hBrush);
        DeleteObject(hBrush);
    }

    if ( m_bValid[2] ) // meta region
    {
        hBrush = CreateHatchBrush(HS_HORIZONTAL, RGB(0, 0xFF, 0));
        FillRgn(hDC, m_hRandomRgn[2], hBrush);
        DeleteObject(hBrush);
    }
}
```

The DrawRegions routine is called after EndPaint returns, using a new device context returned by GetDC, so that it can paint to the whole client area instead of just within the system region. It uses different hatch brushes to draw the stored clipping region and meta region. The API region of a device context should be just their intersection. We know that there are no default clipping, meta, and API regions, so to see anything meaningful, we need to set up our experiments using the TestClipMeta routine shown below. The “ClipRegion” program defines four tests, which can be selected from the main menu. The first test does not set up the clipping and meta regions, so you can see what the default situation is. The second test sets up a clipping region, the third sets up a meta region, and the last sets up both a clipping region and meta region. The clipping region used is an elliptic region covering the left three-quarters of the client area; the meta region is an elliptic region covering the top three-quarters of the client area.

```
void TestClipMeta(HDC hDC, const RECT & rect)
{
    HRGN hRgn;
    switch ( m_test )
    {
        case IDM_TEST_DEFAULT:
            break;

        case IDM_TEST_SETCLIP:
```

```
hRgn = CreateEllipticRgn(0, 0, rect.right*3/4,
                         rect.bottom);
SelectClipRgn(hDC, hRgn);
DeleteObject(hRgn);
break;

case IDM_TEST_SETMETA:
hRgn = CreateEllipticRgn(0, 0, rect.right,
                         rect.bottom*3/4);
SelectClipRgn(hDC, hRgn);
SetMetaRgn(hDC);
break;

case IDM_TEST_SETMETACLIP:
hRgn = CreateEllipticRgn(0, 0, rect.right,
                         rect.bottom*3/4);
SelectClipRgn(hDC, hRgn);
SetMetaRgn(hDC);
DeleteObject(hRgn);
hRgn = CreateEllipticRgn(0, 0, rect.right*3/4,
                         rect.bottom);
SelectClipRgn(hDC, hRgn);
break;
}

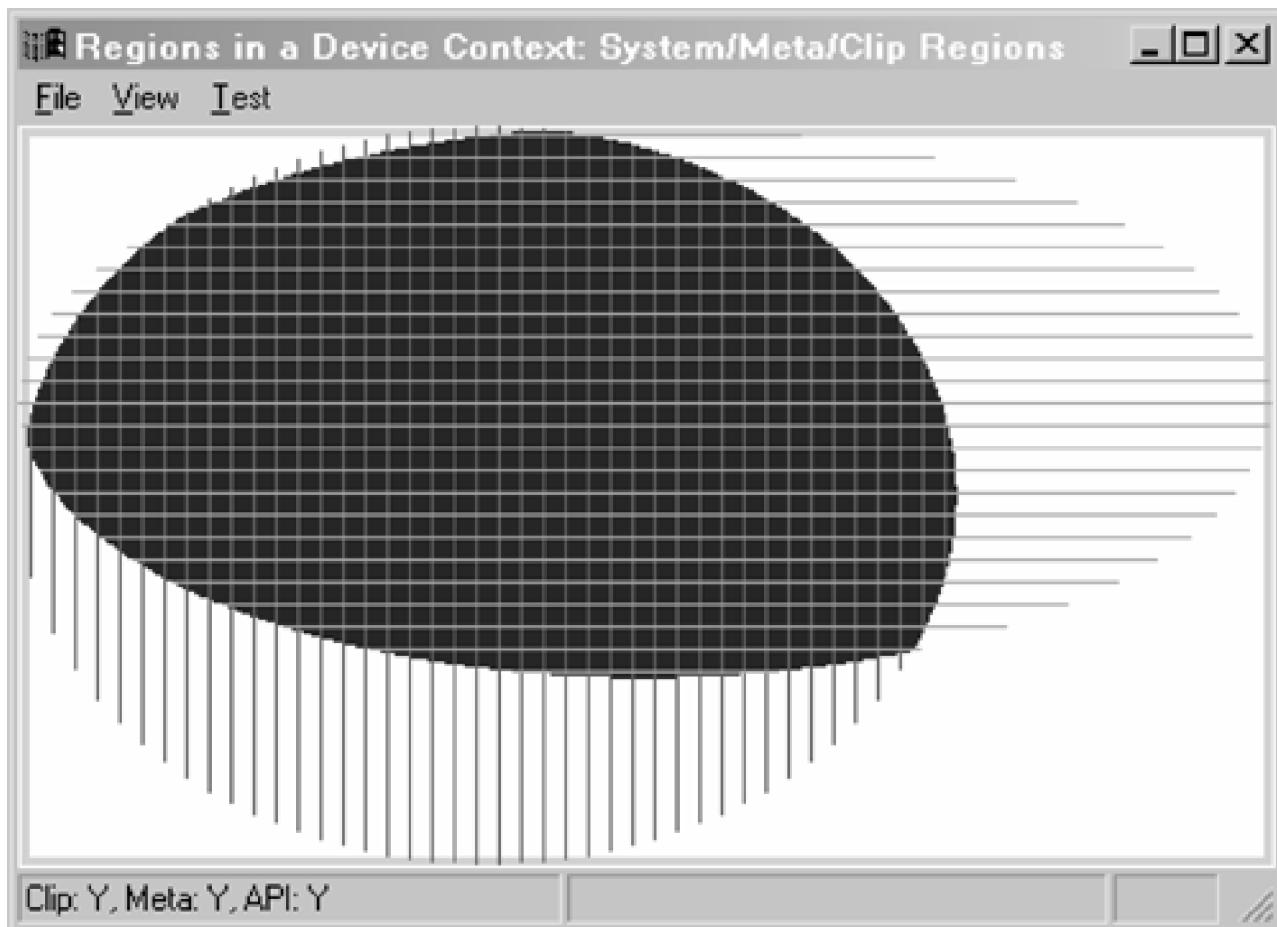
DeleteObject(hRgn);

// with meta and clip region selected, only the intersection
// of system region, meta region, clip region can be painted
HBRUSH hBrush = CreateSolidBrush(RGB(0, 0, 0xFF));
FillRect(hDC, & rect, hBrush);
DeleteObject(hBrush);

DumpRegions(hDC);
}
```

The routine `TestClipMeta` is called by the main drawing routine `KMyCanvas::OnDraw`, after the system region is displayed. So after the clipping and meta regions are set up, all three regions are effective. The code then tries to fill the whole client area using a solid blue brush. If we're right, only the intersection of the system region, meta region, and clipping region should be painted; the rest should be clipped. [Figure 7-2](#) shows the display on screen when both clipping and meta regions are set.

Figure 7-2. System, clipping, meta, API, and Rao regions.



The framed rectangle represents the system region, vertical lines cover the clipping region, horizontal lines cover the meta region, the API region has both lines, and the solid area is their intersection, the drawable area.

Another thing interesting to do is go through the four test cases and look at what's logged in our logging window. Here is an example:

```
// IDM_TEST_DEFAULT
RandomRgn(1) no region
RandomRgn(2) no region
RandomRgn(3) no region
RandomRgn(4) SIMPLEREGION RgnBox=[464, 247, 922, 590] 1 rects

// IDM_TEST_SETCLIP
RandomRgn(1) COMPLEXREGION RgnBox=[0, 0, 342, 342] 201 rects
RandomRgn(2) no region
RandomRgn(3) COMPLEXREGION RgnBox=[0, 0, 342, 342] 201 rects
RandomRgn(4) SIMPLEREGION RgnBox=[464, 247, 922, 590] 1 rects

// IDM_TESTMETA
RandomRgn(1) no region
RandomRgn(2) COMPLEXREGION RgnBox=[0, 0, 457, 256] 189 rects
RandomRgn(3) COMPLEXREGION RgnBox=[0, 0, 457, 256] 189 rects
RandomRgn(4) SIMPLEREGION RgnBox=[464, 247, 922, 590] 1 rects
```

```
// IDM_TESTMETACLIP
RandomRgn(1) COMPLEXREGION RgnBox=[0, 0, 342, 342) 201 rects
RandomRgn(2) COMPLEXREGION RgnBox=[0, 0, 457, 256) 189 rects
RandomRgn(3) COMPLEXREGION RgnBox=[2, 2, 342, 256) 191 rects
RandomRgn(4) SIMPLEREGION RgnBox=[464, 247, 922, 590) 1 rects
```

The log shows the default values for those regions: the system region (RandomRgn[4]) is independent of clipping; the clipping region and the meta region intersect to form the API region (RandomRgn[3]). A more interesting system region can be generated by putting a small window on top of ClipRegion's main window, then moving it away; a WM_PAINT message will be generated to repaint the uncovered area, which may have a complex system region.

[< BACK](#) [NEXT >](#)

7.3 COLOR

Color is our eye's perception of light reflected by objects, a very complex subject. To use color in computer graphics programming, we need a way to describe color numerically that can be easily implemented on common computer hardware. We also need a way to relate these color descriptions to our normal way of describing color, which people can easily understand.

Color description is normally done using several attributes of color within a certain range. These attributes can be seen as coordinates in a coordinate space, where each individual color is a point within the space. Such a coordinate space for color description is called a color space.

Computer graphics uses dozens of different color spaces. Display monitors normally use RGB color space, which employs red, green, and blue as the three primary colors. Color printers tends to use CMYK color space, where every color is a combination of its cyan, magenta, yellow, and black components. Graphics artists prefer to describe color using its hue, saturation, and luminance. Now let's look at several commonly used color spaces.

The RGB Color Space

The Windows graphics system normally uses the RGB color space to specify colors. The RGB color space has three axes: red, green and blue. Each color is a point in this three-dimensional space, represented as a triplet (red, green, and blue).

Computer graphics books and algorithms prefer to normalize each component to be a floating-point number between 0 and 1, which is easy to manipulate mathematically. But a graphics programming interface like GDI needs to be practical and efficient, so each component needs to be represented using scalar values suitable for computer storage and manipulation. The Windows API chooses to use a single byte value to represent 256 possible levels (0 to 255) of each component. So three bytes, or 24 bits, are needed to represent a color in the Windows RGB color space, which gives 2^{24} , or 16.7 million possible colors.

The RGB color space is considered an additive color space. The color represented by the origin (0, 0, 0) is black; adding full red turns it to red (255, 0, 0), adding full green turns it to yellow (255, 255, 0), and finally adding full blue changes it to white (255, 255, 255).

GDI provides several macros to combine the three RGB components into a single 32-bit value of COLORREF type and to take apart a COLORREF data into its RGB components. We can think of them as in-line functions defined below:

```
COLORREF RGB(BYTE byRed, BYTE byGreen, BYTE byBlue);
BYTE GetRValue(COLORREF rgb);
BYTE GetGValue(COLORREF rgb);
BYTE GetBValue(COLORREF rgb);
```

Here are handy definitions for some commonly used colors:

```
const COLORREF black      = RGB( 0,  0,  0);
const COLORREF darkred    = RGB(0x80,  0,  0);
const COLORREF darkgreen   = RGB( 0, 0x80,  0);
const COLORREF darkyellow  = RGB(0x80, 0x80,  0);
const COLORREF darkblue    = RGB( 0,  0, 0x80);
const COLORREF darkmagenta = RGB(0x80,  0, 0x80);
const COLORREF darkcyan    = RGB( 0, 0x80, 0x80);
const COLORREF darygray   = RGB(0x80, 0x80, 0x80);

const COLORREF moneygreen = RGB(0xC0, 0xDC, 0xC0);
const COLORREF skyblue    = RGB(0xA6, 0xCA, 0xF0);
const COLORREF cream      = RGB(0xFF, 0xFB, 0xF0);
const COLORREF mediumgray = RGB(0xA0, 0xA0, 0xA4);
const COLORREF lightgray   = RGB(0xC0, 0xC0, 0xC0);

const COLORREF red       = RGB(0xFF,  0,  0);
const COLORREF green     = RGB( 0, 0xFF,  0);
const COLORREF yellow    = RGB( 0, 0xFF,  0);
const COLORREF blue      = RGB( 0,  0, 0xFF);
const COLORREF magenta   = RGB(0xFF,  0, 0xFF);
const COLORREF cyan      = RGB( 0, 0xFF, 0xFF);
const COLORREF white     = RGB(0xFF, 0xFF, 0xFF);
```

GDI API uses RGB values directly in a device-independent manner. Applications normally do not need to worry about how color is actually converted before it's stored in a video display card's memory buffer, which is handled by the display drivers. In palette-based display modes, user applications do have control over how the system palette should be allocated to improve the display. We will cover the system palette together with bitmaps later in the book.

The easiest way to have some fun with colors is by using the SetPixel function to draw one pixel at a time, which is defined as:

```
COLORREF SetPixel(hDC hDC, int x, int y, COLORREF crColor);
```

SetPixel draws a single pixel of color crColor at location (x, y) on to a surface, subject to logical coordinate space setup and clipping. Drawing a single pixel does not make a statement, so we decide to draw the RGB color cube—that is, the cube formed by 256 levels of red, green, and blue components in a 3D space. [Listing 7-1](#) shows a RGBCube routine, which draws a 3D cube.

Listing 7-1 Drawing a 3D RGB Cube without Transformation

```
void RGBCube(HDC hDC)
{
    int r, g, b;

    // draw and label axes
```

```
// Red = 0xFF
for (g=0; g<256; g++)
for (b=0; b<256; b++)
    SetPixel(hDC, g, b, RGB(0xFF, g, b));

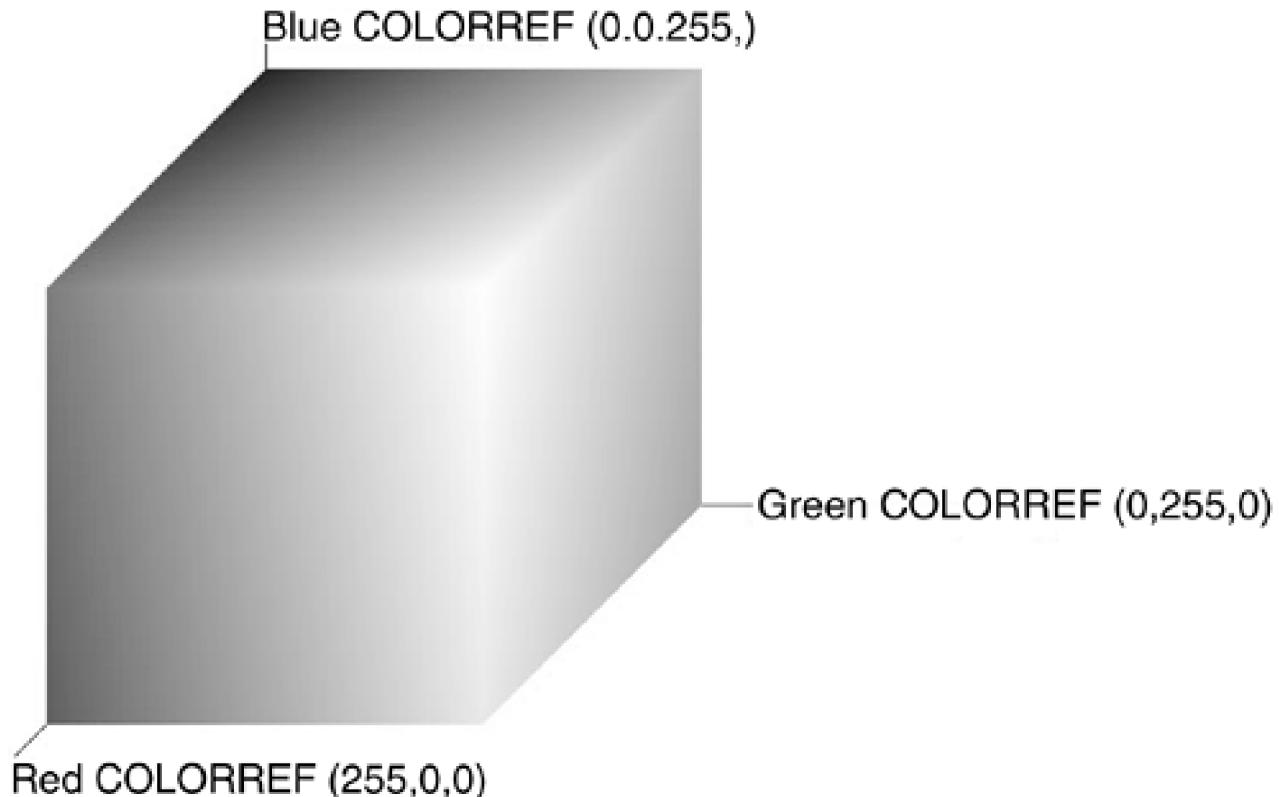
// Blue = 0xFF, top, sheared
for (g=0; g<256; g++)
for (r=0; r<256; r+=2)
    SetPixel(hDC, g+128-r/2, 255+128-r/2, RGB(r, g, 0xFF));

// Green = 0xFF, right, sheared
for (b=0; b<256; b++)
for (r=0; r<256; r+=2)
    SetPixel(hDC, 255+128-r/2, b+128-r/2, RGB(r, 0xFF, b));
}
```

The code draws three faces of a 3D cube; they have red = 255, blue = 255, and green = 255. The first face is drawn as a rectangle; the next two are sheared to show the 3D effect. We would have used world transformation to achieve the shear mapping, but it is not difficult to do that ourselves by removing half of the scan lines and shifting the pixels a little bit. In the color cube displayed, we can see eight corners of the cube, but not the black origin.

With some extra code to set up the proper window to viewport mapping and axes drawing, [Figure 7-3](#) shows the final result (fine print: you have to run the program to see real color).

Figure 7-3. RGB color space.



Too bad—our nice color cube is shown in grayscale in the book, not color. This brings up another topic: how do you convert RGB color into grayscale? Here is a simple formula:

$$\text{Grayscale} = (\text{Red} \cdot 30 + \text{Green} \cdot 59 + \text{Blue} \cdot 11 + 50) / 100, \text{ or}$$
$$\text{Grayscale} = (\text{Red} \cdot 77 + \text{Green} \cdot 151 + \text{Blue} \cdot 28 + 128) / 256$$

Each of the color components contributes differently to the grayscale level, as represented by the factors. Pure blue is quite dark, so its factor is the smallest, followed by red and green. Adding 50 or 128 is for rounding up. If you think your compiler and CPU are not so good at division, you can use the second formula to help the compiler use a shift operation. Modern compilers do quite amazing optimization. Do not be surprised if you can't find a multiplication or division by constant instructions in your binary; they may be replaced by faster instructions. By the same token, if you try to write some assembly code for optimization, don't write code that multiplies by constants like 3 (for 24-bit address calculation). The compiler can do a better job using address-calculation instructions, which handle multiplication by a fixed small number using additions.

The HLS Color Space

While the RGB color space is easier to store and manipulate for computer graphics programming, it does not relate well to our perception of color. The HLS color space uses hue (H), lightness (L), and saturation (S) to describe colors, which is much more intuitive.

The HLS color space can be seen as a rotated RGB color cube. Let's rotate the RGB color cube in 3D space to have the white corner at the top and the black corner at the bottom. If you look along the white (255, 255, 255) to black (0, 0, 0) line, you will find six corners: red, yellow, green, cyan, blue, and magenta; all the other colors lie among them. The hue component in HLS space is measured in angles from 0 up to 360 degrees: degree 0 is red, 60 is yellow, 120 is green, 180 is cyan, 240 is blue, 300 is magenta, and 360 goes back to red. The hue is a measure of a color's angle to the RGB cube's red corner when you look along the white-to-black diagonal line. Lightness is a measure of the height in a rotated color cube where 1 means white and 0 means black. Saturation is a measure of a color's distance from the white-to-black diagonal line.

[Listing 7-1](#) shows a routine to draw a 3D RGB cube without using GDI's world transformation. [Listing 7-2](#) shows how to use world transformation to map each of the three faces into one-third of an HLS hexagon. The code also serves as a world transformation sample.

Listing 7-2 Displaying the Top View of a Rotated RGB Cube

```
void ColorRect(HDC hDC, COLORREF c0, COLORREF dx, COLORREF dy)
{
    for (int x=0; x<256; x++)
        for (int y=0; y<256; y++)
            SetPixel(hDC, x, y, c0 + x * dx + y * dy);

    MoveToEx(hDC, 0, 0, NULL);
    LineTo(hDC, 0, 255); LineTo(hDC, 255, 255);
    MoveToEx(hDC, 0, 0, NULL);
    LineTo(hDC, 255, 0); LineTo(hDC, 255, 255);
```

```
//LineTo(hDC, 0, 0);
}

void RGBCube2HLSHexagon(HDC hDC)
{
    KAffine affine;
    SetGraphicsMode(hDC, GM_ADVANCED);

    FLOAT r = 254;           // even number < 255
    FLOAT x = r / 2;         // cos(60);
    FLOAT y = (FLOAT) (r * 1.732/2); // sin(60);

    // set center of hexagon, allow some margins
    SetViewportOrgEx(hDC, (int)r + 40, (int)y + 40, NULL);

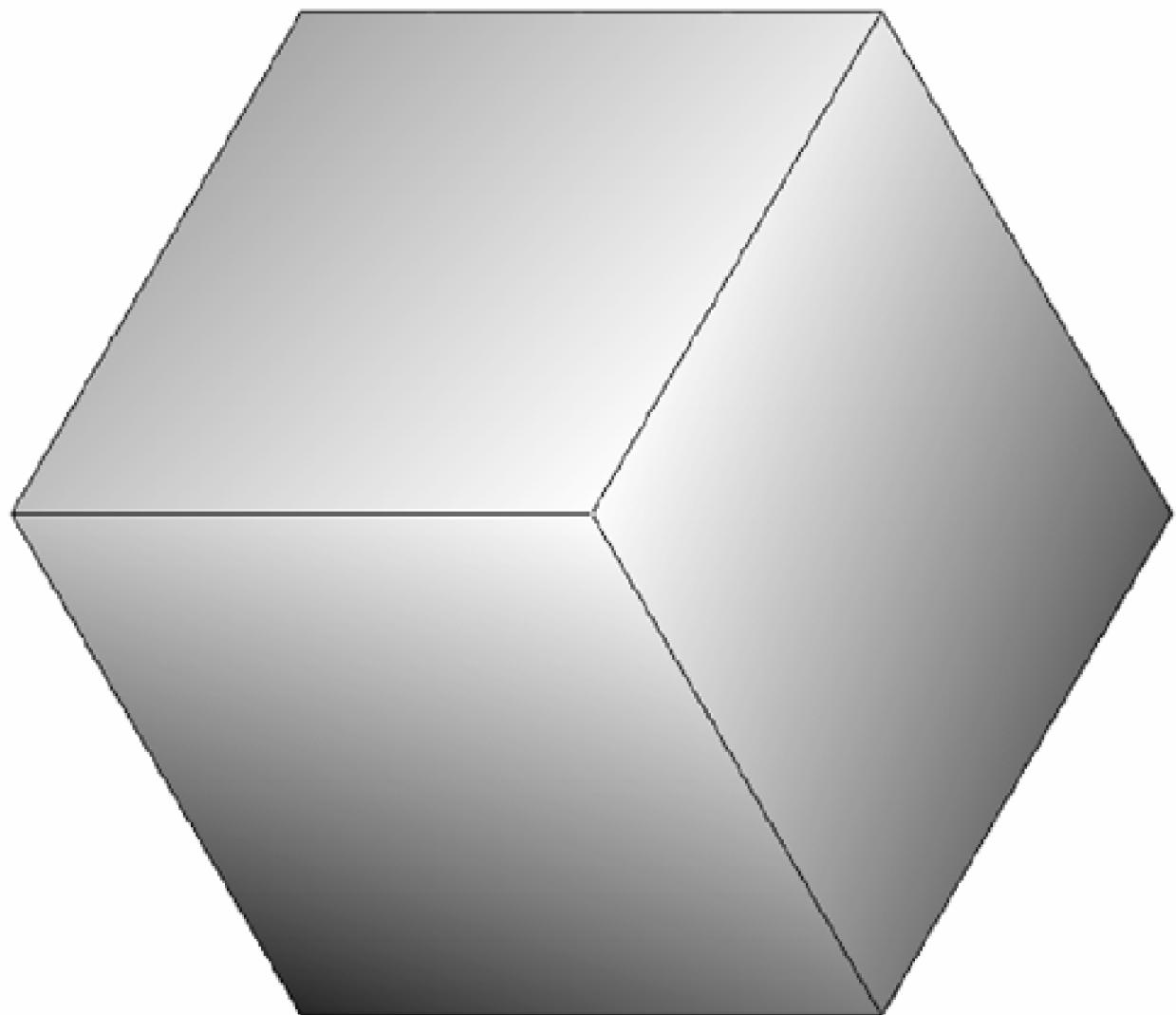
    // Red = 255
    affine.MapTri(0,0, 255,0, 0,255, r,0, x,y, x,-y);
    SetWorldTransform(hDC, & affine.m_xm);
    ColorRect(hDC, RGB(0xFF, 0, 0), RGB(0, 1, 0), RGB(0, 0, 1));

    // Green = 255
    affine.MapTri(0,0, 255,0, 0,255, -x,y, x,y, -r,0);
    SetWorldTransform(hDC, & affine.m_xm);
    ColorRect(hDC, RGB(0, 0xFF, 0), RGB(1, 0, 0), RGB(0, 0, 1));

    // Blue = 255
    affine.MapTri(0,0, 255,0, 0,255, -x,-y, -r,0, x,-y);
    SetWorldTransform(hDC, & affine.m_xm);
    ColorRect(hDC, RGB(0, 0, 0xFF), RGB(0, 1, 0), RGB(1, 0, 0));
}
```

The code draws each of the three faces as a rectangle in world coordinate space, mapping them to parallelograms to form a hexagon. The KAffine class is used to calculate the transformation from world coordinate space to page coordinate space, using three-point mapping. [Figure 7-4](#) illustrates the HLS color space.

Figure 7-4. An RGB cube as seen from the white-to-black diagonal line.



The HLS color space forms a double cone, which is often pictured as a double hexagon. The center of the cone corresponds to a lightness of 0.5, the top tip of the cone to a lightness of 1 (white), and the bottom of the cone to a lightness of 0 (black). The hue component is an angular measurement with red, yellow, green, cyan, blue, and magenta at 0, 60, 120, 180, 240, and 300 degrees. The saturation component describes color intensity; maximum intensity 1 is at the edge of a hexagon, and no color intensity 0 is in the center of hexagon. The lightness component describes the brightness or luminance of color; 0 represents black, and 1 represents white.

In the HLS color space, hue is normally represented using a real number in the range [0..360); both lightness and saturation are in the range [0..1]. [Listing 7-3](#) shows a C++ class for conversion between RGB and HLS space.

Listing 7-3 KColor Class: RGB, HLS Color-Space Conversion

```
class KColor
{
    typedef enum { Red, Green, Blue };
public:
    unsigned char red, green, blue;
    double      lightness, saturation, hue;

    void ToHLS(void);
}
```

```
void ToRGB(void);
};

void KColor::ToHLS(void)
{
    double mn, mx;
    int     major;

    if ( red < green )
    {
        mn = red; mx = green; major = Green;
    }
    else
    {
        mn = green; mx = red; major = Red;
    }

    if ( blue < mn )
        mn = blue;
    else if ( blue > mx )
    {
        mx = blue; major = Blue;
    }

    if ( mn==mx )
    {
        lightness  = mn/255;
        saturation = 0;
        hue        = 0;
    }
    else
    {
        lightness = (mn+mx) / 510;

        if ( lightness <= 0.5 )
            saturation = (mx-mn) / (mn+mx);
        else
            saturation = (mx-mn) / (510-mn-mx);
        switch ( major )
        {
            case Red : hue = (green-blue)*60 / (mx-mn)+360;
                        break;
            case Green: hue = (blue-red)*60 / (mx-mn)+120;
                        break;
            case Blue : hue = (red-green)*60 / (mx-mn)+240;
        }

        if (hue > 360)
            hue = hue - 360;
    }
}
```

```
}

unsigned char Value(double m1, double m2, double h)
{
    if (h > 360)
        h -= 360;
    else if (h < 0)
        h += 360;

    if (h < 60)
        m1 = m1 + (m2 - m1) * h / 60;
    else if (h < 180)
        m1 = m2;
    else if (h < 240)
        m1 = m1 + (m2 - m1) * (240 - h) / 60;

    return (unsigned char)(m1 * 255);
}

void KColor::ToRGB(void)
{
    if (saturation == 0)
    {
        red = green = blue = (unsigned char) (lightness*255);
    }
    else
    {
        double m1, m2;

        if ( lightness <= 0.5 )
            m2 = lightness + lightness * saturation;
        else
            m2 = lightness + saturation - lightness * saturation;

        m1 = 2 * lightness - m2;

        red = Value(m1, m2, hue + 120);
        green = Value(m1, m2, hue);
        blue = Value(m1, m2, hue - 120);
    }
}
```

The HLS color space is commonly used for color selection. For example, the color-selection common dialog box provided by the Windows OS uses the HLS color space to guide color selection. It displays the hue-saturation plane as a rectangle. The user can select a color by moving the mouse within the rectangular area, then adjust the color's luminance in a separate color scrollbar. [Listing 7-4](#) shows how we can use the color conversion provided by the KColor class to simulate the HLS color palette displayed by Windows' color-selection common dialog box. [Figure 7-5](#) shows the display result.

Figure 7-5. Color palette using HLS color space.



Listing 7-4 HLS Color Palette Display

```
void HLSColorPalette(HDC hDC, int scale, CColor & selection)
{
    KColor c;

    for (int hue=0; hue<360; hue++)
        for (int sat=0; sat<=scale; sat++)
    {
        c.hue      = hue;
        c.lightness = 0.5 ;
        c.saturation = ((double) sat)/scale;
        c.ToRGB();

        SetPixel(hDC, hue, sat, RGB(c.red, c.green, c.blue));
    }

    for (int l=0; l<=scale; l++)

```

```
{  
    c = selection;  
    c.lightness = ((double)l)/scale;  
    c.ToRGB();  
  
    for (int x=0; x<64; x++)  
        SetPixel(hDC, scale+20+x, l,  
            RGB(c.red, c.green, c.blue));  
}  
}
```

The first part of the routine demonstrates how color in the HLS color space can be converted to the RGB color space and displayed on screen. Three-dimensional graphics requires simulating objects under different lights, varying the object's luminance according to the distance and angle between the object and the lighting source. The second part of the routine shows how this can be done in the HLS color space by changing a color's lightness without changing its hue and saturation. This technique is also very useful in gradient fill implementations when simple mixing of RGB colors is not adequate.

Indexed Color and Palette

Besides specifying color using RGB values, the Win32 API also supports referencing color using an index into a palette, which is a table of colors specified using RGB values.

Each device context has a logical palette attribute associated with it. A logical palette is a GDI-managed object, referenced by its handle, which is of type HPALETTE. Here are a few simple palette-related functions:

```
HPALETTE CreateHalftonePalette(HDC hDC);  
COLORREF GetNearestColor(HDC hDC, COLORREF crColor);  
UINT GetNearestPaletteIndex(HPALETTE hpal, COLORREF crColor);  
UINT GetPaletteEntries(HPALETTE hPal, UINT iStartIndex,  
    UINT nEntries, LPPALETTEENTRY lppe);  
UINT RealizePalette(HDC hDC);  
HAPLETTE SelectPalette(HDC hDC, HPALETTE hPal,  
    BOOL bForceBackground);
```

CreateHalftonePalette creates a palette with 256 colors evenly distributed in the RGB cube. It's an adequate palette for displaying color-heavy graphics using a halftone technique in a palette-based system mode. GetNearestColor searches the device context's current logical palette for the closest color matching. GetNearestPaletteIndex searches a logical palette for the closest color matching. GetPaletteEntries retrieves the color-table definition from a logical palette. RealizePalette prepares the system palette to display colors in a logical palette. SelectPalette attaches a logical palette to a device context, which returns the original logical palette. You can also retrieve the device context's current palette handle using GetCurrentObject(hDC, OBJ_PAL).

The following macros are provided for specifying palette-related colors:

```
COLORREF PALETTEINDEX(WORD wPaletteIndex);
```

COLORREF PALETTERGB(BYTE bRed, BYTE bGreen, BYTE bBlue);

PALETTEINDEX accepts a palette index and returns a 32-bit palette-entry specifier. When its return value is used in a device context, it's interpreted as the specified entry within the device context's logical palette. PALETTERGB accepts red, green, and blue values just like the RGB macro. Their return values have the same meaning if used on a device context with no system (hardware) palette. But if a PALETTERGB result is used on a device with a system palette, the RGB value is used to find the closest matching entry in the device context's logical palette as though the application had specified a palette index.

So, given a device context, PALETTERGB can be implemented as:

```
COLORREF PALETTERGB(HDC hDC, BYTE bRed, BYTE bGreen, BYTE bBlue)
{
    COLORREF rslt = RGB(bRed, bGreen, bBlue);

    if ( GetDeviceCaps(hDC, RASTERCAPS) & RC_PALETTE )
    {
        HPALETTE hPal = GetCurrentObject(hDC, OBJ_PAL);
        Int     indx = GetNearestPaletteIndex(hPal, rslt);
        return PALETTEINDEX(indx);
    }
    else
        return rslt;
}
```

We mentioned that every device context has a preset logical palette, even in high-color or true-color display modes. Among the functions mentioned so far, only RealizedPalette really needs the presence of a hardware palette to succeed; other functions can be used freely in non-palette-based display modes. [Listing 7-5](#) illustrates how PALETTEINDEX can be used to display every color within a logical palette.

Listing 7-5 Display Colors in a Logical Palette

```
void ShowLogicalPalette(HDC hDC, bool bHalftone)
{
    HPALETTE hPalette = (HPALETTE) GetCurrentObject(hDC, OBJ_PAL);

    if ( bHalftone )
        hPalette = CreateHalftonePalette(hDC);

    PALETTEENTRY entry[256];

    int num = GetPaletteEntries(hPalette, 0, 256, entry);

    HAPLETTE hOld = SelectPalette(hDC, hPalette, FALSE);

    if ( GetDeviceCaps(hDC, RASTERCAPS) & RC_PALETTE )
        RealizePalette(hDC);
```

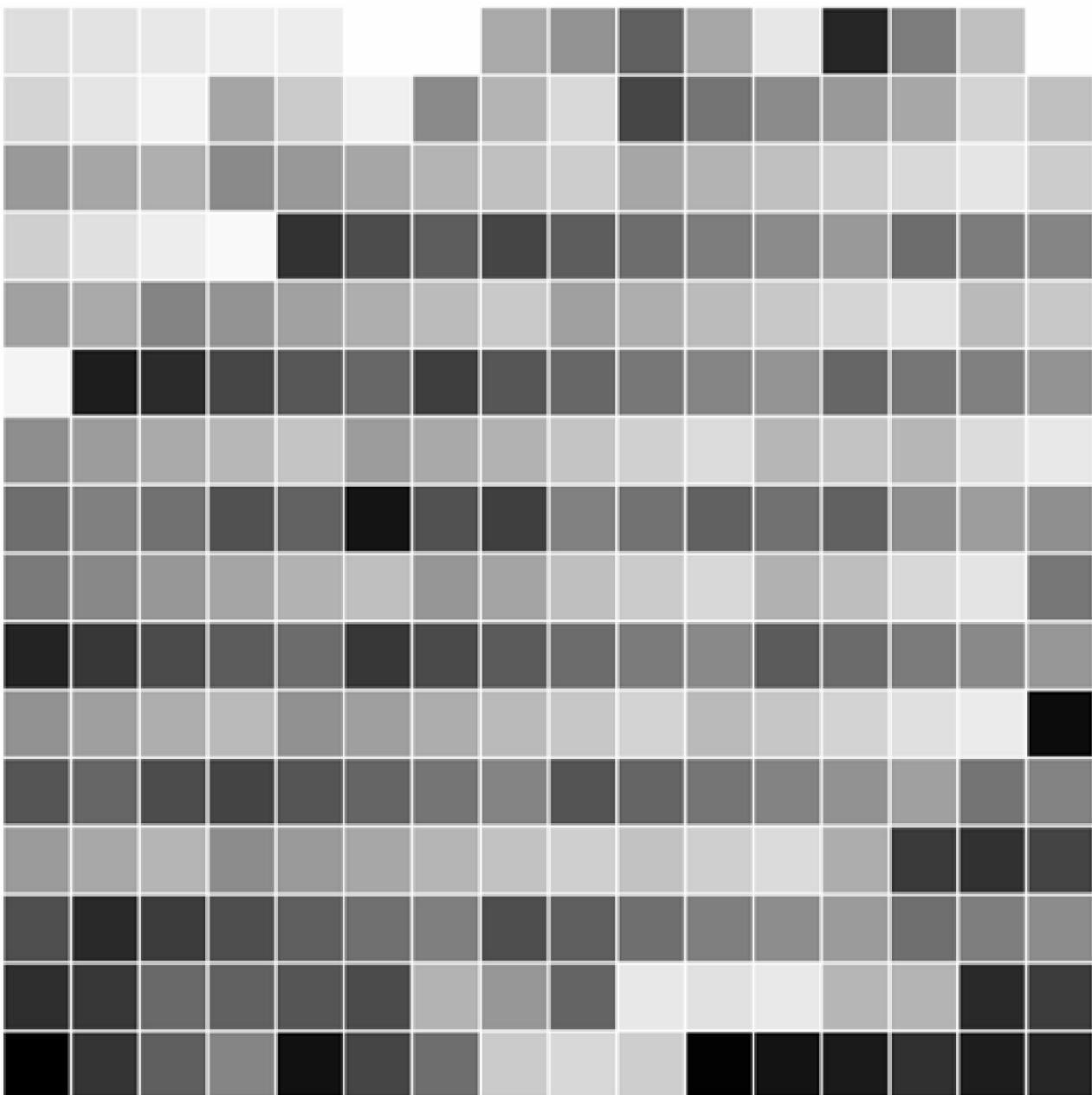
```
for (int j=0; j<(num+15)/16; j++)
for (int i=0; i<16; i++)
    for (int y=0; y<24; y++)
        for (int x=0; x<24; x++)
            SetPixel(hDC, i*25+x, j*25+y, PALETTEINDEX(j*16+i));

SelectPalette(hDC, hOld, FALSE);

if ( bHalftone )
    DeleteObject(hPalette);
}
```

The code works with either the current logical palette or the halftone palette. It uses GetPaletteEntries to find out the number of colors in a logical palette; the return results can also be used to display the RGB components of each color. The code then realizes the palette, if needed, and displays each color in rows of 16 colors using the PALETTEINDEX macro. [Figure 7-6](#) shows the arrangement of 256 colors in the half tone palette.

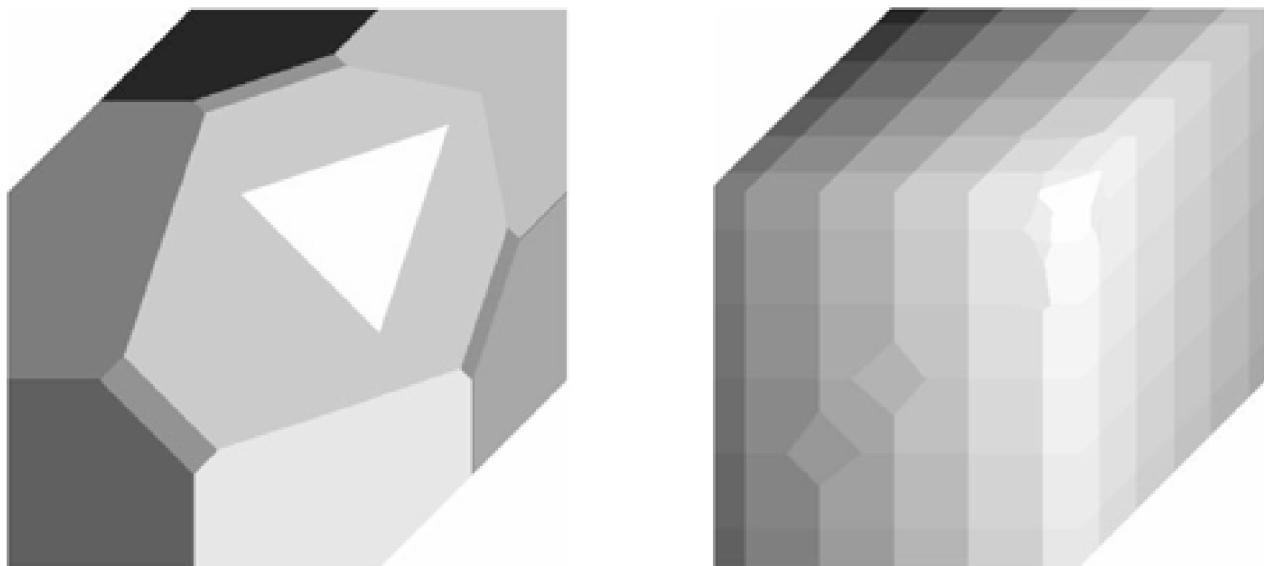
Figure 7-6. Halftone palette displayed using PALETTEINDEX.



The default palette within a device context normally defines only 20 colors, which are the 16 used on the old VGA color display monitor, plus four that define the current Windows color scheme. Twenty colors are barely adequate for a graphics display with a medium number of colors. The halftone palette defines 256 evenly distributed colors within the RGB cube. But if an application wants to display a nice sunset picture, it will care more about warm colors than cold colors. Win32 provides functions allowing applications to define their own palette and managing the interaction with the system palette and other applications on the system competing with the single hardware palette resource. We will cover palette management after discussing bitmap handling in GDI.

Both RGB and PALETTEINDEX should be quite clear now. The RGB macro is best for high-color or true-color display modes. The PALETTEINDEX is handy for palette-based display modes; it also works in high-color or true-color display modes as long as you have a valid logical palette. But what about using the RGB macros in palette-based display modes? The display result is ugly. The left part of [Figure 7-7](#) shows our nice RGB cube in 256-color display mode, which has the same result regardless of whether the halftone palette is selected.

Figure 7-7. An RGB color cube using the RGB vs. PALETTERGB macros in 256-color display mode using the halftone palette.



Quite a nice 3D design, isn't it? But this is not what we want it to be. Only nine colors are used to display the RGB cube, which actually uses $3 \times 256 \times 256$ different colors. To improve the display, you need to create, select, and realize a halftone palette, as shown in [Listing 7-5](#); equally important, you need to replace the RGB macro with the PALETTERGB macro. The right part of [Figure 7-7](#) shows the magic of PALETTERGB. You may find that the cube displayed using the PALETTERGB macro is not purely symmetrical; this reflects the fact that the colors in the color table are not perfectly distributed in an even fashion.

Even the improved version looks ugly compared to the true-color version. This is because the cube is drawn using pixels that are treated individually. The surfaces as a whole are not halftoned during pixel drawing. To achieve a better display without writing our own halftoning algorithm, we need to store the pixels in 24-bit bitmaps and use bitmap drawing commands that handle halftoning.

Beyond the Basics

After a dozen pages, we still say color is a complicated topic. The Win32 API provides more features in creating logical palettes and manipulating the system palette, which will be discussed later in the book. Microsoft also provides an image color management API, ICM 2.0, which is a complicated subject outside of the normal GDI scope. We will not be covering the full ICM 2.0 feature in this book, but some discussion of color space mapping and color adjustment will be provided together with more in-depth coverage of the palette in [Chapter 13](#).

Windows GDI basically supports two types of color spaces: RGB color space and palette-based index color space, reflecting the basic video display hardware. The alpha channel as provided by the latest video display hardware is not supported as a component of GDI color space. Instead, it's supported only in DirectX and a new GDI function, AlphaBlending.

[Figure 7-8](#) shows how the 32-bit COLORREF is specified in the RGB, PALETTERGB, and PALETTEINDEX formats. GDI recognizes their difference by the first byte of a 32-bit value.

Figure 7-8. Three ways of specifying color in GDI.

0	Red	Green	Blue
---	-----	-------	------

RGB(Red, Green, Blue)

1	0	index
---	---	-------

PALETTEINDEX(index)

2	Red	Green	Blue
---	-----	-------	------

PALETTERGB(Red, Green, Blue)



[< BACK](#) [NEXT >](#)

7.4 DRAWING PIXELS

We've used Win32 pixel drawing functions many times in this section. Now that we have done all the groundwork, it's time to introduce them officially and accurately. The Win32 API provides the following function for pixel-level drawing:

```
COLORREF GetPixel (HDC hDC, int X, int Y);  
BOOL SetPixelV(HDC hDC, int X, int Y, COLORREF crColor);  
COLORREF SetPixel (HDC hDC, int X, int Y, COLORREF crColor);
```

Here hDC is a device context handle. First of all, note that not all devices support pixel drawing. To be more precise, no device driver supports pixel drawing directly. There is no DDI-level function that supports per-pixel drawing; instead, GDI maps pixel-drawing calls to bitmap bit-transfer DDI calls. So, when in doubt an application should check `GetDeviceCaps(hDC, RASTERCAPS)` & `RC_BITBLT` to see if a bitmap bit transfer is supported, of which pixel drawing is a special case.

The parameters (X, Y) specify a location in logical coordinate space, which could be in world coordinate space for advanced graphics mode, or page coordinate space for compatible graphics mode. The location goes through world transformation, window-to-viewport mapping, and device-coordinate-space to physical-device-coordinate-space translation, before knowing its final destination.

Whether a drawing actually is performed depends also on whether the location is within the effective clipping region of the device context, or its Rao region, which is the intersection of the device context's system region, meta region, and clipping region. If the location is outside the effective clipping region, an error code is returned (`CLR_INVALID`, which is `0xFFFFFFFF`, or `FALSE` for `SetPixelV`).

For `SetPixel` and `SetPixelV`, the input parameter `crColor` could have three different forms. It could be a result returned by the `RGB` macro's specified color in 24-bit RGB color space. If the device context is for a non-palette-based device—for example, a high-color or true-color display—the RGB value is used directly or after a little truncation to fit into the frame buffer. Otherwise, the graphics engine or device driver finds the closest match and sets the pixel to the color's index within the system palette. If `crColor` is in the `PALETTEINDEX` format, the device context's logical palette is queried to find the corresponding system palette index, which will be the pixel's value in the device's frame buffer. If `crColor` is in the `PALETTERGB` format, running on a non-palette-based device is the same as if the parameter were in the `RGB` format; otherwise, a closest match is sought with the current logical palette, and its result is used.

Note that in a palette-based device context, `RGB` and `PALETTERGB` macros could yield different results. Microsoft documentation does not state clearly how the `RGB` format color value is translated to the palette index, but it seems to be using a palette with only 20 system-defined static colors. Color value specified using `PALETTE RGB` searches the device context's logical palette for a

match, which may have many more colors to choose from.

The difference between SetPixel and SetPixelV is their return value. SetPixelV returns a Boolean value indicating whether the operation was successful, while SetPixel returns the actual matched color value written.

GetPixel returns the color value of the pixel at a specified location. Note that both SetPixel and GetPixel return results in the RGB format, not in the PALETTE INDEX or PALETTERRGB format, even in a palette-based device context. This can cause problems in a palette-based device context. For example, if you apply the following code—which copies a pixel to a new location—to the cube on the right in [Figure 7-7](#), the result looks like the left-side cube, except that only 20 colors are used.

```
void CopyPixel(HDC int x0, int y0, int x1, int y1)
{
    SetPixel(hDC, x1, y1, GetPixel(hDC, x1, y1));
}
```

To make the code work, use GetPixel(hDC, x1, y1) | PALETTERRGB(0, 0, 0), or GetPixel(hDC, x1, y1) | 0x02000000 as the last parameter for SetPixel.

Looking at how the pixel drawing API is actually implemented is especially interesting because it's the simplest drawing call. It also helps us to understand GDI's overhead for individual drawing calls

In Windows NT/2000, both SetPixel and SetPixelV go to the same system service call, _NtGdiSetPixel@16, after some parameter validation on the device context handle. The decoration in the name _NtGdiSetPixel@16 implies it accepts four parameters, all the original inputs. It invokes a software interrupt, which will be dispatched to a routine by the same name in the kernel mode GDI engine (win32k.sys). The kernel NtGdiSetPixel routine locks the device context, maps the logical coordinates to the physical device coordinates, does some simple checking to see if it's out of bounds, translates COLORREF to an index if needed, and calls the graphics device driver's Drv Bit Blt if present, or else the graphics engine's EngBitBlt. The actual color value used in setting the pixel is translated to RGB color if needed. Before the routine returns, the device context is unlocked.

GetPixel goes to the system service call _NtGdiGetPixel@12. Its implementation creates a temporary bitmap and calls DrvCopyBits/EngCopyBits to copy the pixel to it.

What we need to understand here is the performance implication. For each individual GetPixel, SetPixel, or SetPixelV call, the graphics system needs to invoke the software interrupt, switch to and from kernel mode, map the coordinates, translate the palette index, construct a temporary bitmap, and then call the device driver's bitblting function. It's simply lots of work for a single pixel.

Back in [Chapter 1](#), you may remember that we mentioned a way to use a special Pentium instruction to measure the number of clock cycles the CPU needs for a certain operation; we can use the same method to measure the pixel-drawing API performance. [Table 7-2](#) shows some performance measurements.

The measurements show something interesting. First, SetPixel and SetPixelV have similar performance data, although Microsoft documentation claims SetPixelV to be faster because it does not need to return the color value. Converting RGB data to the palette index is a very slow process. It costs about 500 clock cycles to convert RGB to the palette index, and about 3,000 more to convert PALETTERGB to the palette index for the 256-color halftone palette. The time to convert the palette index back to RGB value is negligible since it's a simple table lookup. GetPixel, surprisingly, is much slower than SetPixel.

Overall, the Win32 API pixel-drawing functions are very slow, which should not be a surprise to anyone given that all the overhead is shared by a single pixel. If you can deal with a speed of 30 to 150 thousand pixels a second, you can still use it. If high performance is needed, pixel operations on bitmaps can be easily implemented in C/C++ code using device-independent bitmaps or DIB section. [Chapters 10, 11, and 12](#) cover three different types of bitmaps supported by GDI. With direct pixel access, millions of pixels can be processed within a second.

Table 7-2. Pixel API Performance: 200-MHz Pentium, 100-by-100 calls

API Call	Clock Cycles (256 color)	Clock Cycles (32-bit color)	Best Speed (pixels per second)
SetPixel(RGB())	1,850	1,286	152,881
SetPixel(PALETTERGB())	4,897	1,284	153,374
SetPixel(PALETTEINDEX)	1,345	1,295	153,115
SetPixelV(RGB())	1,880	1,284	153,115
GetPixel()	6,362	6,499	30,251

[< BACK](#) [NEXT >](#)

7.5 SAMPLE PROGRAM: MANDELBROT SET

The Mandelbrot set is a set of points on a 2D surface defined by a simple iteration. Given a point (x, y) , its position on the x - y complex plane $C_{n+1} = x + yi$ is used to define a sequence $C_0, C_1, C_2, \dots, C_n$, where

$C_{n+1} = C_n^2 + C_0$. The point (x, y) belongs to the Mandelbrot set if the values in the sequence fall within a small range and never go to infinity.

Although the definition looks simple, there is no simple mathematical formula to determine whether a point (x, y) is in the set. The only way to do so is to go through the iteration hundreds or even thousands of times. What's more interesting than the set itself is calculating the number of iterations needed to determine if a point is or is not in the set. If you color the points according the iteration number needed, the result picture is very intricate and pleasing to look at.

The Mandelbrot-set graph can never be stored in a fixed-size array of pixels. When you zoom in at a part of the graph, there is no formula to calculate the color value for newly added locations; you have to go through the iterations to uncover another layer of details. So the problem is best represented as a dynamically generated array of pixels, although a bitmap can be used to speed up the display.

The calculation is very time consuming, so for practical reasons, you have to limit the number of iterations to a predefined number like 128, 1024, or 16,384. After the predefined number of iterations, m , several results are possible. It's possible that after $n=m$ iterations, the distance of (x, y) from the origin $(0, 0)$ is larger than 2; then we know for sure that the sequence is going to infinity. Another case is that after $p=m$ iterations, we know that the sequence converges to a single fixed point within a certain error tolerance, or jumps between two, three, four, ... fixed points within a certain error tolerance. The third possibility is that after m iterations, we can't make a determination; the sequence is still quite small and not yet converging well enough.

We can color the diverging points with one color sequence according to m , color the converging points with another color sequence according to p , and finally color the undetermined points with a fixed single color. The HLS color space should give us a good tool to generate color sequences, either by keeping the saturation and lightness and rotating the hue, or by fixing hue, saturation and scale lightness.

[Figure 7-9](#) shows the complete picture of the Mandlebrot set after 128 iterations. [Figure 7-10](#) shows a 64:1 zoom-in view of a tiny portion of the set after 1024 iterations.

Figure 7-9. Complete Mandelbrot set (128 iterations).

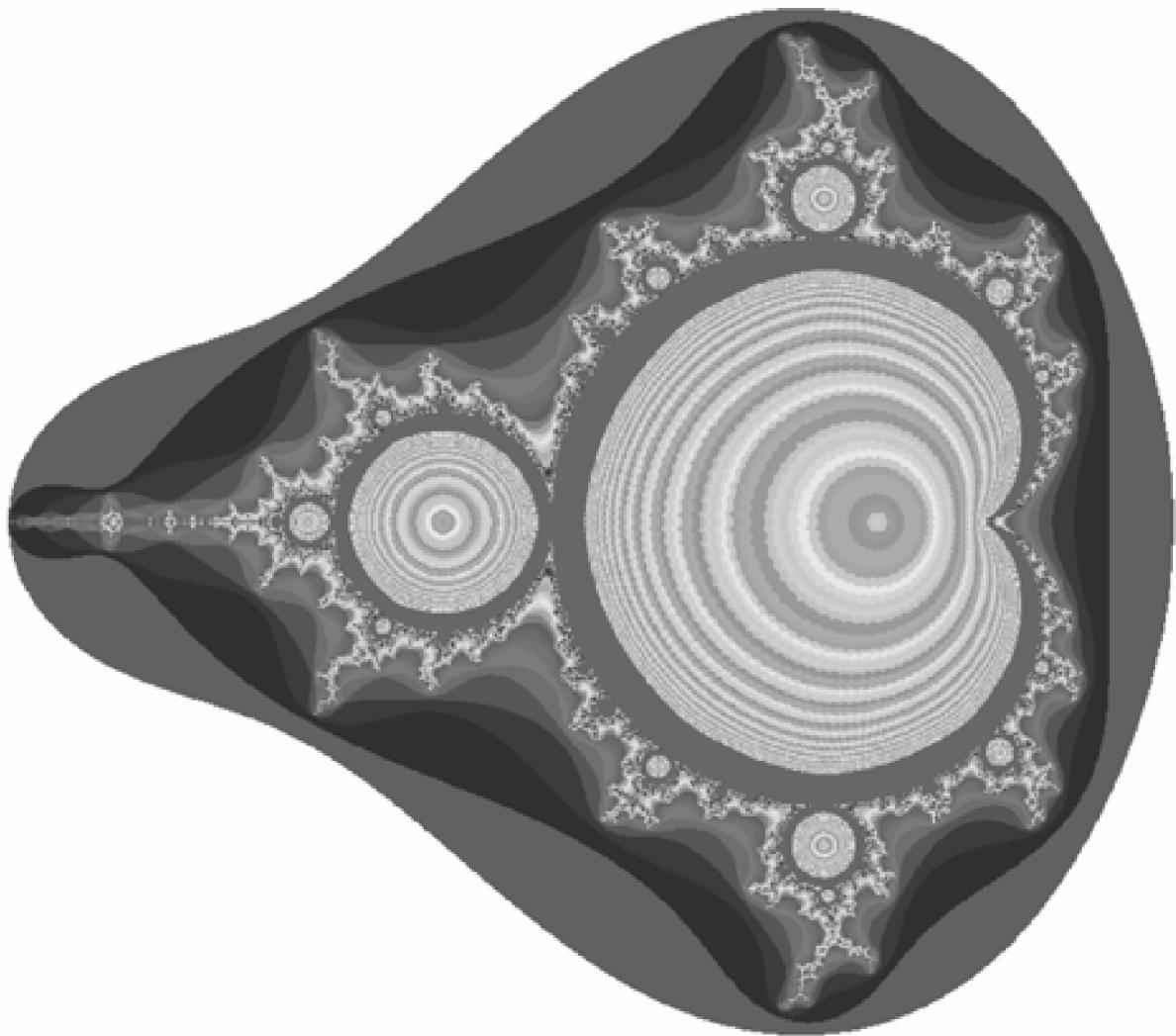
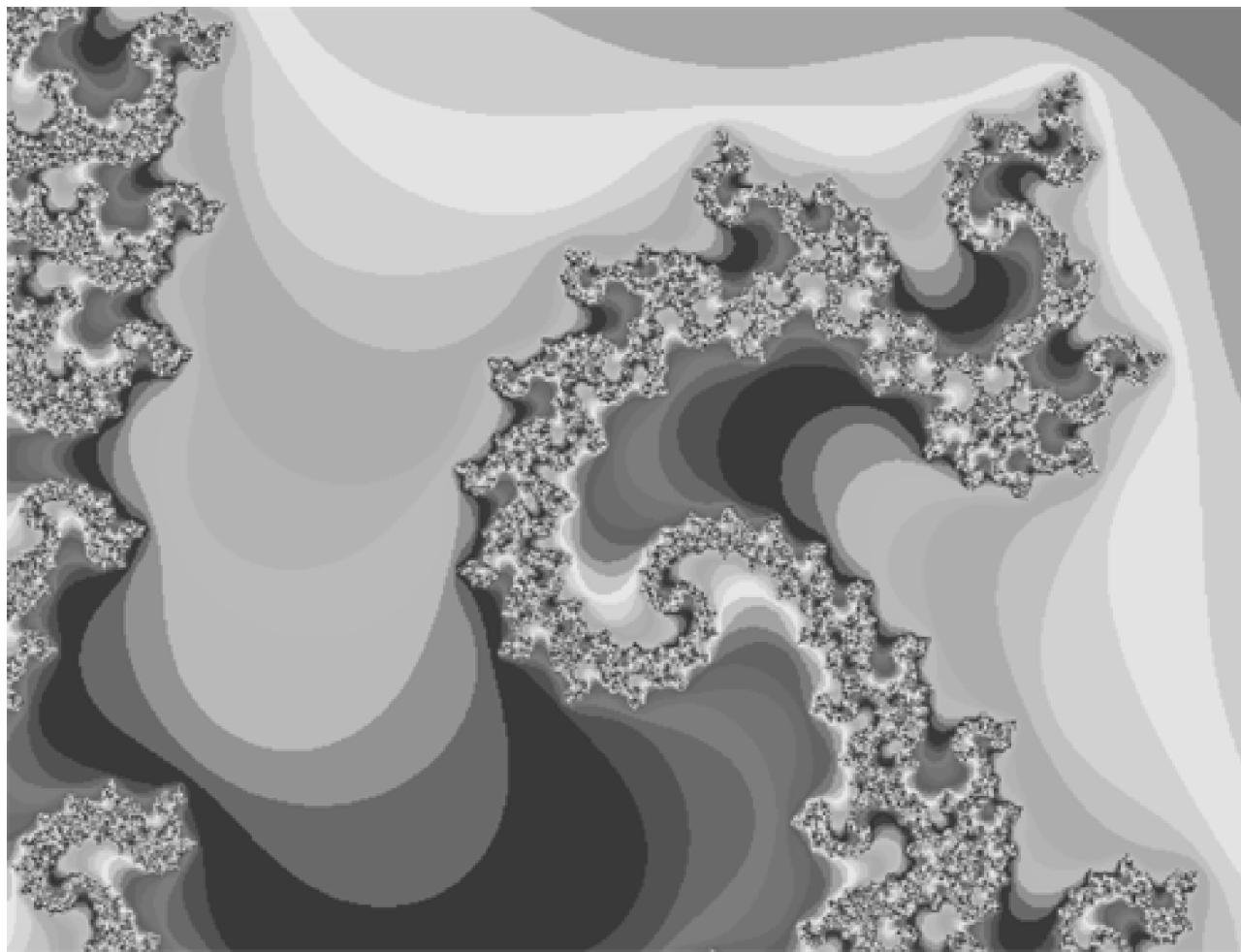


Figure 7-10. 64:1 Zoom-in view of a Mandelbrot subset (1024 iterations).



The Mandelbrot program is based on the KScrollView class to provide the basic mechanism for scrolling and zooming. The OnDraw method implementation queries the system region and checks whether a point is not clipped before starting the lengthy calculation. The calculation routine returns positive numbers for converging points, negative numbers for diverging points, and 0 for undetermined points. Color tables are used to translate the iteration number to COLORREF. Then each pixel is drawn using SetPixel. When we move to other GDI drawing primitives later in the book, we should consider improving the drawing performance by using more advanced draw calls.

Overall, Mandelbrot provides a good sample program to test our understanding of mapping modes, scrolling, clipping, color spaces, and pixel-drawing API.

7.6 SUMMARY

Based on the device context and coordinate spaces covered in [Chapters 5](#) and [6](#), this chapter covered the remaining basic concepts allowing you to draw the basic primitive using GDI: pixels.

[Section 7.1](#) gave a generic description of GDI objects, object handles, and object handle tables. Care must be taken in using GDI objects because they consume system resources, both in the GDI handle table and the kernel mode paged pool. For more details on the internal GDI data structure behind the GDI object handles, refer to [Chapter 3](#) of this book.

[Section 7.2](#) discussed a simple region-creation operation and its use in clipping drawing operations. All drawing in a device context is controlled by its system region, meta region, and clipping region. The system region is controlled from the window manager, while the meta and clipping regions can be controlled by a rich set of GDI functions. A region is not limited to doing clipping in GDI. Its use in GDI painting and DirectDraw will be covered later in this book.

[Section 7.3](#) provided a brief introduction to color spaces and the methods to specify color in GDI. There are three ways to specify a color in GDI: using RGB, PALETTE INDEX, or the PALETTERGB macro. Color will be discussed further when it's used in pens, brushes, and bitmaps, and when a palette is used.

Finally, [Section 7.4](#) talked about pixel-drawing functions provided by GDI, and [Section 7.5](#) gave a sample application using pixels to draw the Mandelbrot set. Pixel drawing is conceptually very flexible and powerful, but its implementation in GDI is slow. Direct pixel access using bitmaps and its application will be discussed in [Chapters 10](#) through [12](#).

[Chapter 8](#) will move up the drawing primitive chain to connect pixels to form more interesting lines and curves.

Further Reading

The Windows GDI provides a color-management API, whose latest version is ICM 2.0. ICM is beyond what can be covered in this book. Refer to MSDN documentation on color management. There is a great deal of information regarding color space and color-space transformation on the Internet; search for key words like "color space," "color-space conversion," or "color-space FAQ."

Sample Programs

Five sample programs are provided with this chapter to monitor GDI object handle usage, demonstrate regions with a device context, demonstrate color space and pixel drawing, and measure pixel-drawing speed (see [Table 7-3](#)).

Table 7-3. Sample Programs for Chapter 7

Directory	Description
Samples\Chapt_07\GDIObj	Monitor the GDI object table to display per-process usage of GDI objects, the warning system of possible GDI object leakage.
Samples\Chapt_07\ClipRegion	Visualize the system region, meta region, and clipping region.
Samples\Chapt_07\ColorSpace	Demonstrate RGB and HLS color space with colors in the half-tone palette.
Samples\Chapt_07\PixelSpeed	Measure pixel-drawing performance
Samples\Chapt_07\Mandelbrot	Draw the Mandelbrot set using pixel-drawing calls.

[< BACK](#) [NEXT >](#)

Chapter 8. Lines and Curves

Connecting pixels together forms lines and curves. So once you know how to draw single pixels, drawing lines and curves is just a matter of mathematics and computer programming. But the lines and curves used in real life come in different colors, styles, widths, patterns and other decorations; drawing them can be quite involved.

This chapter will cover the GDI concepts and features supporting line and curve drawing; these are binary raster operations, background modes, pens, lines, curves, and paths.

8.1 BINARY RASTER OPERATIONS

When you call the SetPixel function to draw a color pixel on a device surface, the color specifier is translated into a color format (physical color) suitable for the device frame buffer, then the data in the destination is replaced with the translated color. If you think of the frame buffer as an array of pixels D, the SetPixel operation can be seen as an assignment $D[x, y] = P$, where P is the translated color, or the physical color.

If we generalize the operation a little bit, we can define a function f that combines the original pixel color $D[x, y]$ and the color P to a new color value, which will be written to the destination. That is: $D[x, y] = f(D[x, y], P)$, or $D = f(D, P)$. Function f takes two parameters, so this is a binary function.

Although innumerable such functions can be defined, GDI supports only one class of functions: bitwise logical operations. Bitwise logical operations pair individual bits in two arguments and apply Boolean logical operations on the bits. For binary Boolean operations, there are 16 possible functions ($2^{2 \times 2}$). These functions are referred to as binary raster operations in GDI, or ROP2 for short. [Table 8-1](#) summarizes the binary raster operations supported by GDI. Note that P here is referred to as the pen color, because ROP2s are used for the line drawing that uses the pen's color.

A device context in GDI has a binary raster operation attribute, also called draw mode, which can be retrieved using GetROP2 and set using SetROP2. The functions are defined as:

```
int SetROP2(HDC hDC, int fnDrawMode);
int GetROP2(HDC hDC);
```

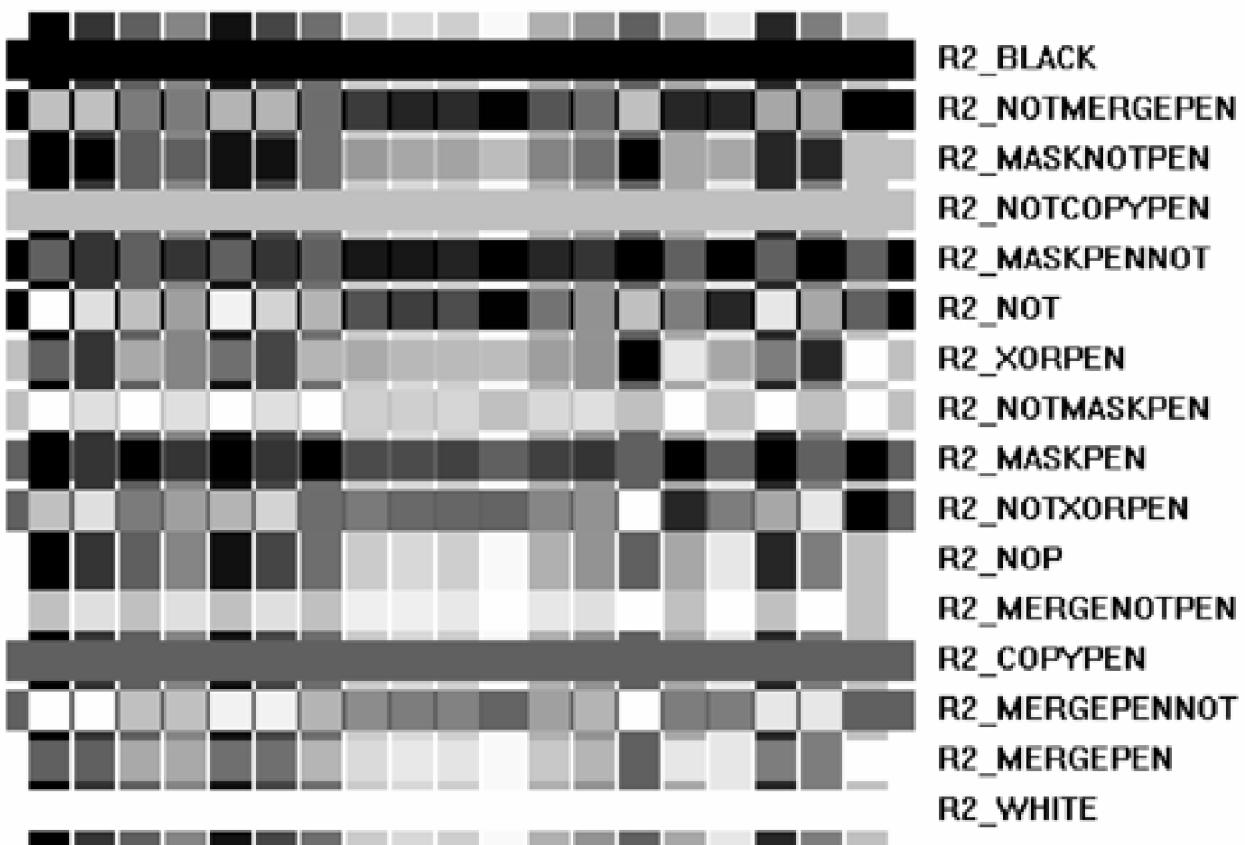
SetROP2 replaces the device context's draw mode with a new value if it's valid and returns the original draw mode; GetROP2 just returns the current draw mode. The default draw mode for a device context is R2_COPYPEN, which sets the destination to the pen color. Draw mode is used whenever the pen is used—that is, for line and curve drawing and framing filled areas.

Table 8-1. Binary Raster Operations

ROP2	Formula	Description
R2_BLACK	$D = 0$	Always 0, black in RGB mode
R2_NOTMERGESEN	$D = \sim(D \mid P)$	Inverse of R2_MERGESEN
R2_MASKNOTPEN	$D = D \& \sim P$	Conjunction of destination and inverse of pen
R2_NOTCOPYPEN	$D = \sim P$	Inverse of pen color
R2_MASKPENNNOT	$D = P \& \sim D$	Conjunction of pen and inverse of destination
R2_NOT	$D = \sim D$	Inverse of destination
R2_XORPEN	$D = D \wedge P$	Exclusive OR of destination and pen
R2_NOTMASKPEN	$D = \sim(D \& P)$	Inverse of R2_MASKPEN
R2_MASKPEN	$D = D \& P$	Conjunction of destination and pen
R2_NOTXORPEN	$D = \sim(D \wedge P)$	Inverse of R2_XORPEN
R2_NOP	$D = D$	No change
R2_MERGENOTPEN	$D = D \mid \sim P$	Disjunction of destination and inverse of pen
R2_COPYPEN	$D = P$	Pen
R2_MERGESENNOT	$D = P \mid \sim D$	Disjunction of pen and inverse of destination
R2_MERGESEN	$D = P \mid D$	Conjunction of pen and destination
R2_WHITE	$D = 1$	Always 1, white in RGB mode

[Figure 8-1](#) shows the effects of applying all 16 binary raster operations on 20 different color bars running under true-color display mode. The vertical color bars are drawn first with colors taken from the device context's default palette. The horizontal bars are drawn after each binary raster operation is set. Just look for the simple cases; you will find that R2_BLACK draws black, R2_NOT inverses the color, R2_NOP does not change the background, and R2_WHITE draws white. You should expect to see different colors in a 256-color display mode, unless the system palette is systematically set up to follow Boolean algebra.

Figure 8-1. Effects of binary raster operations (true-color mode).



When using draw mode, one has to keep in mind that the operation is defined on the physical color, instead of on the logical COLORREF color. So the result of the operations is more or less device dependent. For devices using RGB color space, operations are applied to each of the three RGB components, so the result is predictable but not always meaningful. RGB color space stores the intensity of primary colors, so applying bitwise logical operations does not always have a corresponding meaning in terms of color perception. For palette-based devices, raster operations are applied to color indexes, so the display result depends on how the colors in the palette are arranged. On a multitasking OS like Windows, each application does not have direct control over the system's hardware palette. The GDI functions are provided for the application for changes in the system palette, and priority is given to windows running in the foreground. Special window messages are provided for the application to respond to a system palette changes. [Chapter 13](#) covers palette in detail.

Binary raster operations have an important role in computer graphics. R2_BLACK can be used to clear pixels to all black (0), R2_WHITE sets pixels to white (1), or 0xFFFF in a 24-bit frame buffer. R2_NOTCOPYOPEN provides another pen having the opposite color. R2_NOP turns line and curve drawing off, a very handy feature if you don't want to draw lines around a rectangle.

R2_MASKPEN selectively turns off bits in a device context's drawing surface. For example, if you use R2_MASKPEN with a pen of solid color, RGB(0xFF, 0, 0) in an RGB frame buffer, wherever lines are drawn, the blue and green channel data are masked out, leaving only the red channel data. If you are using color RGB(0x7F, 0x7F, 0x7F), the bright colors will be turned off, because after the drawing the maximum intensity for each channel is only 127 instead of 255.

R2_NOT and R2_XOROPEN are very useful in interactive computer graphics to draw crosshairs or rubber banding. Crosshairs consist of a horizontal and a vertical line across the whole window. The intersection of the two lines is the current cursor position. Crosshair is a useful technique to align objects in graphics editing software. Rubber banding is the dynamically changing line that shows a line the user is trying to define using keyboard arrow keys or a mouse. Rubber banding, rubber rectangle, or even other shapes are used in geometric object definition and

selection in graphics software. When the user is using a mouse to define rectangles or ellipses, the shapes are still under construction, so they can't be drawn permanently. Instead, the application needs to draw them quickly, erase them, restore the original contents and move to a new location when the user moves the mouse. R2_NOT, R2_XORPEN, and R2_NOTXORPEN provide easy and efficient ways of drawing temporary lines and removing them without a trace, because applying the operation twice restores the original contents, their Boolean operation property.

[Listing 8-1](#) shows how crosshairs can be implemented using R2_NOT. The window class keeps the position of the last crosshairs in (m_lastx, m_lasty). For every WM_MOUSEMOVE message, the crosshairs drawing routine is called twice, first to remove the old lines, second to draw the new lines.

Listing 8-1 Crosshair Drawing using R2_NOT

```
void KMyCanvas::DrawCrossHair(HDC hDC)
{
    if ( m_lastx<0 ) return;

    RECT rect;
    GetClientRect(m_hWnd, &rect);

    SetROP2(hDC, R2_NOT);
    MoveToEx(hDC, rect.left, m_lasty, NULL);
    LineTo(hDC, rect.right, m_lasty);
    MoveToEx(hDC, m_lastx, rect.top, NULL);
    LineTo(hDC, m_lastx, rect.bottom);
}

LRESULT KMyCanvas::WndProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam)
{
    switch( uMsg )
    {
        case WM_CREATE:
            m_lastx = m_lasty = -1; // no crosshair yet
            return 0;
        case WM_MOUSEMOVE:
        {
            HDC hDC = GetDC(hWnd);
            DrawCrossHair(hDC);
            m_lastx = LOWORD(lParam); m_lasty = HIWORD(lParam);
            DrawCrossHair(hDC);
            ReleaseDC(hWnd, hDC);
        }
        ...
    }
}
```

Although R2_NOT, R2_XORPEN, and R2_NOTXORPEN can all be used to draw erasable lines, they do have subtle differences. R2_NOT inverts bits in the frame buffer, white turns to black, and black turns to white. So you can

always see a line, but you can't control the color of the line. R2_XORPEN is more generic than R2_NOT; you just need to define a pen whose physical color has 1 in every bit to get the same effect as R2_NOT. In other words, R2_XORPEN with a white pen is the same as R2_NOT in RGB mode. If you know that the background of the client area is mostly in a single color, C, you can use the pen color P and R2_XORPEN to let the line be mostly in color C^P.

Raster operations are not restricted to binary raster operations. For bitmaps, Win32 GDI defines raster operations with three and four operands, which are called ternary or quaternary raster operations. Windows 98 and Windows 2000 start to support nonlogical operations in pixel-level mixing through alpha blending. These techniques will be discussed when we cover bitmaps.

8.2 BACKGROUND MODE AND BACKGROUND COLOR

For certain drawing primitives, GDI divides the pixels to be drawn into two classes: foreground pixels and background pixels. For example, for text display, pixels forming the character glyphs are considered foreground pixels, while other pixels in the text box are considered background pixels. For dash line drawing, pixels forming a dash are foreground pixels; pixels between dashes are background pixels. Foreground pixels are always drawn, but the drawing of background pixels is optional.

Every device context has a background mode attribute that controls whether background pixels should be drawn. Background pixels are drawn using a color specified in a device context's background color attribute. Background mode and background color can be accessed using:

```
int GetBkMode(HDC hDC);
int SetBkMode(HDC hDC, int iBkMode);
COLORREF GetBkColor(HDC hDC);
COLORREF SetBkColor(HDC hDC, COLORREF crColor);
```

The valid value for background mode is OPAQUE for drawing background pixels and TRANSPARENT for ignoring background pixels. A device context's default background mode is OPAQUE; the default background color is white.

Background mode and background color are used in drawing styled lines, hatch brushes, and texts. Background color is also used in converting bitmaps between color and black/white format.

8.3 PENS

Lots of attributes determine how lines are drawn. Some of the attributes that are line specific are grouped together to form a pen object in GDI, so that multiple groups of settings can be stored and referred to easily. To be more specific, a pen object in Win32 GDI captures a line or a curve's width, style, color, end cap, joint, and pattern.

A pen's width determines how wide the lines would be. Single-pixel lines are very useful for screen display and for the thinnest lines in engineering drawings. Lines in fixed physical width are essential to make sure documents are printed the same way when rendered on printers with different resolutions. A pen's style controls whether the line is a solid line, dotted line, dashed line, or even a line with a user-defined style. Pens are normally defined with a solid color that will be applied to every pixel on the line, but Win32 GDI also allows using a pattern defined by a brush object to draw lines. A pen's end cap and joint attributes control extra details at both ends and at the connection joints of lines.

Logical Pen Objects

GDI allows the creation of pen objects (or, more precisely, logical pen objects). A logical pen is merely a description of an application's requirements for a pen, which does not have every detail on how the lines are going to be drawn on a physical device surface. The graphics device driver may keep its own data structure on how a logical pen should be implemented; such an internal object will be called a physical pen.

The exact data structure of logical pens is managed by GDI, together with other objects like device contexts, logical brushes, logical fonts, etc. Once a logical pen is created, its handle is returned to the application, which serves as its future reference when using it. The generic type for the GDI object handles is HGDIOBJ; HPEN is reserved for logical pen handles. When the STRICT macro is defined, GDI include files that define HGDIOBJ as a void pointer and HPEN and other specific handle types as totally different structure pointers. So an HPEN can be treated as an HGDIOBJ, but treating an HGDIOBJ as an HPEN needs type casting. But when the STRICT macro is not defined, all handle types are declared as void pointers, so the compiler is not responsible for finding handle mismatches.

Each device context has a logical pen object attribute. Unlike other attributes, such as the background mode and binary raster operation, the logical pen object attribute is accessed using several generic functions.

```
HGDIOBJ GetCurrentObject(HDC hDC, UINT uObjectType);
HGDIOBJ SelectObject(HDC hDC, HGDIOBJ hgdiobj);
int GetObject(HGDIOBJ hgdiobj, int cbBuffer, LPVOID lpvObject);
int EnumObjects(HDC hDC, int nObjectType,
    GOBJENUMPROC lpObjectProc, LPARAM lParam);
```

GetCurrentObject returns one of the current GDI object handles kept in a device context, according to the uObjectType parameter. For example, GetCurrentObject(hDC, OBJ_PEN) returns the current logical pen object handle. SelectObject replaces a GDI object handle in a device context with a new object handle and returns the old handle as a return result. Thus, if a valid logical pen object handle is given, Select Object changes the logical pen object attribute in a device context. GetObject returns the original definition of a GDI object. EnumObjects calls a user-provided routine for every user-accessible GDI object handle within a device context.

Like other GDI objects, a logical pen object consumes user process resources, kernel resources, and an entry in the GDI handle table. So once the logical pen is no longer needed, it should be deselected from a device context and deleted by calling Delete Object. For the same reason, an application should refrain from creating too many GDI objects and free them only when the application quits. For Win32 systems, all processes in the system share 16,384 GDI handles. So if one application creates 1024 GDI object handles, a maximum of 16 such applications can be run at the same time. The good news is that the operating system is able to delete all the GDI objects created by a process when it quits to recover all the GDI resources it consumes.

Stock Pens

GDI defines four predefined pen objects, called stock pens, which can be used easily by applications. To get a stock pen handle, call GetStockObject with a stock object index. GetStockObject(BLACK_PEN) returns a solid, single-pixel-wide black pen, which is also the default pen in a device context. GetStockObject(WHITE_PEN) returns a solid, single-pixel-wide white pen. GetStockObject(NULL_PEN) returns a pen that draws nothing. It can be used to disable a line drawing.

The black, white, and null stock pens have been around for a long time. Windows 98/2000 finally adds a new stock pen, the DC pen, as returned by GetStock Object (DC_PEN). The DC pen, or, more generically, a device context GDI object, is a totally new concept. Normally, when a GDI object is created, it's cast in iron. It can only be used and then destroyed, never changed. If you want a slightly different GDI object, you need to create a new one and destroy the old one. This could cause performance problems if a huge number of GDI objects are needed (for example, in implementing a gradient fill without direct GDI support).

The DC pen belongs to a new type of GDI objects, of which the DC brush is another member. You may call it a *device context GDI object*, because it's special only when attached with a device context. A device context GDI object is modifiable to a certain degree, after it's selected into a device context. If changes needed are supported by GDI, no new GDI objects need to be created or old ones replaced.

The default DC pen is a single-pixel, solid black pen. Once it's selected into a device context, only its color can be changed. The DC pen's color can be accessed using:

```
COLORREF GetDCPenColor(HDC hDC);
COLORREF SetDCPenColor(HDC hDC, COLORREF crColor);
```

GetDCPenColor returns the current color of the DC pen in a device context. SetDCPenColor sets a new color and returns the old color. These functions can be used even when the DC pen is not selected into a device context. The DC pen color can be seen as a new attribute of a device context, which is used in line drawing only when the DC pen is selected as the current pen object. The following code shows how the DC pen can be used to draw a gradient fill with only a single pen:

```
HGDIOBJ hOld = SelectObject(hDC, GetStockObject(DC_PEN));
for (int i=0; i<128; i++)
{
    SetDCPenColor(hDC, RGB(i, 255-i, 128+i));
    MoveToEx(hDC, 10, i+10, NULL); LineTo(hDC, 110, i+10);
}
```

```
SelectObject(hDC, hOld);
```

After retrieving and selecting the DC pen, the code uses SetDCPenColor to switch the pen color gradually from RGB(0, 255, 128) to RGB(127, 128, 255) to form a gradient fill using lines. The code restores the original pen after finishing drawing. Without the DC pen, the code would have to create a new pen, select it into the device context, and delete the previous pen for every iteration of the loop.

Stock objects are precreated by the operating system and shared by all processes running on the system. There is no need to delete the stock object handles after an application is done with them. But it's completely safe to call DeleteObject on a stock object handle. DeleteObject returns TRUE without doing anything.

Simple Pens

Stock pens are all solid single-pixel pens. To draw lines with nonsolid styles or to draw thick lines, applications need to create custom logical pen objects. Here are two easy ways to create simple pens:

```
HPEN CreatePen(int fnPenStyle, int nWidth, COLORREF crColor);
HPEN CreatePenIndirect(CONST LOGPEN * lplgpn);
```

The LOGPEN structures stores the three parameters of a logical pen, namely, its style, width, and color reference. So, the above two functions are two variants of the same function. In a real implementation, CreatePenIndirect dereferences data in the LOGPEN structure and calls CreatePen.

A pen's style specifies pixel cycles along the line and the placement of the pen. [Table 8-2](#) summarizes different pen styles and implementation limitations.

Part of a pen's style is the pixel on-off cycle along the line. PS_SOLID and PS_INSIDEFRAME draw solid lines, PS_NULL does not draw lines at all, so we need to worry about only 4 other styles. PS_DASH is implemented using an 18-pixels-on, 6-pixels-off cycle. PS_DOT is implemented using a 3-pixels-on, 3-pixels-off cycle. PS_DASHDOT is 9 pixels on, 6 pixels off, 3 pixels on, 6 pixels off. PS_DASHDOTDOT is 9 pixels on, 3 pixels off, 3 pixels on, 3 pixels off, 3 pixels on, and 3 pixels off. Clearly, Microsoft thinks a single pixel is too small to be a dot; instead, three pixels are used as a dot.

Table 8-2. Simple Pen Styles

Style	Pixel Cycle	Placement	Restrictions
PS_SOLID	Solid, all pixels drawn	Center frame	
PS_DASH	Dashed	Center frame	Width =1
PS_DOT	Dotted	Center frame	Width =1
PS_DASHDOT	Alternating dashes and dots	Center frame	Width =1
PS_DASHDOTDOT	Dash, and two dots	Center frame	Width =1
PS_NULL	No line is drawn	NA	
PS_INSIDEFRAME	Solid, all pixels drawn	Inside frame	Width >1, used only in framing certain GDI area fills

[Figure 8-2](#) shows the different pixel cycles for the line styles listed in [Table 8-2](#). The left column shows the style name,

the middle column shows the single-pixel style lines, and the last column shows a zoomed-in view of the style lines. The lines are drawn using the OPAQUE background mode, with a dark pen color and a light background color. The graph shows that PS_NULL does not draw anything, not even background pixels. PS_INSIDEFRAME is the same as PS_SOLID when the width of a line is a single pixel in the device coordinate space. The pixel cycles of the other style lines can be seen clearly.

Figure 8-2. Simple pen styles.

PS_SOLID	—		
PS_DASH	— —		
PS_DOT		
PS_DASHDOT	— - -		
PS_DASHDOTDOT	— - -		
PS_NULL			
PS_INSIDEFRAME	—		

The second parameter of CreatePen is the width of a line in logical coordinate space. The actual line width in physical device coordinate space depends on the world transformation and window-to-viewport mapping. For example, in MM_LO ENGLISH mapping mode with identity world transformation, a pen with a logical width of 10 should always draw a line that's close to 0.1 inch in width on a printer, independent of the printer resolution. One-tenth of an inch is 30 pixels on a 300-dpi printer, and 120 pixels on a 1200-dpi printer. A pen with 0 width has a special interpretation; it always draws single-pixel lines in physical device coordinate space.

When the physical width of a pen is more than one pixel, a pen created using CreatePen fails to draw true style lines—for example, dashed or dotted lines. All such pens draw solid thicker lines only. In other words, the logical pen created by CreatePen draws only single-pixel styled lines.

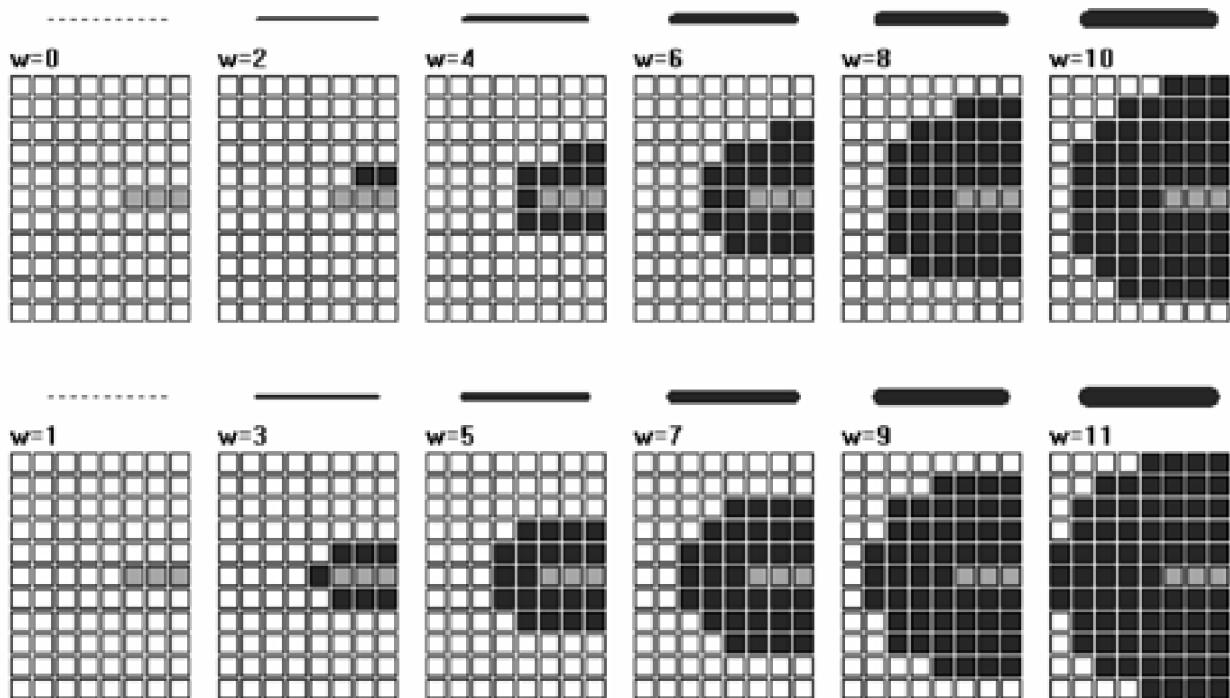
Wider pens bring added complexity to the exact lines they draw. Each line is defined by a base line in the coordinate space. For example, two points (0, 0) and (100, 0) define a horizontal line. If the line is a single pixel wide, it's quite clear that (0, 0), (1, 0), (2, 0), and up to (99, 0) should be in the line. But if the line is 5 pixels wide, there are different ways the lines could be drawn. So the first question is, how should GDI place the multiple pixels across the width of a line relative to the base line? All pens, except pens with the PS_INSIDEFRAME style, center the line on the base line definition. When a PS_INSIDEFRAME style pen is used to frame certain GDI shapes such as rectangles, rounded rectangles, or ellipses, the center of the line is moved inwards to the area to be filled to make sure the complete line is within the shape definition. Note that to draw a line inside the frame, GDI needs to know exactly which side of the two sides of a line is inside and which is outside. So for a normal line drawing, or even polygon drawing, an inside frame line is treated the same as a solid line, that is centered on the base line, because GDI does not know which direction is inside. The PS_INSIDEFRAMR style is active only when drawing certain shapes having a clear inside-outside distinction. Not every line can be drawn perfectly centered on a base line. For vertical and horizontal lines, only lines with odd widths can have one pixel in the center and the others evenly distributed on the two sides.

The second question about a wider line concerns how GDI should draw the start point and end point of a line or the end caps of a line. For pens created with CreatePen, semicircular end caps are always drawn.

[Figure 8-3](#) illustrates a PS_DOT style line created with CreatePen with different widths. When the physical width is 0 or 1, the line is truly a dotted line. When the pen gets wider, extra pixels (dark dots) are added to both sides of the base line (lighter dots) to form thicker lines and round end caps. Notice that only those lines with odd widths center

evenly on the base line.

Figure 8-3. Round end cap, center frame lines with different widths.



Note

Windows 95/98 does not display a thicker line in the manner shown in [Figure 8-3](#), which was captured on Windows 2000. Pixels are not evenly distributed on the two sides of the base line, and the round-end shape does not resemble a good semicircular shape.

Extended Pens

After understanding all the rules, you will realize that the simple pens created by CreatePen or CreatePenIndirect have only two forms: a single-pixel style pen or a wider solid pen with round end caps. The inside frame style is not a very useful feature as designed and implemented in Windows GDI because its usage is limited to a few shapes defined by a rectangular bounding box. But for those shapes, an application can easily shrink the bounding box to use a normal pen to achieve the same effect as an inside frame pen.

To overcome part of this limitation, Win32 API provides a new function, Ext CreatePen, to create extended pens with richer attributes.

HPEN ExtCreatePen(DWORD dwPenStyle, DWORD dwWidth,
CONST LOGBRUSH * lpb, DWORD dwStyleCont,

```
CONST DWORD * lpStyle);
```

ExtCreatePen allows pens to have 2 types, 9 styles, 3 end caps, and 3 different joins, all specified in a single dwPenStyle parameter, by combining flags shown in [Table 8-3](#) using the bitwise OR operator (|).

Two types of pens are defined: cosmetic pens and geometric pens. The dw Width parameter specifies the pen width. For cosmetic pens, only one is allowed. For geometric pens, the width is in logical coordinate space, so the actual width of lines depends on world transformation and window-to-viewport mapping. Extended pens use a LOGBRUSH structure to define both their color and patterns. Cosmetic pens use only the solid color and solid fill, while geometric pens can use dithered color and various brush patterns. The last two parameters to ExtCreatePen define a user-defined pixel cycle for PS_USERSTYLE pens.

Cosmetic Pens

Cosmetic pens always draw single-pixel lines. Although MSDN claims that cosmetic pens have arbitrary widths specified in a device coordinate space unit, the dw Width parameter accepts only 1 as parameter; all other values fail.

Windows NT/2000 adds two more line styles: PS_USERSTYLE and PS_ALTERNATE. PS_ALTERNATE can be used only in cosmetic pens to generate true dotted lines. PS_USERSTYLE allows an application to define its customized pens' pixel cycle, or style bits. A user style pen needs two more parameters: a DWORD passed in the dwStyleCount parameter and a DWORD array passed in the lpStyle parameter. The first item in the array is the length of the first pixel run, the second item is the length of the space, the third item is the length of the second pixel run, and so on. One unit here is translated to 3 pixels instead of one pixel. So user style pens can simulate PS_DASH, PS_DOT, PS_DASHDOT, and PS_DASHDOTDOT styles, but not the PS_ALTERNATE style. Here is how to create a cosmetic pen with a {4, 3, 2, 1} pixel cycle—that is, 12 pixels on, 9 pixels off, 6 pixels on, and 3 pixels off.

Table 8-3. Extended Pen Types, Styles, End Caps, and Joins

	Flag	Meaning	Restrictions
Type	PS_COSMETIC	Cosmetic pen, single-pixel pen.	
	PS_GEOMETRIC	Geometric pen, pen width specified in logical units.	
Style	PS_SOLID	Solid, all pixels drawn. Center frame.	
	PS_DASH	Dashed. Center frame.	Win 95/98 has no geometric pen with these styles.
	PS_DOT	Dotted. Center frame.	
	PS_DASHDOT	Alternating dashes and dots. Center frame.	
	PS_DASHDOTDOT	Dash and two dots. Center frame.	
	PS_NULL	No line is drawn.	
	PS_INSIDEFRAME	Solid, all pixels drawn. Inside frame.	Geometric pens only, used only in framing certain GDI area fills.
	PS_USERSTYLE	Pixel cycle specified using dwStyleCount, lpStyle. Center frame.	Supported only on Windows NT/2000.
	PS_ALTERNATE	One pixel on, one pixel off.	Supported only on Windows NT/2000. Cosmetic pens only.
End Cap	CapPS_ENDCAP_ROUND	Round, semicircle added.	Geometric pens only.
	PS_ENDCAP_SQUARE	Square, semisquare added.	Geometric pens only.
	PS_ENDCAP_FLAT	Flat, no extra cap.	Geometric pens only.
Join	PS_JOIN_BEVEL	Joins are beveled.	Geometric pens only.
	PS_JOIN_MITER	Joins are mitered with limit set by SetMiterLimit.	Geometric pens only.
	PS_JOIN_ROUND	Joins are round.	Geometric pens only.

```

const DWORD  cycle[4] = { 4, 3, 2, 1 };
const LOGBRUSH brush  = { BS_SOLID, RGB(0,0,0xFF), 0 };
HPEN hPen = ExtCreatePen(PS_COSMETIC | PS_USERSTYLE, 1, &
    brush, sizeof(cycle)/sizeof(cycle[0]), cycle);

```

It may appear that cosmetic pens are similar to simple pens created using Create Pen with 0 width. But they have a very important undocumented difference, or perhaps it's an implementation defect: Cosmetic pens always draw in transparent mode. That is, the off pixels are not drawn using background color, even when background mode is opaque. [Figure 8-4](#) illustrates each of the cosmetic pen styles. The user style line uses a {4, 3, 2, 1} cycle. Note that PS_INSIDEFRAME is an invalid cosmetic pen style that is not automatically converted to solid line style.

Figure 8-4. Cosmetic pen styles.

PS_SOLID	—	
PS_DASH	— —	
PS_DOT	-----	
PS_DASHDOT	— - -	
PS_DASHDOTDOT	-----	
PS_NULL		
PS_INSIDEFRAME		
PS_USERSTYLE	— -	
PS_ALTERNATE	-----	

As the name may suggest, cosmetic pens are good for drawing fine lines, especially on screen displays. When drawing to a printer device context which has a much higher resolution, style lines look like solid lines of a lighter color, and even solid lines are visible only when they have a high contrast with the background color.

Geometric Pens

Geometric pens draw lines with geometric shapes. To be more specific, a geometric pen has a scalable width, different styles, end caps, and joins. Now let's look at these attributes in more detail.

The width of a geometric pen is specified in logical coordinates, but unlike pens created using CreatePen, 0 is an invalid parameter for a geometric pen's width. The actual physical width of a geometric is not that simple. If MM_TEXT mode is used with identity transformation, one logical device unit is translated into one physical device unit, so the physical width is the same as the logical width. If world transformation and window-to-viewport mapping have the same scale factors on both axes, logical pen width is scaled by the scale factor. But if they are different, vertical and horizontal lines drawn using the same geometric pen may appear to have different widths. The same is true of pens created using CreatePen. A line drawn with a geometric pen should be considered a geometric shape instead of simply a series of pixels. For example, line (0, 0) to (100, 0) with a pen width 10 has the same geometric shape as a rectangle defined by its two diagonal corners (0, 0) and (100, 10). World transformation and mapping affect the line in the same way as a rectangle. [Figure 8-5](#) illustrates the geometric control points of a rotated geometric line: (x_0, y_0) to (x_1, y_1) .

[Figure 8-5. Geometric line as geometric shape.](#)

$$dx = pen_width * \sin(\theta)/2$$

$$dy = pen_width * \cos(\theta)/2$$

$(x_0 - dx, y_0 + dy)$

(x_0, y_0)

$(x_0 + dx, y_0 - dy)$

$(x_1 - dx, y_1 + dy)$

(x_1, y_1)

$(x_1 + dx, y_1 - dy)$

The end cap attribute of a geometric line determines an extra attachment to the both ends of a line or segments within a line. [Figure 8-5](#) shows a line with no extra attachment at either end, corresponding to a flat end cap (PS_ENDCAP_FLAT). Wider lines using simple pens created with CreatePen have an extra semicircle on both ends, which is called a round cap (PS_ENDCAP_ROUND). Another option, PS_END_CAP_SQUARE, adds a semisquare to both ends.

On Windows NT/2000, all geometric line styles are fully implemented except for the PS_ALTERNATE style, which is reserved for cosmetic lines. Unlike simple pens created using CreatePen, wider geometric lines are drawn using the style specification and are not converted to solid lines. Geometric pens do not draw one dot using three pixels; instead, the size of dots or dashes scales with the line width. Single-pixel geometric pens draw one dot as one pixel, just like cosmetic lines with the PS_ALTERNATE style. When the pens get wider, the dots also get bigger. Each dash or dot within a style line is drawn with end caps on both ends. So you can have dashes with a flat cap, round cap or square cap. The bad news is that the same Win32 GDI API is not implemented the same way on Windows 95/98. On those 16-bit GDI-based operating systems, wide geometric pens are implemented as solid pens.

The drawing of geometric lines is not restricted to a solid color any more. The Ext Create Pen function accepts a LOGBRUSH structure, which has a color specifier, a brush style, and a hatch style. The LOGBRUSH is normally used to define logical brushes that are used to do area fills instead of line drawing. But geometric lines have no essential difference from a geometric shape, so GDI naturally allows filling a geometric line with a brush.

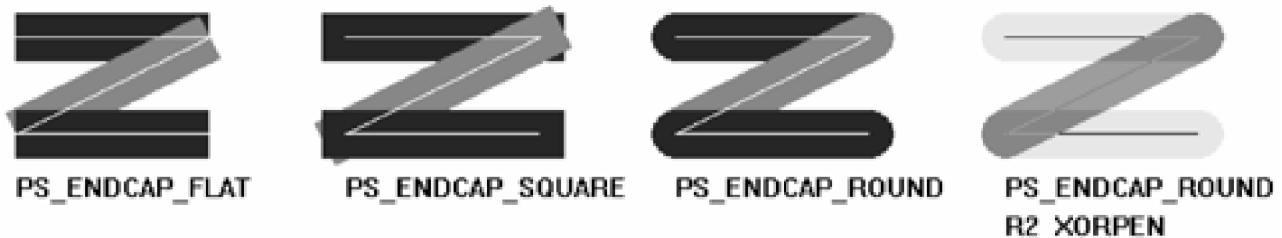
[Figure 8-6](#) illustrates geometric lines with different end caps, style, and pattern. Note that PS_ALTERNATE is not a valid style for a geometric line, and a PS_NULL pen does not draw anything. Like cosmetic pens, geometric pens draw lines in transparent mode, ignoring the background mode and background color except that hatch brushes do use them to draw between the hatches. Also worth noting here is that the user style line's pixel cycle array is interpreted in pixel units instead of logical units, which does change with pen width. As the pens get wider or end caps get added, dashes within a user style line can bump into each other.

Figure 8-6. Geometric lines with different widths, end caps, hatches, and styles.

	w=3, flat	w=7, flat	w=11, square	w=15, round, hatch
PS_SOLID				
PS_DASH				
PS_DOT				
PS_DASHDOT				
PS_DASHDOTDOT				
PS_NULL				
PS_INSIDEFRAME				
PS_USERSTYLE				
PS_ALTERNATE				

When thick lines with end caps meet, their connection may not be what you may expect. [Figure 8-7](#) shows how a Z-shape is drawn using three thick lines under different end cap attributes. Extra base lines are drawn using thin white color lines, which exposes the end caps added by PS_ENDCAP_SQUARE and PS_ENDCAP_ROUND flags. Both flat and square end caps show bad connections. Although the round end cap connects lines smoothly, when drawn using the R2_XORPEN raster operation the common parts of the lines are not in the same color as the disjoint parts because they are painted multiple times.

Figure 8-7. Line caps joining multiple lines.



To have smooth joins between lines, GDI allows merging multiple lines and curves into a single drawing call. When a geometric pen is used to draw a line or curve with multiple segments, its join attribute controls how they are connected together. There are three types of joins. The bevel join can be generated based on the shape of drawing two segments using a flat end cap; just add a triangle to change the concave part to convex. The miter join can be generated from the same shape by extending lines until they meet. The shape of the round join is the same as when drawing two lines using the round end cap. [Figure 8-8](#) illustrates drawing the same Z-shape using the Polyline GDI function, which draws multiple lines in a single call with three different join types. Besides the nicer connections between the line segments, Polyline draws each pixel only once, so that even when a raster operation like R2_XORPEN is used, the whole line has a uniform color.

Figure 8-8. Join types in Polyline drawing.



For line connections with small angles, the extension needed for a miter join could be very long. To avoid extending the miter join indefinitely, GDI allows an application to limit the miter join by using SetMiterLimit.

```
BOOL SetMiterLimit(HDC hDC, FLOAT eNewLimit,  
PFLOAT peOldLimit);
```

SetMiterLimit sets the maximum ratio of miter length compared with the width of the line. For the second graph in [Figure 8-8](#), the miter length is the distance from the intersection of line walls on the inside of the join to the intersection of line walls on the outside of the join. The y-axis distance of the two intersections is the width of the pen. The default miter limit in a device context is 10.0. The graph in [Figure 8-8](#) has a miter ratio of about 4.35. If the miter ratio needed is larger than the miter limit, the join uses a bevel join instead.

If you prefer some mathematics, given two lines with angle theta, their miter ratio as defined above is $1/\sin(\theta/2)$, which is independent of the pen width. In the case shown above, the Z-shape's width is twice its height, so $\sin(\theta) = 1/\sqrt{5}$, $\sin(\theta/2) = 0.229753$, and its miter ratio is officially 4.352502.

Query Logical Pens

Given a logical pen object handle, an application can query its object type and object definition using two generic functions.

```
DWORD GetObjectType(HGDIOBJ h);  
int GetObject(HGDIOBJ hgdiobj, int cbBuffer, LPVOID lpvObject);
```

GetObjectType returns the type identifier of a GDI object. For a simple pen, it returns OBJ_PEN; for an extended pen, it returns OBJ_EXTPEN. GetObject fills a memory buffer with the definition of a GDI object. For a simple pen, it fills a LOGPEN structure; for an extended pen, it fills an EXTLOGPEN structure. LOGPEN is a fixed-size structure, so calling GetObject for a simple pen is quite straightforward. But EXTLOGPEN is a variable-length structure due to the variable style bits array, so calling GetObject should be done in two calls. The first call is to figure out the exact size of the EXTLOGPEN structure, and the second call fills in the structure after the required amount of memory is secured. This two-step calling conversion is quite common in Win32 API. Here is a code fragment on how these two functions should be used to query for the LOGPEN or EXTLOGPEN structure depending on the type of pen object handle:

```
LOGPEN logpen;  
EXTLOGPEN * pextlogpen = NULL;  
int size = 0;
```

```
switch ( GetObjectType(hPen) )
{
    case OBJ_PEN:
        GetObject(hPen, sizeof(logpen), & logpen);
        break;

    case OBJ_EXTPEN:
        size = GetObject(hPen, 0, NULL);
        pextlogpen = (EXTLOGPEN *) new char[size];
        GetObject(hPen, size, pextlogpen);
        break;
    default:
        ...
}

if ( pextlogpen )
{
    delete [] (char *) pextlogpen;
    pextlogpen = NULL;
}
```

A Wrapper Class for GDI Pen Objects

To use a GDI custom pen object, it needs to be created and selected in a device context; after its use, it should be deselected from the device context and deleted. Although it's not hard to do these little tasks, it's not interesting to do them either. Here is a simple C++ wrapper class, the KPen class, which handles simple pens and normal extended pens:

```
// (Another) wrapper for GDI object selection
class KSelect
{
    HGDIOBJ m_hOld;
    HDC     m_hDC;

public:

void Select(HDC hDC, HGDIOBJ hObject)
{
    if ( hDC )
    {
        m_hDC = hDC;
        m_hOld = SelectObject(hDC, hObject);
    }
    else
    {
        m_hDC = NULL;
```

```
m_hOld = NULL;
}

}

void UnSelect(void)
{
    if ( m_hDC )
    {
        SelectObject(m_hDC, m_hOld);
        m_hDC = NULL;
        m_hOld = NULL;
    }
}
};

// Wrapper class for GDI pen object
class KPen : public KSelect
{
public:
    HPEN    m_hPen;

KPen(int style, int width, COLORREF color, HDC hDC=NULL)
{
    m_hPen = CreatePen(style, width, color);
    Select(hDC);
}

KPen(int style, int width, COLORREF color, int count, DWORD *
     gap, HDC hDC=NULL)
{
    LOGBRUSH logbrush = { BS_SOLID, color, 0 };

    m_hPen = ExtCreatePen(style, width, &logbrush, count, gap);
    Select(hDC);
}

void Select(HDC hDC)
{
    KSelect::Select(hDC, m_hPen);
}

~KPen()
{
    UnSelect();
    DeleteObject(m_hPen);
}
};
```

The KPen class has two constructors: one for creating simple pens and the other for creating extended pens without

a pattern brush. One more constructor can be added to handle extended pens with a pattern brush. Both constructors have an optional device context handle parameter. If it's given, the GDI pen handle created will be selected into the given device context. Its destructor deselects it from the device context if it's still selected and deletes the object. Two extra methods are provided to select and deselect the pen handle explicitly.

Using the KPen class is extremely simple. If you have only a single pen in a piece of drawing-call code, make sure the KPen class instance is in a proper block, pass a device context handle to the constructor, and then the GDI object creation, selection, deselection, and deletion will be handled automatically. If you have a piece of code using multiple pens, do not pass a device context handle to the constructor; instead, call Select and UnSelect methods when needed. Here are some simple examples:

```
{  
    KPen red(PS_SOLID, 1, RGB(0xFF, 0, 0), hDC);  
    // draw with red pen  
    ...  
    // pen automatically deselected & deleted when this block ends  
}  
  
{  
    KPen red (PS_SOLID, 1, RGB(0xFF, 0, 0));  
    KPen green(PS_SOLID, 1, RGB(0, 0xFF, 0));  
  
    red.Select(hDC);  
    // draw with red pen  
    red.UnSelect(hDC);  
  
    green.Select(hDC);  
    // draw with green pen  
    green.UnSelect(hDC);  
  
    ...  
    // both pens automatically deleted when this block ends  
}
```

8.4 LINES

Once a valid logical pen object handle is selected into a device context, the following GDI functions can be used to draw single or multiple straight lines, or prepare for line drawing:

```
BOOL MoveToEx(HDC hDC, int X, int Y, LPPOINT lpPoint);
BOOL LineTo(HDC hDC, int nXEnd, int nYEnd);
BOOL PolylineTo(HDC hDC, CONST POINT * lppt, DWORD cCount);
BOOL Polyline(HDC hDC, CONST POINT * lppt, int cPoints);
BOOL PolyPolyline(HDC hDC, CONST POINT * lppt,
                  CONST DWORD * lpdwPolyPoints, DWORD nCount);
```

`MoveToEx` does not draw lines; rather, it moves the current pen position in a device context to a new location (x, y) ; the original position is returned via `lpPoint`. Functions like `LineTo`, `PolylineTo`, `PolyBezierTo`, and even text display routines use the current pen position as their starting point. All the coordinates here are in logical coordinate space.

`LineTo` draws a single line segment from the current pen position to $(nXEnd, nYEnd)$ and sets the current pen position to $(nXEnd, nYEnd)$. All pen attributes described in the last section will determine how the line is drawn. Other device context attributes like world transformation, mapping, binary raster operation, background mode, background color, and miter limit may also be involved in drawing the line.

There are a few things to keep in mind. First, the pixel in the physical device coordinate space mapped to by $(nXEnd, nYEnd)$ is not painted, but the starting point is. For example, when you draw a line from $(0, 0)$ to $(100, 0)$ under identity mapping from logical to physical coordinate space, pixel $(0, 0)$ through $(99, 0)$ are painted, but not $(100, 0)$. In this case, $(100, 0)$ becomes the current pen position, so the next line starting from it paints it. If multiple connected lines are drawn using `LineTo`, each pixel on the line is painted exactly once, except the very last pixel. This is still true when the coordinate space is translated, scaled, rotated, or otherwise transformed. If binary raster operations like `R2_XORPEN` are used in drawing the line, the intersections of the line segment will look the same as other pixels. If GDI did draw the last pixel in `LineTo`, the intersection points will be drawn multiple times that turn them back to the background color. Because of this inclusive-exclusive rule, and for other reasons, line drawing is directional. That is, given two points (x_0, y_0) and (x_1, y_1) , drawing from (x_0, y_0) to (x_1, y_1) , and drawing from (x_1, y_1) to (x_0, y_0) yield slightly different lines. Another thing to remember is that every line-drawing call for a style pen starts a new styling pixel cycle for that style pen. GDI does not support aligning style lines across multiple calls; it is not like brush origin alignment. For example, given three points (x_0, y_0) , (x_1, y_1) , and (x_2, y_2) on a single straight line, drawing two lines $(x_0, y_0) - (x_1, y_1)$ and $(x_1, y_1) - (x_2, y_2)$ may yield different result from drawing a single line $(x_0, y_0) - (x_2, y_2)$.

The code shown below uses `MoveToEx` and `LineTo` to draw a rosette, a polygon with each vertex linked to every other vertex. The lines are colored according to the distance between two vertexes. DC pen is used here to switch pen color easily, which can be easily converted to a normal pen.

```
const int N = 19;
const int Radius = 200;
const double theta = 3.1415926 * 2 / N;

SelectObject(hDC, GetStockObject(DC_PEN));
```

```
const COLORREF color[] = {
    RGB(0, 0, 0), RGB(255,0,0), RGB(0,255,0), RGB(0,0, 255),
    RGB(255,255,0), RGB(0, 255, 255), RGB(255, 255, 0),
    RGB(127, 255, 0), RGB(0, 127, 255), RGB(255, 0, 127)
};

for (int p=0; p<N; p++)
for (int q=0; q<p; q++)
{
    SetDCPenColor(hDC, color[min(p-q, N-p+q)]);

    MoveToEx(hDC, (int)(220 + Radius * sin(p * theta)),
              (int)(220 + Radius * cos(p * theta)), NULL);
    LineTo(hDC, (int)(220 + Radius * sin(q * theta)),
           (int)(220 + Radius * cos(q * theta)));
}
```

PolyLineTo accepts a counted array of POINT structures, each with x and y coordinates. It draws the first line segment from the current pen position to the first POINT in the array, and then one line segment for each subsequent POINT in the array. The current pen position is set to be the last point. PolyLine accepts the same parameters as PolyLineTo, but its meaning is a little different. It does not use and update the current pen position. PolyLine draws a line segment from the first point in the array to the second point, and then one line segment for each subsequent POINT in the array. For an array with n POINTs, PolyLineTo draws n line segments, and PolyLine draws $n - 1$ line segments.

PolyLineTo and PolyLine can't simply be replaced by calls to MoveToEx and LineTo. If a nonsolid style pen is used, PolyLineTo and PolyLine continue to use the same style cycle across multiple segments of a line, while multiple LineTo calls start their independent cycles. If a geometric pen is used, the end cap attribute is applied to the first and last points, and the join attribute is applied to each join. With PolyLineTo and PolyLine, an application can create a nicer join, which can't be achieved easily using LineTo. This is especially important during zoomed-in displays or in printing, where wide pens are frequently used. As a visual comparison, [Figure 8-7](#) uses multiple LineTo calls, while [Figure 8-8](#) uses a single PolyLine call. There are also performance differences between calling GDI multiple times and calling it once for multiple line segments. When output to a metafile device context, PolyLineTo and PolyLine take less space than multiple LineTo calls.

But PolyLineTo and PolyLine are not perfect; they don't have the concept of a closed graph. An end cap is always applied to the first and last points drawn, even if their coordinates are exactly the same. If you are drawing geometric shapes with angles all multiples of 90 degrees, a square end cap and a miter join draw the connections and close the shape nicely. Or you can always use a round end cap and a round join to draw closed shapes with rounded corners. But if you are drawing an arbitrary triangle or polygon, you would prefer the corners to be sharp. Miter join can take care of all the joins except at the first point, which is also the last point. One trick is to add an extra line segment from the first point to the second point and the end. Even if you are using a binary raster operation, in which drawing once and twice make a difference, a figure drawn using a single PolyLineTo or PolyLine call will not have pixels drawn twice.

The following routine uses PolyLine to draw a triangle, with an option to add an extra segment to make the closing point nicer:

```
void Triangle(HDC hDC, int x0, int y0, int x1, int y1,
             int x2, int y2, bool extra=false)
{
```

```
POINT corner[5] = {x0,y0, x1,y1, x2,y2, x0,y0, x1,y1};
```

```
if ( extra ) // extra segment for nicer closing  
    Polyline(hDC, corner, 5);  
else  
    Polyline(hDC, corner, 4);  
}
```

PolyPolyline draws multiple polylines in a single call. Its last parameter nCount is the count of polylines instead of vertexes. Its third parameter lpdwPolyPoints is an array of vertex counts for each polyline. PolyPolyline draws the shape as a whole; it does not simply call Polyline multiple times for each polyline in it. The difference is visible when multiple polylines are overlapping with each other and raster operations like R2_XORPEN are used. If the shape is drawn in one piece, every pixel is painted only once; otherwise, the overlapping pixels may be painted multiple times, generating different colors than those of the pixels painted once.

On a raster-based device, such as a screen display or a printer, the line drawing uses a class of algorithms called DDA (digital differential analyzer) to select the pixels to paint to represent a line. The Bresenham algorithm is a classical incremental DDA algorithm. The Windows NT/2000 graphics engine uses 28.4 fixed-point coordinate space on physical device surfaces. Lines are drawn following the so-called GIQ (Grid Intersection Quantization) diamond convention, in which each pixel is thought of as having a 1-by-1 pixel size diamond shape around it. A pixel is painted if its diamond shape touches a line segment.

GDI provides a function LineDDA that allows an application-specified routine to receive the coordinates of each point GDI is supposed to draw.

```
BOOL LineDDA(int nXStart, int nYStart, int nXEnd, int nYEnd,  
            LINEDDAPROC lpLineProc, LPARAM lpData);
```

The prototype for LineDDA does not look too exciting. There is neither a device context nor a logical pen. So LineDDA can only return points in the same coordinate space as the parameters, with no consideration of the physical device coordinate space and the pen styles.

[Listing 8-2](#) shows how LineTo, Polyline and LineDDA can be used to draw equilateral triangles. Its result is shown in [Figure 8-9](#). Because a wide geometric pen is used, the first graph using LineTo has an ugly join, the second graph using Polyline with 4 points has an ugly closing join, and the third graph adds an extra segment for a perfect closing join. The fourth graph shows how LineDDA can be used to place small triangles along the baseline of the triangle. The callback routine LineDDAProc draws a small triangle for every 32nd call.

Figure 8-9. Drawing triangle using LineTo, Polyline, and LineDDA.



Listing 8-2 Line Drawing Using LineTo, Polyline, and LineDDA

```
void CALLBACK LineDDAProc(int x, int y, LPARAM lpData)
{
    HDC hDC = (HDC) lpData;
    POINT cur;

    GetCurrentPositionEx(hDC, & cur);
    if ( (cur.x & 31) == 0 ) // every 32th call
        Triangle(hDC, x, y-16, x+20, y+18, x-20, y+18);

    cur.x++;
    MoveToEx(hDC, cur.x, cur.y, NULL);
}

void KMyCanvas::TestLine2(HDC hDC)
{
    LOGBRUSH logbrush = {BS_SOLID, RGB(0, 0, 0xFF), 0 };

    HPEN hPen = ExtCreatePen(PS_GEOMETRIC | PS_SOLID |
        PS_ENDCAP_FLAT | PS_JOIN_MITER,
        15, & logbrush, 0, NULL);

    HGDIOBJ hOld = SelectObject(hDC, hPen);

    // drawing triangle using mutlple LineTo calls
    SetViewportOrgEx(hDC, 100, 50, NULL);
    MoveToEx(hDC, 0, 0, NULL); LineTo(hDC, 50, 86);
    LineTo(hDC, -50, 86); LineTo(hDC, 0, 0);

    // using polyline
    SetViewportOrgEx(hDC, 230, 50, NULL);
    Triangle(hDC, 0, 0, 50, 86, -50, 86, false );

    // using polyline, extra segment
    SetViewportOrgEx(hDC, 360, 50, NULL);
    Triangle(hDC, 0, 0, 50, 86, -50, 86, true );

    // using LineDDA
    SetViewportOrgEx(hDC, 490, 50, NULL);
    SelectObject(hDC, hOld);
    DeleteObject(hPen);

    hPen = ExtCreatePen(PS_GEOMETRIC | PS_DOT | PS_ENDCAP_ROUND,
        3, & logbrush, 0, NULL);
    SelectObject(hDC, hPen);

    LineDDA( 0, 0, 50, 86, LineDDAProc, (LPARAM) hDC);
    LineDDA( 50, 86, -50, 86, LineDDAProc, (LPARAM) hDC);
    LineDDA(-50, 86, 0, 0, LineDDAProc, (LPARAM) hDC);
    SetViewportOrgEx(hDC, 0, 0, NULL);
```

```
SelectObject(hDC, hOld);
DeleteObject(hPen);
}
```

[< BACK](#) [NEXT >](#)

8.5 BEZIER CURVES

Given two points P1 and P2 in two-dimensional coordinate space, there is a line segment defined linking the two points together. Let t be a variable between 0 and 1; we can define a point $P_{12}(t)$ on the line segment $P1 \rightarrow P2$ as:

$$P_{12}(t) = (1-t)P1 + tP2$$

Line segment $P1 \rightarrow P2$ is the trace of $P_{12}(t)$ when t varies from 0 to 1.

If we add another point P3 into the picture, we can define $P_{12}(t)$ as a point between P1 and P2, and $P_{23}(t)$ to be a point between P2 and P3. Now if we apply the same method to define $P_{123}(t)$ as a point between $P_{12}(t)$ and $P_{23}(t)$, we get:

$$P_{12}(t) = (1-t)P1 + tP2$$

$$P_{23}(t) = (1-t)P2 + tP3$$

$$\begin{aligned} P_{123}(t) &= (1-t)((1-t)P1 + tP2) + t((1-t)P2 + tP3) \\ &= (1-t)^2P1 + 2t(t-1)P2 + t^2P3 \end{aligned}$$

The trace of $P_{123}(t)$ when t varies from 0 to 1 is not a straight line any more; it's a quadratic curve, or a second-degree curve, and it is a parabola. Such curves were developed by P. de Casteljau in 1959, and later independently by P. Bezier in 1962, while working on Citroen and Renault automobile CAD design systems. Bezier first made the curve design known to the public, so such curves are now called Bezier curves.

Quadratic Bezier curves are used in TrueType fonts to define the outline of glyphs. Computer graphics commonly uses curves defined by four points, P1, P2, P3, and P4, following the same process. Such curves are naturally called cubic Bezier curves. [Figure 8-10](#) illustrates the construction of a cubic Bezier curve. Here are the formulae:

$$P_{12}(t) = (1-t)P1 + tP2$$

$$P_{23}(t) = (1-t)P2 + tP3$$

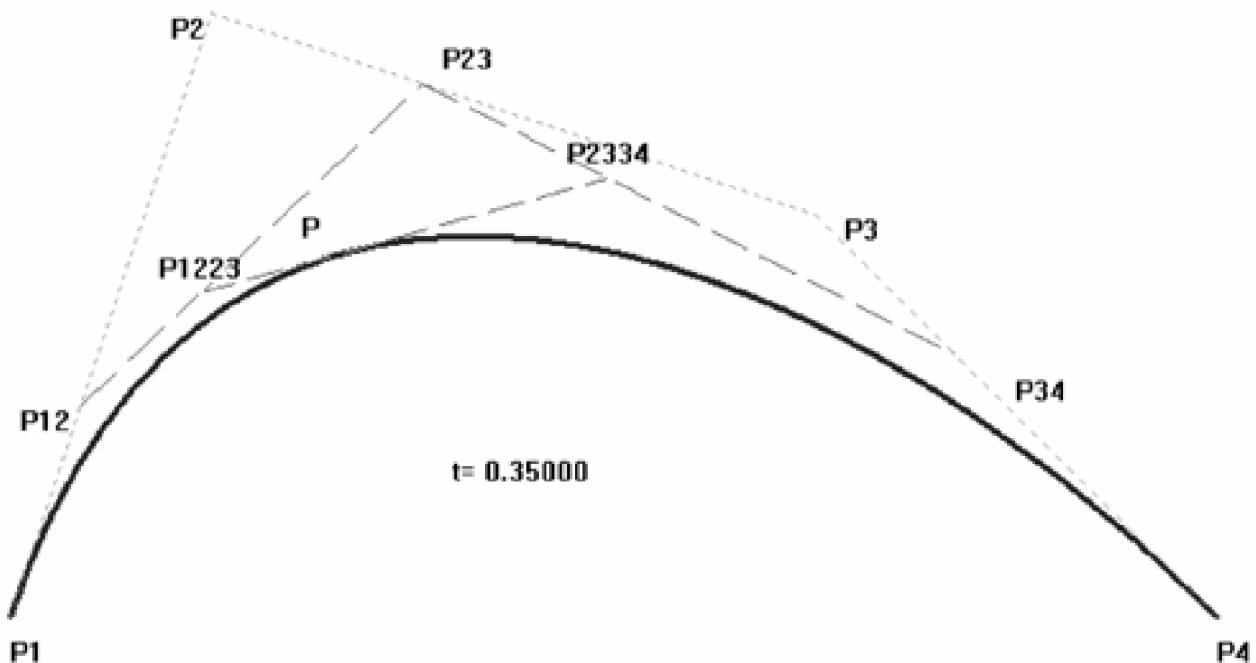
$$P_{34}(t) = (1-t)P3 + tP4$$

$$P_{123}(t) = (1-t)^2P1 + 2t(t-1)P2 + t^2P3$$

$$P_{234}(t) = (1-t)^2P2 + 2t(t-1)P3 + t^2P4$$

$$P(t) = (1-t)^3P1 + 3(1-t)^2tP2 + 3(1-t)t^2P3 + t^3P4$$

Figure 8-10. Constructing a cubic Bezier curve from four control points.



The construction process shown in [Figure 8-10](#) is normally called the de Casteljau algorithm. The points P1, P2, P3, and P4 defining a Bezier curve are called its control points. Sometimes P1 and P4 are referred to as endpoints, and P2 and P3 as off-line control points. Bezier curves have some interesting properties that make them popular in computer graphics computer-aided design and computer-aided manufacturing.

- **Endpoint Interpolation.** A Bezier curve always passes through its first and last control points, but not necessarily other control points.
- **Affine Invariant.** A Bezier curve is mapped to a Bezier curve under an affine transformation, which is used by GDI for world-coordinate-space to page-coordinate-space transformation. So the graphics engine needs only to transform the control points and draw the curve in device coordinate space using the transformed control points.
- **Convex Hull Bounded.** A Bezier curve always lies fully within the convex hull defined by its control points.
- **Tangent at Endpoints.** The line connecting P1 and P2 has the same tangent as the curve at P1; the line connecting P3 and P4 has the same tangent as the curve at P4. If two connected Bezier curves (P1, P2, P3, P4) and (P4, P5, P6, P7) are to have a smooth link (first derivative continuous), just make sure P3, P4, and P5 are on the same line.
- **Divisibility.** A Bezier curve can be easily divided into two Bezier curves. The curve shown [Figure 8-10](#) can be divided into two curves joining at P; the first curve is defined by (P1, P12, P1223, P), and the second curve is defined by (P, P2334, P34, P4).

The divisibility property brings an algorithm to draw a Bezier curve using straight-line drawing calls. We just need to divide a Bezier curve at its midpoint ($t = 0.5$) and then recursively divide the two subcurves until their control points are close enough to be drawn using straight lines. [Listing 8-3](#) shows a recursive routine to draw a Bezier curve using lines. The routine first checks if (x_2, y_2) and (x_3, y_3) is less than a single unit away from the line $(x_1, y_1) - (x_4, y_4)$. If so, the line is drawn; otherwise, the curve is divided in the middle into two curves, and two recursive calls are made to handle them. Floating-point numbers are used for accuracy.

Listing 8-3 Drawing a Bezier Curve Using Lines

```
void Bezier(HDC hDC, double x1, double y1, double x2, double y2,
            double x3, double y3, double x4, double y4)
{
    double A = y4 - y1;
    double B = x1 - x4;
    double C = y1 * (x4-x1) - x1 * (y4-y1);
    // Ax + By + C = 0 is the line (x1,y1) - (x4,y4)

    double AB = A * A + B * B;

    // distance from (x2,y2) to the line is less than 1
    // distance from (x3,y3) to the line is less than 1
    if ( (A * x2 + B * y2 + C) * (A * x2 + B * y2 + C) < AB )
    if ( (A * x3 + B * y3 + C) * (A * x3 + B * y3 + C) < AB )
    {
        MoveToEx(hDC, (int)x1, (int)y1, NULL);
        LineTo(hDC, (int)x4, (int)y4);

        return;
    }

    double x12 = x1+x2;
    double y12 = y1+y2;
    double x23 = x2+x3;
    double y23 = y2+y3;
    double x34 = x3+x4;
    double y34 = y3+y4;

    double x1223 = x12+x23;
    double y1223 = y12+y23;
    double x2334 = x23+x34;
    double y2334 = y23+y34;

    double x = x1223 + x2334;
    double y = y1223 + y2334;

    Bezier(hDC, x1, y1, x12/2, y12/2, x1223/4, y1223/4, x/8, y/8);
    Bezier(hDC, x/8, y/8, x2334/4, y2334/4, x34/2, y34/2, x4, y4);
}
```

Now let's look at the two GDI functions to support Bezier curves.

```
BOOL PolyBezier (HDC hDC, CONST POINT * lppt, DWORD cPoints);
BOOL PolyBezierTo(HDC hDC, CONST POINT * lppt, DWORD cCount);
```

Both functions draw multiple Bezier curves in a single call. To draw n curves, PolyBezier needs $3n + 1$ points in the array pointed to be lppt; cPoints should be $3*n + 1$. The first four points lppt[0], lppt[1], lppt[2], and lppt[3] define the first curve, then lppt[3] and three points after that define the second curve, and so on. PolyBezier does not use and update the current pen position. PolyBezierTo uses the current pen position and updates it with the last point passed. To draw n curves, PolyBezier needs $3n$ points.

[Figure 8-10](#) shows a normal Bezier curve, which has two off-line control points on the same side of the P1 ? P4 line; x-coordinates for all the points go by increments. We can change the positions of the off-line control points to change the curve dramatically. [Listing 8-4](#) shows a routine that uses the PolyBezier function to draw five series of Bezier curves.

Listing 8-4 Use PolyBezier to Draw Bezier Curve Series

```
HPEN hRed = CreatePen(PS_DOT, 0, RGB(0xFF, 0, 0));
HPEN hBlue = CreatePen(PS_SOLID, 3, RGB(0, 0, 0xFF));

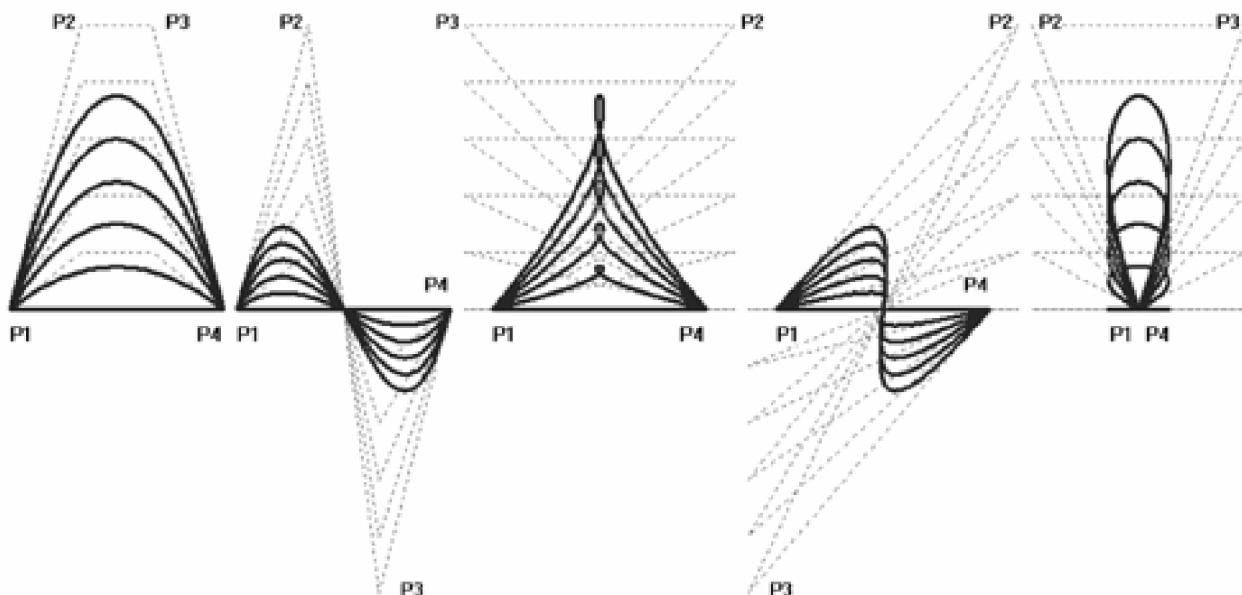
for (int z=0; z<=200; z+=40)
{
    int x = 50, y = 240;
    POINT p[4]={x,y, x+50,y-z, x+100,y-z, x+150,y}; x+= 160;
    POINT q[4]={x,y, x+50,y-z, x+100,y+z, x+150,y}; x+= 180;
    POINT r[4]={x,y, x+170,y-z, x-20,y-z, x+150,y}; x+= 200;
    POINT s[4]={x,y, x+170,y-z, x-20,y+z, x+150,y}; x+= 180;
    POINT t[4]={x+75,y, x,y-z, x+150,y-z, x+75,y};

    SelectObject(hDC, hRed);
    Polyline(hDC, p, 4); Polyline(hDC, q, 4);
    Polyline(hDC, r, 4); Polyline(hDC, s, 4);
    Polyline(hDC, t, 4);

    SelectObject(hDC, hBlue);
    PolyBezier(hDC, p, 4); PolyBezier(hDC, q, 4);
    PolyBezier(hDC, r, 4); PolyBezier(hDC, s, 4);
    PolyBezier(hDC, t, 4);
}
SelectObject(hDC, GetStockObject(BLACK_PEN));
DeleteObject(hRed);
DeleteObject(hBlue);
```

The code shows how different-looking Bezier curves can be defined. The first group of curves has control points on the same side, while the second group has them on two opposite sides. The x-coordinates for the two control points in the third group are swapped, the fourth group is a combination of the second and the third, and the last group has the two endpoints at the same position. [Figure 8-11](#) shows the display result with some extra labels.

Figure 8-11. Bezier curve gallery.



As far as pens are concerned, PolyBezier and PolyBezierTo are like Polyline and PolylineTo. Simple pens use background mode and background color for styled lines. Geometric and cosmetic pens always draw in transparent mode, ignoring background mode and color. When drawing multiple curves, the join attribute affects the joins, end caps are applied to the starting and finishing points, and the whole call is drawn as a single operation without drawing anything twice.

PolyDraw

PolyBezier and PolyBezierTo draw only continuous Bezier curves. On Windows NT/2000, GDI provides a new powerful function, PolyDraw, which draws multiple disjoint lines, Bezier curves, and even closed shapes.

```
BOOL PolyDraw(HDC hDC, CONST POINT * lppt,  
CONST BYTE * lpbTypes, int nCount);
```

PolyDraw accepts two counted arrays: lppt is an array of points, lpbTypes is an array of point types, one for each point in the first array. [Table 8-4](#) lists the five allowed point types.

Table 8-4. Point Types in PolyDraw and GetPath

Type	Meaning
PT_MOVETO	Start a new disjoint figure. This point becomes the current position.
PT_LINETO	Draw a line from the current position to this point, update the current position.
PT_LINETO PT_CLOSEFIGURE	Same as PT_LINETO, except add a line to the last PT_MOVETO position to close the figure.
PT_BEZIERTO	Draw a Bezier curve using the current position and three points starting from this point. The next two points must have a PT_BEZIERTO flag. Update current position with the third point.
PT_BEZIERTO PT_CLOSEFIGURE	Only for the last point in a PT_BEZIERTO, add a line to the last PT_MOVETO position to close the figure.

Several features make PolyDraw such a powerful function. First, straight lines and Bezier curves can be mixed together. Although every line can be converted to a Bezier curve by adding two more points between the endpoints, it would not be natural to do so. Second, it allows drawing of multiple disjoint figures in a single call. We mentioned before that drawing multiple lines or curves in a single call makes sure no pixel is drawn twice, which is essential for dragging a complex figure using the R2_XORPEN raster operation in a graphics package. Third, PolyDraw introduces the concept of a closed figure. A closed figure automatically adds a line to its starting point, so it's a handy feature for applications. Besides, for geometric pens a closed figure does not use end caps; it uses a join even at the closing point to make the whole curve look seamless.

[Listing 8-5](#) shows a routine that uses two different methods to draw a combined 8 shape and a diamond shape. The code uses a wide solid geometric pen with a flat end cap and miter join, drawing using R2_XORPEN. The first part draws the 8 shape using PolyBezier and the diamond shape using Polyline. Although both shapes are closed, the closing points have visible gaps. The overlapping part of the two curves is turned white. The second part draws the two closed figures using a simple PolyDraw call, which solves both problems. [Figure 8-12](#) shows the display result.

Figure 8-12. Drawing closed figures with and without PolyDraw.

Listing 8-5 Drawing Closed Figures with and without PolyDraw

```
const POINT P[12] =
{ 50, 200, 100, 50, 150, 350, 200, 200,
 150, 50, 100, 350, 50, 200,
 125, 275, 175, 200, 125, 125, 75, 200, 125, 275
};

const BYTE T[12] =
{ PT_MOVETO, PT_BEZIERTO, PT_BEZIERTO, PT_BEZIERTO,
  PT_BEZIERTO , PT_BEZIERTO, PT_BEZIERTO | PT_CLOSEFIGURE,
  PT_MOVETO, PT_LINETO, PT_LINETO, PT_LINETO,
  PT_LINETO | PT_CLOSEFIGURE
};

SetROP2(hDC, R2_XORPEN);

LOGBRUSH logbrush = { BS_SOLID, RGB(0xFF, 0xFF, 0), 0 };
HPEN hPen = ExtCreatePen(PS_GEOMETRIC| PS_SOLID| PS_ENDCAP_FLAT|
    PS_JOIN_MITER, 15, & logbrush, 0, NULL);
SelectObject(hDC, hPen);

PolyBezier(hDC, P, 7); // two Bezier curves
Polyline(hDC, P+7, 5); // diamond shape

SetViewportOrgEx(hDC, 200, 0, NULL);
PolyDraw(hDC, P, T, 12); // two shapes together

SetViewportOrgEx(hDC, 0, 0, NULL);
SelectObject(hDC, GetStockObject(BLACK_PEN));
DeleteObject(hPen);
```

Since it is such a powerful function, not having PolyDraw available on Windows 95/98 is really a headache. MSDN Knowledge offers an implementation of PolyDraw on Windows 95 in article Q135059. The suggested code implements PolyDraw by calling MoveToEx, LineTo, and PolyBezierTo. The code has quite a few problems. It does not implement PT_CLOSEFIGURE, uses multiple GDI drawing calls that may paint things multiple times, and uses a geometric pen end cap; join would not work on Windows 95/98 because it's not part of a GDI path object. The correct implementation should simply use the GDI path feature that supports multiple disjoint closed figures, with end cap and join. We will discuss path objects later in this chapter.

Alternative Bezier Curve Definition: Pass All Points

The normal Bezier curve definition uses two endpoints and two off-line control points. The definition process is very intuitive geometrically, so it's quite easy for interactive manipulation. But for programmers, it's much easier to define a curve using control points that are all on the curve, with no off-curve control points. For a cubic Bezier curve, a commonly asked question is how to define a Bezier curve using four points A, B, C, D, the curve passing A when $t=0$, B when $t=1/3$, C when $t=2/3$, and D when $t=1$.

The problem really comes down to calculating the four points P1, P2, P3, and P4 according to A, B, C, D, where P1 and P4 are endpoints, and P2 and P3 are off-line control points. According to the parametric definition of a Bezier curve, we have the following linear equation system:

$$A = P1$$

$$B = (2/3)^3 P1 + 3(2/3)^2(1/3) P2 + 3(2/3)(1/3)^2 P3 + P4$$

$$C = (1/3)^3 P1 + 3(1/2)^2(2/3) P2 + 3(1/3)(2/3)^2 P3 + P4$$

$$D = P4$$

Points P1 and P2 are easy. The equations for B and C can be transformed to:

$$12P2 + 6P3 = 27B - 8A - D$$

$$6P2 + 12P3 = 27C - A - 8D$$

Solving for P2 and P3, we get the whole solution:

$$P1 = A$$

$$P2 = (-5A + 18B - 9C + 2D) / 6$$

$$P3 = (2A - 9B + 18C - 5D) / 6$$

$$P4 = D$$

The formulae also serve as a constructive proof that for any four points, there exists a Bezier curve that passes through them at $t = 0, 1/3, 2/3$, and 1.

[< BACK](#) [NEXT >](#)

8.6 ARCS

The most famous curves may be ellipses, of which circles are special cases. The simplest form of the ellipse has axes parallel to the axes in the coordinate space. Its equation is

$$(x-x_0)^2/a^2 + (y-y_0)^2/b^2 = 1$$

where (x_0, y_0) is the center and a and b are its major and minor axes.

GDI's method of defining an ellipse, or any shape based on an ellipse, is very close to this simple equation. GDI defines an ellipse by its bounding box—that is, the rectangle $(x_0 - a, y_0 - b, x_0 + a, y_0 + b)$.

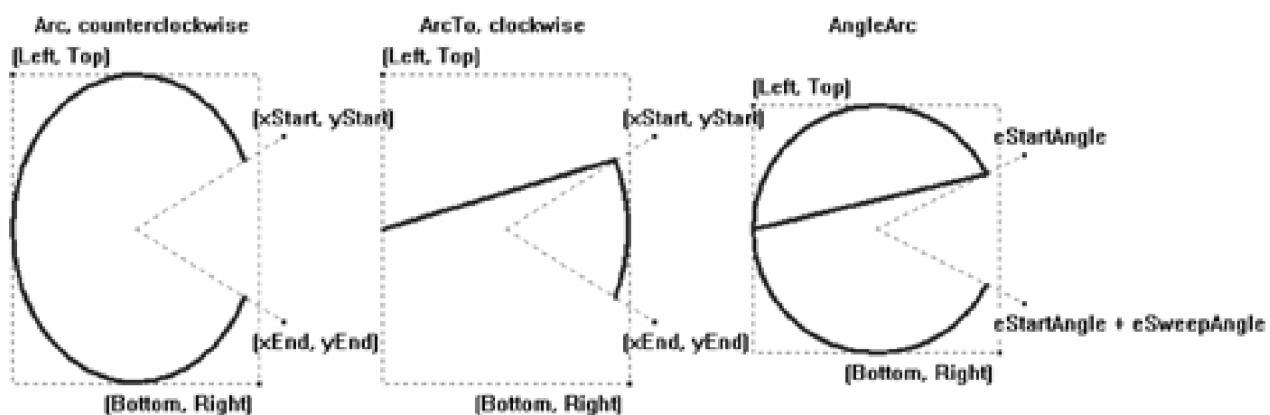
An arc can be either a full ellipse perimeter or part of it. The most intuitive way to specify part of an ellipse's perimeter is to use a starting and an ending angle; the curve is defined to be the trace of a point on the ellipse when the angle moves from the starting angle to the end angle. But in order to avoid floating points and still provide the required accuracy, GDI uses two points $(xStart, yStart)$ and $(xEnd, yEnd)$ from which the starting and end points of an arc can be calculated. The starting point is the intersection of the line connecting the ellipse center and $(xStart, yStart)$ with the perimeter; likewise, the end point is the intersection of the line connecting the ellipse center and $(xEnd, yEnd)$ with the perimeter.

Here are the three GDI functions to draw an arc:

```
BOOL Arc(HDC hDC, int nLeft, int nTop, int nRight, int nBottom,  
         int nXStart, int nYStart, int nXEnd, int nYEnd);  
BOOL ArcTo(HDC hDC, int nLeft, int nTop, int nRight, int nBottom,  
           int nXStart, int nYStart, int nXEnd, int nYEnd);  
BOOL AngleArc(HDC hDC, int X, int Y, DWORD dwRadius,  
               FLOAT eStartAngle, FLOAT eSweepAngle);
```

For Arc and ArcTo, nLeft, nTop, nRight, and nBottom specify the bounding box for the ellipse, of which the arc is part. The center of the ellipse is in the center of the box. The four parameters after that are used to calculate the start and end angles of the arc. The function Arc draws an arc from the starting angle to the end angle. On Windows 95/98, the arc is drawn counterclockwise; on Windows NT/2000, the device context's arc direction flag (GetArcDirection, SetArcDirection) controls the direction of the arc. The first two arcs in [Figure 8-13](#) illustrate the parameters of Arc and ArcTo, together with the impact of arc direction.

Figure 8-13. Defining an arc the GDI way.



The Arc function does not use or update the current pen position. ArcTo is a little different. It uses the current pen position to draw a line to the real starting point of the arc and then draws the arc. After finishing the arc, it sets the current pen position with the real ending point of the arc.

With Arc or ArcTo, you can also draw a full ellipse perimeter; just set the end reference point to be the same as the start reference point. To draw the complement part of the perimeter, you need to swap the start and end reference points. In case you want to calculate the actual start or end point, here is the formula (the start point is shown here):

```
X0 = (nLeft + nRight)/2; // center of the ellipse  
Y0 = (nTop + nBottom)/2;  
DXs = nXStart - X0;  
DYs = nYStart - Y0;  
Ds = sqrt(DXs * DXs + DYs * DYs); // dis ref to center  
Xs = X0 + (nRight-nLeft)/2 * DXs / Ds;  
Ys = Y0 + (nBottom-nTop)/2 * Dys / Ds;
```

Specifying Arc Using Degrees: AngleArc

The AngleArc function, available only on NT-based OSs, departs from the no-floating-point party line. It accepts an arc's start angle and sweep angle in degrees (not in radians) as floating-point numbers. So the sweep angle tells both how large the arc angle is and its direction, and the device context's arc direction attribute is not needed here. Another oddity about AngleArc is that only the radius of a circle can be specified in logical coordinate space. To draw part of an ellipse, an application has to set up the right transformation or mapping. AngleArc uses the current pen position to draw a line segment to the arc start position and updates it with the end position, just like ArcTo. So you may wonder why it's not named AngleArcTo. With AngleArc, it's very easy to draw a full circle: just draw 360 degrees. But what's a 540-degree arc? MSDN claims that if the sweep angle is more than 360 degrees, the arc is swept multiple times. This implies that a 540-degree arc using the R2_XORPEN background mode draws a 360-degree full circle first, then sweeps 180 degrees more to restore the original background, so that in the end you get a 180-degree arc. Our experiment shows that when the sweep angle is more than 360 degrees, a full circle is drawn only once.

AngleArc can be quite easily implemented using ArcTo, which will be useful for non-NT based platforms. Here is a try:

```
BOOL AngleArcTo(HDC hDC, int X, int Y, DWORD dwRadius,
```

```
FLOAT eStartAngle, FLOAT eSweepAngle)
{
const FLOAT piover180 = (FLOAT) 3.141592653586/180;

if ( eSweepAngle >=360 ) // more than a circle is a circle
    eSweepAngle = 360;
else if ( eSweepAngle <= -360 )
    eSweepAngle = - 360;

FLOAT eEndAngle = (eStartAngle + eSweepAngle ) * piover180;
eStartAngle = eStartAngle * piover180;

int dir; // sweepangle determines direction
if ( eSweepAngle > 0 )
    dir = SetArcDirection(hDC, AD_COUNTERCLOCKWISE);
else
    dir = SetArcDirection(hDC, AD_CLOCKWISE);

// angle is based on device coordinate space
BOOL rslt = ArcTo(hDC, X - dwRadius, Y - dwRadius,
                  X + dwRadius, Y + dwRadius,
                  X + (int) (dwRadius * 10 * cos(eStartAngle)),
                  Y - (int) (dwRadius * 10 * sin(eStartAngle)),
                  X + (int) (dwRadius * 10 * cos(eEndAngle)),
                  Y - (int) (dwRadius * 10 * sin(eEndAngle)));
SetArcDirection(hDC, dir);
return rslt;
}
```

The routine makes sure that no more than one full circle is drawn, converts degrees to radians, sets the arc direction according to the sweep angle sign, calculates the start and end reference points using 10 times the circle radius for accuracy, and then draws the arc using the ArcTo function.

Drawing Arcs with Inside Frame Pens

With arcs, which are bounded by a rectangle, GDI finally honors the inside frame pen style. When a pen with PS_INSIDEFRAME is used to draw an arc, the center line of the curve is moved inward by half the pen width. The outmost pixels touch the bounding rectangle, and out pixels are drawn outside the rectangle. We should know how easy it is for GDI to implement the inside frame pen for arcs; GDI just needs to shrink the bounding box by half the pen width. Doing that for a generic curve—for example, a closed Bezier curve sequence—is much harder.

The design of the inside frame pen is peculiar. PS_INSIDEFRAME is a pen style on the same level as PS_SOLID and PS_DOT, not a flag like the end cap and join attributes. The end result of this design is that an inside frame pen is always a solid pen. Inside frame should have been an independent pen attribute.

Convert Arcs to Bezier Curves

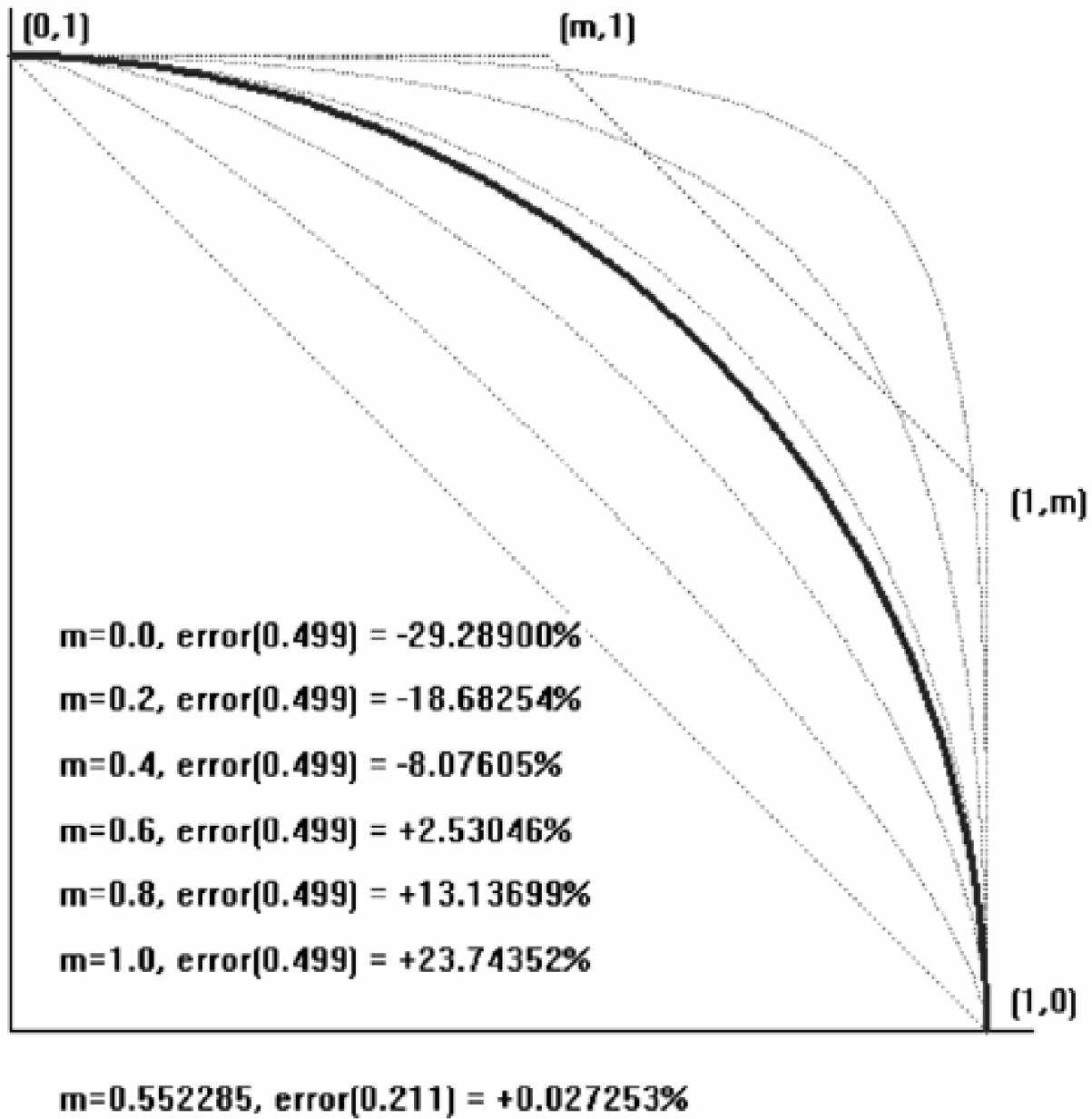
We should be aware by now that arcs are hard to deal with; you need to use floating-point computation and struggle with the confusing arc direction concept to figure out which part of the perimeter you want to draw. What's even worse is that you can draw only part of an ellipse whose axes are parallel to the logical coordinate system axes. If you want to draw a rotated arc or a sheared arc, you have to figure out the right transformation, which is a feature not supported on non-NT-based OSs.

Here we need Bezier curves to solve the problem. Bezier curves are extremely easy to manipulate. Drawing them involves only simple calculation. Affine transformations still map Bezier curves to Bezier curves, and you would never lose direction, because four points define one curve. The only question is, how do you design the right Bezier curve to approximate an elliptic arc?

Using a single Bezier curve to approximate a full circle is very inaccurate. One Bezier curve for half a circle is possible but it still may not be accurate enough. So let's try to deduce the control points for a Bezier curve to approximate a quarter of a circle, or 90 degrees. For a quarter of a unit circle in the first quadrant in Cartesian coordinate space, we know the endpoints $(0, 1)$ and $(1, 0)$. The 90-degree arc is symmetric to the $X = Y$ line, so the two control points need to be symmetric, too. We know that the line from the first endpoint to the first control point is tangent to the arc at the first endpoint. This means the line should be at 0 degrees, or the first control point is $(m, 1)$ for an unknown variable m . From the symmetric property, the second control point is $(1, m)$. Now we know all four control points $P1:(0, 1)$, $P2:(m, 1)$, $P3:(1, m)$ and $P4:(1, 0)$ with a single unknown variable m to solve.

[Figure 8-14](#) shows a 90-degree arc from the unit circle in the first quadrant. The light color lines link the four control points of the Bezier curve we are trying to find. The six light color curves show the Bezier curve approximations when m varies from 0 to 1.0 at the 0.2 increment. The dark color curve is the target arc curve.

Figure 8-14. Convert 90-degree arc to Bezier curve.



The midpoint of the curve is given by substituting $t = 0.5$ into the formula in the last section, or:

$$\begin{aligned}P(0.5) &= (1-t)^3 P_1 + 3(1-t)^2 t P_2 + 3(1-t)t^2 P_3 + t^3 P_4 \\&= (P_1 + 3P_2 + 3P_3 + P_4)/8\end{aligned}$$

Using this to calculate the midpoint of the curve $(0, 1)$, $(m, 1)$, $(1, m)$ and $(1, 0)$, we have:

$$\begin{aligned}X_{\text{mid}} &= (0+3m+3+1)/8 = (3m+4)/8 \\Y_{\text{mid}} &= (1+3+3m+1)/8 = (3m+4)/8\end{aligned}$$

We know $(X_{\text{min}}, Y_{\text{mid}})$ of a quarter circle in a unit circle is $(\cos(1/4), \sin(1/4))$, or $(\sqrt{2}/2, \sqrt{2}/2)$, so we have a solution $m = 4(\sqrt{2} - 1)/3$, or roughly $m = 0.552285$.

If we use four such Bezier curves to approximate a full ellipse, at any angle multiple of 45 degrees the Bezier curves match the circle exactly. The remaining question is how to close the rest of the positions? By changing t from 0 to 0.5 at a small increment, we can get the coordinates of points on the Bezier curve, calculate their distance from the origin, and make a comparison with the unit circle. The largest relative error is 0.027253%, achieved when $t = 0.211$. So how much is the error in pixels? If you are drawing a full-screen-size ellipse, which normally means no more than 1600×1200 pixels, using the four-Bezier-curve approximation misses the true curve by 0.436 pixel in the worst case.

[Listing 8-6](#) shows two routines. The first routine, EllipseToBezier, draws a full ellipse using the Bezier-curve approximation. The routine calculates 13 control points for four Bezier-curve segments according to the method described above. The second routine, AngleArcToBezier, draws an arc with an arbitrary starting and sweep angle, using a single Bezier curve.

Listing 8-6 Drawing Full Ellipse Perimeter Using Bezier Curves

```
BOOL EllipseToBezier(HDC hDC, int left, int top,
                     int right, int bottom)
{
    const double M = 0.55228474983;
    POINT P[13];
    int dx = (int) ((right - left) * (1-M) / 2);
    int dy = (int) ((bottom - top) * (1-M) / 2);

    P[ 0].x = right;           // . . . .
    P[ 0].y = (top+bottom)/2; // 4 3 2
    P[ 1].x = right;          //
    P[ 1].y = top + dy;       // 5      1
    P[ 2].x = right - dx;    //
    P[ 2].y = top;            // 6      0,12
    P[ 3].x = (left+right)/2; //
    P[ 3].y = top;            // 7      11
                                //
    P[ 4].x = left + dx;     // 8 9 10
    P[ 4].y = top;
    P[ 5].x = left;
    P[ 5].y = top + dy;
    P[ 6].x = left;
    P[ 6].y = (top+bottom)/2;

    P[ 7].x = left;
    P[ 7].y = bottom - dy;
    P[ 8].x = left + dx;
    P[ 8].y = bottom;
    P[ 9].x = (left+right)/2;
    P[ 9].y = bottom;
    P[10].x = right - dx;
    P[10].y = bottom;
```

```
P[11].x = right;
P[11].y = bottom - dy;
P[12].x = right;
P[12].y = (top+bottom)/2;

return PolyBezier(hDC, P, 13);
}

BOOL AngleArcToBezier(HDC hDC, int x0, int y0, int rx, int ry,
                      double startangle, double sweepangle)
{
    double XY[8];
    POINT P[4];

    // Compute bezier curve for arc centered along x axis
    // Anticlockwise: (0,-B), (x,-y), (x,y), (0,B)
    double B = ry * sin(sweepangle/2);
    double C = rx * cos(sweepangle/2);
    double A = rx - C;

    double X = A*4/3;
    double Y = B - X * (rx-A)/B;

    XY[0] = C;
    XY[1] = -B;
    XY[2] = C+X;
    XY[3] = -Y;
    XY[4] = C+X;
    XY[5] = Y;
    XY[6] = C;
    XY[7] = B;

    // rotate to the original angle
    double s = sin(startangle + sweepangle/2);
    double c = cos(startangle + sweepangle/2);

    for (int i=0; i<4; i++)
    {
        P[i].x = x0 + (int) ((XY[i*2] * c - XY[i*2+1] * s) * rx);
        P[i].y = y0 + (int) ((XY[i*2] * s + XY[i*2+1] * c) * ry);
    }

    return PolyBezier(hDC, P, 4);
}
```

The arc-to-Bezier conversion method used by the EllipseToBezier routine can't handle an arbitrary arc, because it is designed to handle arcs having a 90-degree or a multiple -of-90-degree sweep angle. The AngleArcToBezier routine uses a new method to calculate a single Bezier curve for an arbitrary arc, although you should expect errors to increase when the sweep angle is more than 90 degrees. The routine first rotates the arc to be symmetric along the

x-axis, which has half the arc above $y = 0$ and half below. In this way, the formulae for the control points are much easier to figure out. After calculating the control points in the rotated position, it rotates them back to its original position.

The relative error of the calculated Bezier curve relative to the original arc varies greatly with the sweep angle. At 45 degrees, the error is 0.00042%; at 90 degrees, it grows to 0.02725%; and at 180 degrees, the error is 1.835%.

The real beauty of approximating arcs with Bezier curves is to be able to add transformation to the Bezier curves' control points, to draw irregular arcs without using GDI's world-transformation support, which is available only on Windows-NT-based platforms. You can also easily break Bezier curves into line segments to implement new line styles not supported by GDI or Windows 95/98 systems. One thing to notice is that when an arc is converted to Bezier curves, an inside frame pen is treated the same as a solid pen. If an inside frame drawing is intended, the bounding rectangle should be shrunk before calling conversion routines.

[< BACK](#) [NEXT >](#)

8.7 PATHS

When we draw lines and curves using GDI API, two pieces of information are needed. The first is the pen and the other device context related attributes that determine how lines and curves are drawn. The second is the geometric shape information—that is, specified points and their types. A nicely designed graphics system should allow these two different classes of information to be separated and manipulated individually for maximum flexibility. The GDI pen objects manage pens. The device context objects manage other line and curve drawing attributes. Now we need something to manage geometric shapes. This brings us to the GDI path objects.

A path is a GDI object that represents geometric shapes. To be more specific, a path is an ordered sequence of figures, either open or closed, where a figure is an ordered sequence of lines and curves. A path object is an object managed by GDI, so each path is an entry in the GDI handle table with a corresponding GDI handle. But unlike other commonly seen GDI objects, such as logical pen, device context, etc., path objects are hidden in GDI API. In GDI level, a path object is always attached to a device context and is never independent from it. Its creation, modification, selection, usage, and destruction are managed by GDI when certain path-related functions are called. Here are the GDI path-related functions:

```
BOOL BeginPath(HDC hDC);
BOOL EndPath(HDC hDC);
BOOL AbortPath(HDC hDC);
BOOL CloseFigure(HDC hDC);
int GetPath(HDC hDC, PPOINT pPoints, PBYTE pTypes, int nCount);
BOOL FlattenPath(HDC hDC);
BOOL WidenPath(HDC hDC);

BOOL StrokePath(HDC hDC);
BOOL StrokeAndFillPath(HDC hDC);
BOOL FillPath(HDC hDC);
HRGN PathToRegion(HDC hDC);
BOOL SelectClipPath(HDC hDC, int iMode);
```

Path Construction

A path needs to be constructed first before it can be used. BeginPath starts the construction of a path in a device context. It puts the device context in a path construction state and resets the implicit path object associated with it to an empty path. When a device context is in the path construction state, all allowed path construction function calls do not draw into the device surface; instead, they are appended to the implicit path object associated with the device context. EndPath is the closing call for path construction, which finishes path construction and puts the device context back into a drawing state. Now the device context has a valid path object selected, which can be used by path-consuming functions like StrokePath. AbortPath is also a closing call for path construction, but it discards any path object in the device context; thus no valid path is generated. Only the GDI functions that generate lines and curves can be used in constructing a path; other functions still go untouched to the device surface. [Table 8-5](#) summarizes the valid path construction functions. Note that area fill functions and even text output functions generate lines and curves, too, so they are included.

All functions listed in the table use lines and curves in certain ways, and they are added to the path object. The pixel drawing function—that is, SetPixel—does not generate lines, so it's not included. For line and curve draw functions like MoveToEx, LineTo, Polyline, PolylineTo, PolyPolyline, PolyBezier, PolyBezierTo, PolyDraw, Arc, ArcTo, and AngleArc, their behavior in path construction should be very easy to understand. Pen objects normally do not affect path construction, because the path is supposed to be a geometric shape only. But the arc drawing function with the inside frame pen selected in a device context is an exception. The bounding rectangle for the arc is shrunk by half the pen width. Note that the tricky thing here is that PolyDraw, Arc, ArcTo, and AngleArc are either not implemented or are not allowed in path construction on non-NT based platforms. To write a cross-platform program, an application could convert them to Bezier curves to incorporate them into a path.

Table 8-5. Path Construction Functions

Function	Description	Restriction
AngleArc, Arc, ArcTo	Add a line (except Arc), and an arc.	NT-based only
Ellipse, Chord, Pie	Add a full ellipse perimeter, one line and an arc, or two lines and an arc.	NT-based only
CloseFigure	Add the last point as closing point.	
ExtTextOut, TextOut	Add a character outline as individual closed figures.	Not for raster fonts
LineTo	Add a line.	
MoveToEx	Start a new figure.	
PolyBezier, PolyBezierTo	Add a line (PolyBezierTo only) and multiple Bezier curves.	
PolyDraw	Add a sequence of figures.	NT-based only
Polygon, PolyPolygon	Add one or more polygons as individual closed figures.	
Polyline, PolylineTo, PolyPolyline	Add one or more individual polylines.	
Rectangle	Add a rectangle as a new closed figure.	
RoundRect	Add a closed figure with four lines and four arcs.	NT-based only

The second class of path construction functions are area filling functions in GDI—for example, Ellipse, Chord, Pie, Polygon, PolyPolygon, Rectangle, and Round Rect. They normally fill the enclosed area using a brush and draw the outline using a pen. Their details will be covered in [Chapter 9](#). For now, we just need to remember that these functions do define a geometric shape, either one or multiple closed figures. When drawing into a device context in the path construction state, the geometric shape is added to the current path in the device context.

The third class of path construction functions is text output functions. Windows uses three types of fonts: raster fonts that use bitmaps, vector fonts that use lines, and TrueType fonts that use lines and Bezier curves. In the path construction state, text output calls with vector or TrueType fonts add glyph outlines to the current path.

Beside these three classes of functions, GDI throws in a special function, CloseFigure. CloseFigure does only one thing: It marks the last point as a closing point. The closing-point mark is a hint to add a line to the figure's starting point and use the pen join attribute instead of the end cap attribute for a geometric pen when connecting to the starting point.

Here is a quick example of path construction. The code draws two ellipses using the same coordinates, one with the default pen, one with a 21-unit-wide inside frame pen. If the default pen is not a 21-unit-wide inside frame pen, the

path constructed is made up of two concentric ellipses. If a solid pen was used, the two ellipses will be exactly the same.

```
BeginPath(hDC);
Ellipse(hDC, 0, 0, 100, 100);

HPEN hPen = CreatePen(PS_INSIDEFRAME, 21, RGB(0xFF, 0, 0));
HPEN hOld = (HPEN) SelectObject(hDC, hPen);

Ellipse(hDC, 0, 0, 100, 100);
SelectObject(hDC, hOld);
DeleteObject(hPen);
EndPath(hDC);
```

Querying Path Data

After calling EndPath, if a valid path is constructed, an application can query the contents of the path object using the GDI function GetPath. Within the graphics engine, a path object is quite a complex data structure, designed to limit memory consumption and to be easy to grow at the same time.

Note

[Chapter 3, Section 3.7](#), describes the details of the path data structure on the Windows NT/2000.

GetPath converts GDI's internal path representation to two arrays: an array of points and an array of flags. The point array stores the coordinates of all vertexes and control points defining the path. The flag array matches the point array in number of entries. For each point, the flag tells whether the point is a line point or a Bezier curve control point, a figure starting point or a closing figure point. The available flags are described in [Table 8-4](#). The data returned by GetPath is in exactly the same format as the data accepted by PolyDraw. But where do the elliptic curves go? There is no point flag for elliptic curves. They are converted to Bezier curves in a manner similar to that described in [Section 8.6](#).

GetPath returns points in logical coordinate space. Internally, points in a GDI path object are stored in device coordinate space in fixed-point notation for maximum accuracy. When GetPath is called, GDI uses the inverse matrix of the world-to-device-coordinate-space transformation to calculate the path points in logical coordinate space. Note that logical coordinate space is represented using integers. So if there is a big scale factor from logical to device coordinate space, the data returned by GetPath may lose precision.

Calling GetPath is tricky, because the caller needs to allocate storage for two arrays of unknown size. The normal Win32 API tradition is followed again in this case. GetPath is normally called twice, the first call to get the number of points, and the second call to get the real data after allocating the two arrays. An application can also allocate two reasonable-size arrays on the stack, call GetPath with the two arrays, and deal with dynamic memory allocation only when it fails to avoid costly heap allocation. A rule of thumb is to allocate fixed-size data on the stack, which will cover 80% of the cases, and use heap allocation for the remaining 20%.

The class KPathData shown below demonstrates the first method. KPathData has two member variables to keep the two arrays returned by GetPath. The GetPathData method calls GetPath first to get the number of points. After allocating the necessary memory, it calls GetPath again to retrieve the true path data. The MarkPoints routine draws small marks to each point according to its type.

```
class KPathData
{
public:
    POINT * m_pPoint;
    BYTE  * m_pFlag;
    int    m_nCount;

    KPathData()
    {
        m_pPoint = NULL;
        m_pFlag  = NULL;
        m_nCount = 0;
    }

    ~KPathData()
    {
        if ( m_pPoint ) delete m_pPoint;
        if ( m_pFlag ) delete m_pFlag;
    }

    int GetPathData(HDC hDC)
    {
        if ( m_pPoint ) delete m_pPoint;
        if ( m_pFlag ) delete m_pFlag;

        m_nCount = ::GetPath(hDC, NULL, NULL, 0);

        if ( m_nCount>0 )
        {
            m_pPoint = new POINT[m_nCount];
            m_pFlag  = new BYTE[m_nCount];

            if ( m_pPoint!=NULL && m_pFlag!=NULL )
                m_nCount = ::GetPath(hDC, m_pPoint, m_pFlag,
                                     m_nCount);
        }
        else
            m_nCount = 0;
    }

    return m_nCount;
}
```

```
void MarkPoints(HDC hDC, bool bShowLine=true);
};
```

GetPath is a great way to learn how line and curve drawing functions are implemented by GDI because it lets you see the path data. For example, if you want to understand how AngleArc is converted to Bezier curves, try this:

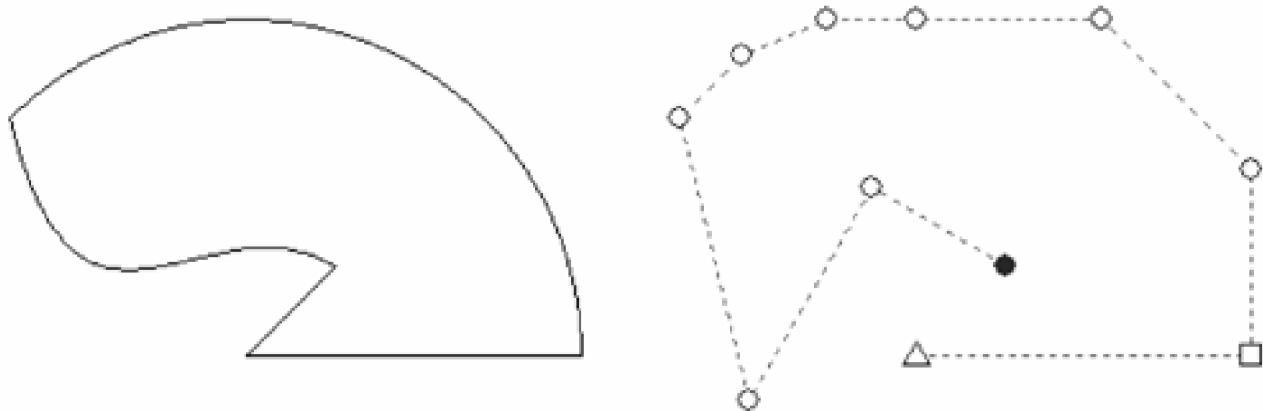
```
BeginPath(hDC);
MoveToEx(hDC, 0, 0, NULL);
AngleArc(hDC, 0, 0, 150, 0, 135);
POINT P[] = { -75, 20, -20, -75, 40, -40 };
PolyBezier(hDC, P, 3);
CloseFigure(hDC);
EndPath(hDC);

KPathData org;
org.GetPathData(hDC);
```

Between BeginPath and EndPath, there are four GDI calls. MoveToEx positions the current cursor position at the origin, AngleArc draws a 135-degree arc, PolyBezier continues with a Bezier curve, and then CloseFigure closes the figure. After calling Get PathData, you can either examine the path data in a debugger window, dump it to a text file, or mark the points and flags graphically on screen.

[Figure 8-15](#) shows two pictures. The one on the left shows the path as defined by the code shown above. It's drawn using PolyDraw on the data returned by GetPath. The one on the right shows the points and flags returned by GetPath. The figure starting points are marked with triangles, the line points are marked with rectangles, the Bezier curve points are marked with circles, and the figure closing points are painted using a solid fill marker.

Figure 8-15. Path and path data returned by GetPath.



The pictures reveal that AngleArc adds a line from the current pen position (0, 0) to the start point of the arc (150, 0); two Bezier curves are used to represent the 135-degree arcs. The first Bezier curve takes care of the first 90 degrees, the second handles the remaining 45 degrees. They also show that CloseFigure does not add a line segment to the path data; it just sets a flag in the last point. The Microsoft documentation never explicitly states where the PT_CLOSEFIGURE flag should be put. One book mistakenly puts it on the first off-line control point of a Bezier curve. But if you examine the data returned by GetPath, things become very clear. The PT_CLOSEFIGURE flag should be put on the last point of a figure.

The data returned by GetPath can be fed directly to PolyDraw, as we did here on the left-hand picture. This is very useful, considering that GDI does not give a user application a path handle. An application can keep the path data in its own storage and use it whenever needed. If you want to visualize the points instead of the curve (for editing purposes, for example) the point array returned can be passed to the Polyline call, as we did here in the right-hand picture.

Data can also be transformed using any transformation before being passed back to GDI again. Remember that GDI supports only affine transformation on NT-based systems. Non-NT based platforms do not support world transformation at all. Implementing transformations for line and curve drawing is not hard. For affine transformation that maps lines into lines, you only need to transform all control points, and then draw them using the same flags. For non-affine transformation that may map lines into curves, you may need to break down the lines and curves into smaller segments to get more accuracy.

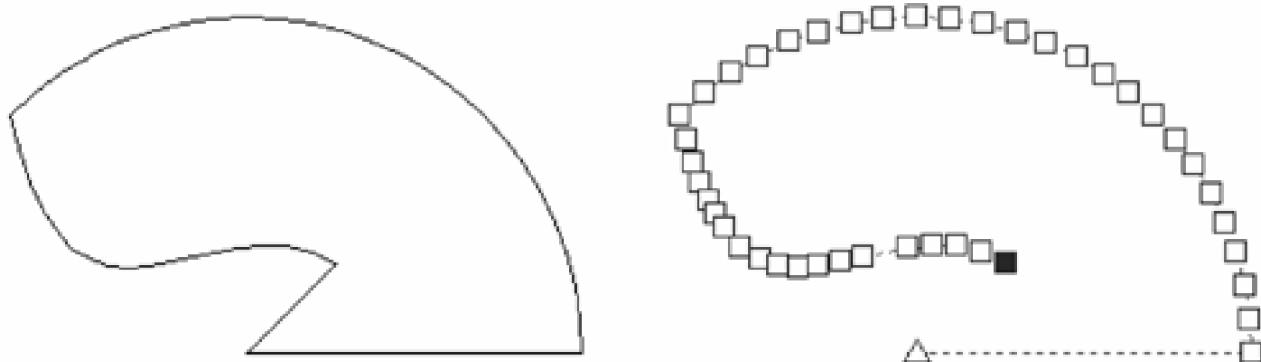
Transform Path Object

GDI defines two data transformations for a path object: curve-to-line conversion and wide-line expansion.

FlattenPath transforms any curve in a path object to a sequence of line segments good enough to approximate the curve in logical coordinate space. FlattenPath needs to transform only Bezier curves. The recursive algorithm shown in [Listing 8-3](#), which divides one curve into two curves until the error is insignificant, can be used. The name FlattenPath has the bad connotation that it generates gross approximations. But actually, the result generated by FlattenPath is the best possible in the integer current logical coordinate space. Precision is surely lost due to the integer-only restriction. But if you don't scale the result to a bigger size, the difference is not noticeable. If the result is intended for a higher-resolution device context, an application may increase the resolution of logical coordinate space to match it or implement its own floating-point version of FlattenPath.

FlattenPath is called after EndPath constructs a valid path. It modifies the current path object in a device context. The updated path object is still selected in the device context, ready to be used by other path functions like GetPath. Flattening a path may take considerable memory for a really complex path. [Figure 8-16](#) shows the result of FlattenPath. The left-hand picture is drawn using PolyDraw on the data after calling FlattenPath; the right-hand picture is drawn using Polyline with marks. You will not see a visible difference on the left side. But on the right side, sure enough, all the curve points are gone, replaced by many line points to make the path as smooth as before.

Figure 8-16. FlattenPath: approximating curves using lines.



Converting curves to lines has lots of useful applications; after all, lines are much easier to manipulate than curves. There is no simple formula to compute the length of a Bezier curve. A CAD/CAM application may let users design

patterns using Bezier curves, whose lengths are calculated when broken into lines. When closed figures are broken into lines, they are basically polygons. Algorithms are readily available to compute the areas covered by polygons, calculate bounding boxes, and check for intersections. A curve's length is also the guideline used in drawing styled lines. Styled-line drawing can be visualized as a process of repetitively taking a fixed-length segment from a curve, painting it as a solid line, skipping another fixed-length gap, and moving on. [Section 8.8](#) shows how flattened-path data can help in creating your own styled-lines.

The second path transformation provided by GDI is WidenPath. Microsoft documentation explains that WidenPath is a function that redefines the current path as the area that would be painted if the path were stroked using the pen currently selected in the device context. Note the key words here: WidenPath redefines a path as an area. But if a path is a path, how can it be an area? MSDN provides no explanation or illustration.

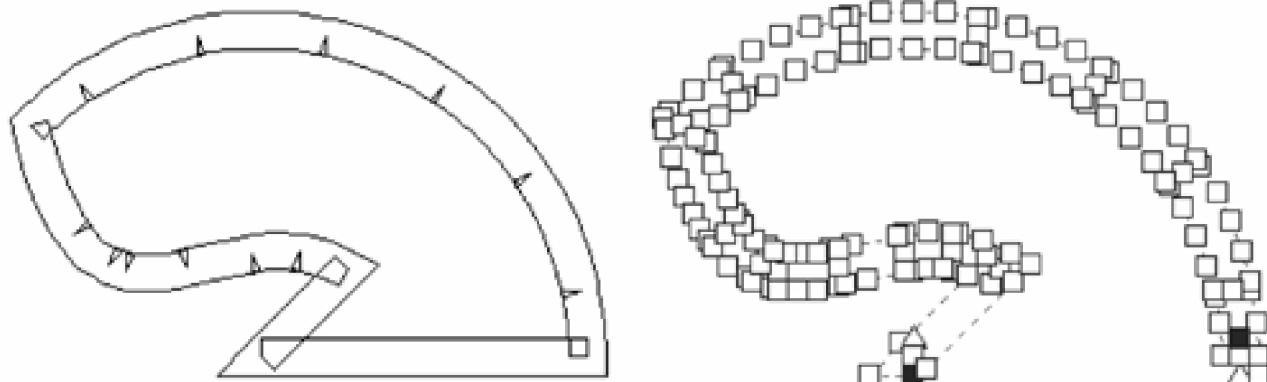
Actually, WidenPath converts the current path into the perimeters of the area that would be painted if the path were stroked using the current pen in the device context. WidenPath works only if the current pen is not a cosmetic pen created using Ext CreatePen. For pens created using CreatePen and geometric pens created using Ext Create Pen, WidenPath uses the pen width, pen style—including the end cap and join attributes—and miter limit in the device context to calculate the perimeters of all areas that need to be covered.

WidenPath always generates closed figures. WidenPath also converts curves to lines, just like FlattenPath. This shows again that lines are easier to work with. There is no need to call FlattenPath after calling WidenPath. But if you call FlattenPath before calling WidenPath, the path generated contains more points (with shorter lines).

One thing to notice, at least on Windows NT/2000, is that to be effective the pen needs to be selected into the device context before the last drawing call. If the pen selection is placed after the last drawing call, but before the CloseFigure and EndPath calls, the previous pen is effective.

[Figure 8-17](#) shows the ugly duckling generated by WidenPath. The pictures are generated using a 17-unit-wide geometric pen with a flat end cap and miter join. Widen Path converts a closed figure into two closed figures, one half the pen width inside the path line, and another half the pen width outside the path line. Without a doubt, Widen Path should convert an open path into one closed figure surrounding the original path by half the pen width on each side. If a styled pen were used in place of a solid pen, there would be multiple closed figures for all the isolated dots and dashes.

Figure 8-17. WidenPath converts one closed figure into two closed figures.



The path generated by WidenPath is already broken into lines. But it seems that GDI is using a better algorithm than calling FlattenPath first and then widening the path. If an application does that, the path generated has more points and more spikes.

The ugly spikes on the left are generated when thick lines less than 180 degrees apart meet. They represent the overlapping areas when two lines are drawn separately. If the result of WidenPath is used only to implement wide geometric lines, these spikes will be completely hidden when the closed figures are painted using a brush. But if an application wants to use the result to draw two closed curves, one inside and one outside the original path, a serious cleanup of the path data is needed.

WidenPath, or its internal implementation, may be the secret function used by GDI to convert geometric lines into area fills. Closed figures can be converted to regions that can be filled with brushes. This also explains why geometric logical pens are defined using a LOGBRUSH structure, allowing the same patterns as brushes. To GDI, drawing with geometric pens is a complex form of area fill.

WidenPath gives an application a chance to have the path data GDI would normally use to draw a geometric line. But Microsoft never explains why an application would want such internal data. Remember GDI supports only affine transformations; non-affine transformations, such as 1-point, 2-point, or 3-point perspective transformations, are not directly supported. Geometric lines are unique in that they have noticeable width. But when GDI draws geometric lines, they are always drawn using the same width. An accurate 3D screen needs to simulate the effect of depth, which maps further objects into smaller objects, including geometric lines. WidenPath provides a nice separation of jobs between GDI and applications. An application can get path data after WidenPath is applied, transform that using a perspective transformation or other transformation that may affect line width, and pass back to fill the path with a brush. The end result is nonuniform-width lines.

The code below shows an abstract class for a two-dimensional transformation K2D Map and a derived class KBiLinearMap. The KBiLinearMap class maps a rectangular window into an arbitrary quadrilateral. The K2DMap::Map method can be used to map individual points in path data.

```
class K2DMap
{
public:
    virtual Map(long & px, long & py) = 0;
};

class KBiLinearMap : public K2DMap
{
    double x00, y00, x01, y01, x10, y10, x11, y11;
    double orgx, orgy, width, height;

public:
    void SetWindow(int x, int y, int w, int h)
    {
        orgx = x;
        orgy = y;
        width = w;
        height = h;
    }

    void SetDestination(POINT P[])
    {
        x00 = P[0].x; y00 = P[0].y;
```

```
x01 = P[1].x; y01 = P[1].y;  
x10 = P[2].x; y10 = P[2].y;  
x11 = P[3].x; y11 = P[3].y;  
}  
  
virtual Map(long & px, long & py)  
{  
    double x = (px - orgx ) / width;  
    double y = (py - orgy ) / height;  
  
    px = (long) ( (1-x) * ( x00 * (1-y) + x01 * y ) +  
                 x * ( x10 * (1-y) + x11 * y ) );  
  
    py = (long) ( (1-x) * ( y00 * (1-y) + y01 * y ) +  
                 x * ( y10 * (1-y) + y11 * y ) );  
}  
};
```

Drawing Using a Path

A path object can be directly drawn using several GDI functions: StrokePath, FillPath, and StrokeAndFillPath.

StrokePath draws lines and curves contained in the current path, using the current pen and miter setting in the device context. Conceptually, StrokePath is similar to calling GetPath to retrieve the path data and calling PolyDraw with it. They draw the same lines. The main difference is that GetPath and PolyDraw do not affect the path in the device context, while StrokePath frees it after drawing. So after calling StrokePath, a new path needs to be constructed if it's needed. A workaround is to call SaveDC before StrokePath and RestoreDC after it. This little pair of brackets saves the path object from being destroyed.

It's not a good idea to call StrokePath after calling WidenPath with the same wide geometric pen. If the same wide pen used to widen the path is still selected, the original path is drawn using a pen double the width, resulting in ugly gaps and possibly bigger spikes. If a thin pen is used instead, the ugly spikes generated by WidenPath become quite visible, as shown in the left-hand picture in [Figure 8-17](#).

Here is a small code fragment that uses WidenPath to widen the paths with very thick geometric pens, and then uses StrokePath with thin cosmetic pens to stroke the paths generated by WidenPath:

```
for (int i=0; i<<3; i++)  
{  
    const WideStyle[] = { PS_ENDCAP_SQUARE | PS_JOIN_MITER,  
                         PS_ENDCAP_ROUND | PS_JOIN_ROUND,  
                         PS_ENDCAP_FLAT | PS_JOIN_BEVEL  
    };  
    const ThinStyle[] = { PS_ALTERNATE, PS_DOT, PS_SOLID };  
    const Color [] = { RGB(0xFF,0,0), RGB(0,0,0xFF), RGB(0,0,0) };  
  
    KPen wide(PS_GEOMETRIC | PS_SOLID | WideStyle[i], 70,
```

```
    RGB(0, 0, 0xFF), 0, NULL);
wide.Select(hDC);

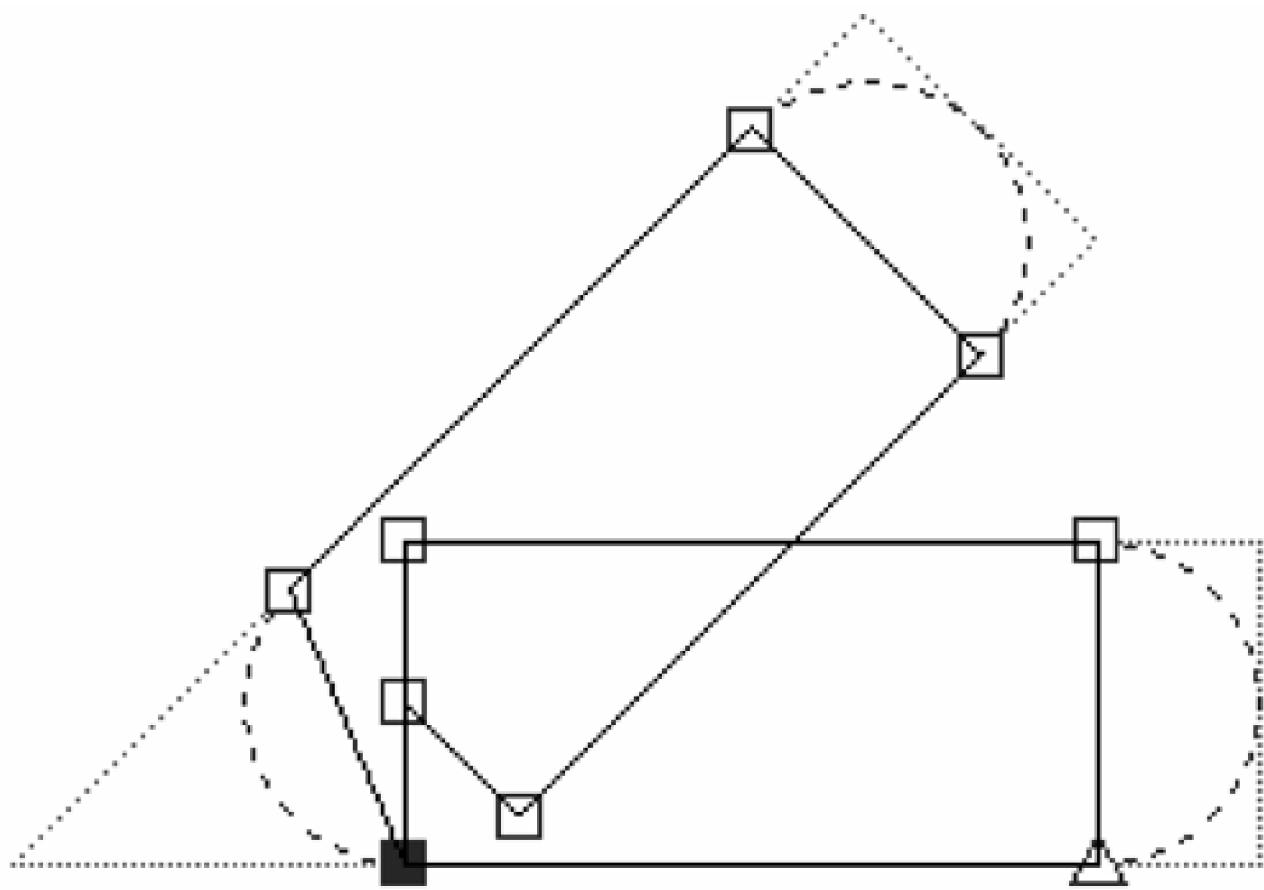
BeginPath(hDC);
MoveToEx(hDC, 150, 0, NULL);
LineTo(hDC, 0, 0);
LineTo(hDC, 100, -100);
EndPath(hDC);
WidenPath(hDC);
wide.UnSelect(hDC);

if (i==2 )
{
    KPathData pd; pd.GetPathData(hDC);
    pd.MarkPoints(hDC, false);
}

KPen thin(PS_COSMETIC | ThinStyle[i], 1, Color[i], 0, NULL);
thin.Select(hDC);
StrokePath(hDC);
thin.UnSelect(hDC);
}
```

The code loops three times, first for a pen with a square end cap and miter join, second for a pen with a round end cap and a round join, and finally for a pen with a flat end cap and a bevel join. It constructs a path each time with two lines forming a 45-degree angle. The paths are widened and stroked using different cosmetic lines. For the last pen, the control points from the widened path are marked according to their types. The code illustrates clearly how geometric pens with different end caps and join attributes are handled, as shown in [Figure 8-18](#).

Figure 8-18. Using WidenPath and StrokePath to show geometric pens.



The picture also shows how WidenPath generates spikes. For a flat end cap with a bevel join, as marked by a solid dark line in [Figure 8-18](#), WidenPath generates a closed figure, starting from the right bottom corner marked with a triangle. The path moves to the center of the other end of the line, where it meets the second line. It then moves to the second line, follows its perimeter all the way to the join, draws the join, and closes the figure. The spike, or the loop, is generated when two lines meet at an angle of less than 180 degrees. The GDI algorithm should have calculated the intersection of the perimeters of the two lines to remove the spikes instead of simply following the perimeters.

For non-NT-based platforms, StrokePath is especially important for geometric pens because their end cap and join attributes are implemented only when lines and curves are drawn within a path. That is to say, functions like LineTo, Arc, and BezierTo do ignore a geometric pen's end cap and join attributes and use the default values when used outside of a path.

FillPath fills the area covered by a path using a brush. StrokeAndFillPath fills the area like FillPath and draws the lines and curves like StrokePath. But a FillPath call followed by a StrokePath call can't replace a single StrokeAndFillPath call because all these three functions free the path object before they return. FillPath and StrokeAndFillPath will be examined in the next chapter.

Converting Path to Region

The two remaining GDI path functions convert a path object to a region object and use it for clipping. PathToRegion converts the current path in a device context to an independent region, which can be used for various purposes. SelectClipPath converts the current path in a device context to a region and uses it to update the application-defined clipping region in the device context.

We will revisit these two functions when we have a more in-depth discussion about regions and clipping.

8.8 SAMPLE: DRAWING YOUR OWN STYLED-LINES

Wide geometric styled-lines are not supported on non-NT-based platforms. Thus, graphics applications supporting both NT-based and non-NT-based platforms can't rely on GDI for wide geometric styled-line support. Even on NT-based platforms, GDI's support of styled-lines is not powerful enough. You can't repeat an arbitrary user-defined pattern along a path, and you can't define your own end cap and join types. When a path is broken into lines, it's easier to implement these algorithms. If you use the Bezier-curve-to-line conversion algorithm, though, working directly on Bezier curves could save memory and be more accurate.

[Listing 8-7](#) shows two classes defined to allow drawing many more styled-lines than are supported directly by GDI.

Listing 8-7 Classes for More Styled-Lines

```
class KDash
{
public:
    virtual double GetLength(int step)
    {
        return 10;
    }

    virtual BOOL DrawDash(double x1, double y1,
                          double x2, double y2, int step)
    {
        return FALSE;
    }
};

class KStyleCurve
{
    int      m_step;
    double   m_x1, m_y1;
    CDash   * m_pDash;

public:
    KStyleCurve(KDash * pDash)
    {
        m_x1 = 0; m_y1 = 0; m_step = 0;
        m_pDash = pDash;
    }

    BOOL MoveTo(double x, double y);
    BOOL LineTo(double x, double y);
    BOOL PolyDraw(const POINT *ppt, const BYTE *pbTypes,
                  int nCount);
};
```

```
BOOL KStyleCurve::LineTo(double x, double y)
{
    double x2 = x;
    double y2 = y;
    double curlen = sqrt((x2-m_x1)*(x2-m_x1)+(y2-m_y1)*(y2-m_y1));
    double length = m_pDash->GetLength(m_step);
    while ( curlen >= length )
    {
        double x1 = m_x1;
        double y1 = m_y1;
        m_x1 += (x2-m_x1) * length / curlen;
        m_y1 += (y2-m_y1) * length / curlen;

        if ( ! m_pDash->DrawDash(x1, y1, m_x1, m_y1, m_step) )
            return FALSE;

        curlen -= length;
        m_step++;
        length = m_pDash->GetLength(m_step);
    }
    return TRUE;
}
```

```
BOOL KStyleCurve::PolyDraw(const POINT *ppt,
                           const BYTE *pbTypes, int nCount)
{
    int lastmovex = 0;
    int lastmovey = 0;

    for (int i=0; i<nCount; i++)
    {
        switch ( pbTypes[i] & ~ PT_CLOSEFIGURE )
        {
            case PT_MOVETO:
                m_x1 = lastmovex = ppt[i].x;
                m_y1 = lastmovey = ppt[i].y;
                break;

            case PT_LINETO:
                if ( ! LineTo(ppt[i].x, ppt[i].y) )
                    return FALSE;
                break;
        }
        if ( pbTypes[i] & PT_CLOSEFIGURE )
            if ( ! LineTo(lastmovex, lastmovey) )
                return FALSE;
    }
    return TRUE;
}
```

}

The task of drawing a styled-line is broken into two subtasks, each managed by a class. The KDash class handles the styling cycle and drawing of dashes. This class is defined as an abstract class with two virtual methods to allow for drawing lines of different styles using different implementation classes. The GetLength method returns the length of a dash, given a step parameter. It has the same effect as the custom style bits array. For a styled-line with the same dash and gap lengths, GetLength could simply return a constant. If the style cycles are defined by a fixed array, GetLength could return $lpStyle[step \% dwStyleCount]$. The DrawDash method performs the actual drawing of dashes and gaps. It's given two points in floating-point numbers for precision and the current step count. The step count tells the routine whether the current segment is a dash or a gap. For example, a dashed line could draw every other segment.

The KStyleCurve class controls dividing lines and curves into segments of the required length. Currently, it handles MoveTo for defining the starting position, LineTo for drawing lines, and PolyDraw for drawing the data returned by GetPath on a flattened path. It can be easily extended to support Bezier curves and elliptic curves without using paths. The constructor of the KStyleCurve class has a pointer to a KDash class object, which can be used to query an instance of the KDash class. The main routine in the KStyleCurve class is KStyleCurve::LineTo. It queries KDash class for the current segment length and tries to cut a segment from the current line of the same length. This process is repeated until the current line is too short, in which case the processing is deferred until the next line. The implementation always returns a segment of the required size; any shorter remainder is merged with the next line. This design makes sure that segments are of the required length, but it may cut corners occasionally. A more sophisticated design would allow for the bending of dashes, even distribution of dashes, or clipping of dashes.

The relationship between the KStyleCurve class and the KDash class is very similar to the relationship between LineDDA and its callback routines. The main differences are that KStyleCurve allows curve handling through a path, and the KDash class uses the C++ virtual functions to handle a callback.

Here is a sample derived class: the KDiamond class of the KDash class. The class draws styled-lines with diamond shapes of different sizes and colors. GetLength returns 8 or 16 according to whether the current segment is odd or even. DrawDash uses (x_1, y_1) and (x_2, y_2) as two corners of a diamond shape and calculates the other two. It paints the diamond shape as a polygon of different colors. Polygon drawing will be discussed in the next chapter.

```
class KDiamond : public KDash
{
    HDC m_hDC;

    virtual double GetLength(int step)
    {
        return 8 + ( step & 1 ) * 8;
    }

    virtual BOOL DrawDash(double x1, double y1, double x2,
                         double y2, int step);

public:
    KDiamond(HDC hDC)
    {
        m_hDC = hDC;
    }
};

BOOL KDiamond::DrawDash(double x1, double y1,
```

```
    double x2, double y2, int step)
{
    HBRUSH hBrush = CreateSolidBrush(PALETTEINDEX(step % 20));
    HGDIOBJ hOld = SelectObject(m_hDC, hBrush);
    SelectObject(m_hDC, GetStockObject(NULL_PEN));
    double dy = (x2-x1)/2;
    double dx = (y2-y1)/2;

    POINT P[5] = { (int)x1, (int)y1,
        (int)((x1+x2)/2-dx), (int)((y1+y2)/2+dy),
        (int)x2, (int)y2,
        (int)((x1+x2)/2+dx), (int)((y1+y2)/2-dy),
        (int)x1, (int)y1 };

    Polygon(m_hDC, P, 5);
    SelectObject(m_hDC, hOld);
    DeleteObject(hBrush);
    return TRUE;
}
```

[Figure 8-19](#) shows the result of a more versatile class, which draws circles, squares, diamonds, and triangles as dashes.

Figure 8-19. Drawing your own styled-lines.



[< BACK](#) [NEXT >](#)

8.9 SUMMARY

Based on previous chapters, this chapter discusses the full details of line and curve drawing in Windows GDI. It covers binary raster operations, background modes, logical pens, lines, Bezier curves, arcs, and finally paths that combine lines and curves together.

Binary raster operations determine how pen pixel colors are combined with destination pixels to form the new destination pixels. R2_COPYPEN, which overwrites the destination with the pen color, is the normal case. Reversible raster operations are useful for interactive computer graphics, like rubber banding, object dragging, and hairline cursor display. The sixteen raster operations are rather limited. Alpha blending, which is also a way of combining a painting pixel and a destination pixel, is not supported in line drawing API.

For lines and curves, background modes control whether background pixels should be drawn. It's applicable only to simple pens. Cosmetic and geometric pens draw foreground pixels only.

Logical pens defined in NT-based platforms are quite powerful. But non-NT-based Win32 platforms have limited pen support, making it harder to write applications for both platforms. On Windows 95/98 systems, geometric pens do not support styled-lines, wide lines are not centered properly, end cap and join are supported only in path drawing functions, and alternate and user style pens are not supported. [Section 8.8](#) shows a class for defining your own fancy styled-lines. It can also be used to make up for the nonexistence of geometric styled pens.

GDI provides quite good line, Bezier curve, and elliptic curve drawing support. However, AngleArc, ArcTo, and PolyDraw are implemented only on NT-based systems. This chapter offers enough description and sample code to allow applications to provide their own implementations. We have presented detailed descriptions of Bezier curve theory and construction and of the conversion of a full ellipse or an elliptic arc into Bezier curves. Some mathematics is used here to help us solve practical application problems.

Path is a very important, though underused graphics programming technique in GDI. This chapter illustrates what's in a path and how it's constructed, converted, and used. We find some practical usage for paths in implementing fancy styled-lines and applying nonaffine transformations to generate nonuniform lines. Paths are used extensively in the Windows NT/2000 graphics engine and the DDI interface through which the graphics engine interfaces with graphics device drivers. Refer to [Chapter 3](#) for details on the paths' internal representation.

Geometric lines, which can be quite wide, are undoubtedly implemented using area fills instead of simply laying pixels along a path. So, on several occasions in this chapter we have referred to the next chapter.

Now, finally, let's move on to a new dimension, the GDI area fills.

Further Reading

Bezier curves are an important topic in computer graphics. You can refer to any good computer graphics book for a more complete description of Bezier curves—for example, *Computer Graphics*, by Francis S. Hill, Jr.

GDI provides support for basic line drawing but no help in geometric manipulation. *Graphics Gems*, Volumes I–V, edited by Andrew S. Glassner, James Arvo, David Kirk, Paul S. Heckbert, and Alan W. Paeth, are great sources for graphics algorithms. For example, Volume I has an article on the properties of Bezier curves and an article on solving the nearest-point-on-a-curve problem. Volume II has an article on fast anti aliased circle generation.

Sample Program

There is only one sample program for this chapter, “LineCurve,” but it is big enough to cover all the topics in this chapter (see [Table 8-6](#)). In fact, all the figures in this chapter are screen captures from this program.

Table 8-6. Sample Program for Chapter 8

Directory	Description
Samples\Chapt_08\LineCurve	The Test menu contains dozens of options to demonstrate binary raster operations, DC pen, simple pen, extended pen, line, Bezier curve, arc, path, and user-defined style curve shown in Section 8.8 .

[**< BACK**](#) [**NEXT >**](#)

Chapter 9. Areas

Modern calculus has its origin in two old mathematical problems: find a line tangent to a given curve, and find the area enclosed by a given curve. The first problem was solved by differential calculus, the second by integral calculus. In the Windows graphics API, line and curve API handles layering pixels along lines and curves; area-filling API handles painting the areas enclosed by closed figures formed by lines and curves.

We discussed line and curve drawing in great depth in the last chapter. In this chapter, we're moving up a grade to explore a new dimension: area filling. More specifically, we're going to examine the basic tools in area fill: brushes; basic data structures: rectangles and regions; common shapes: rectangles, polygons, ellipses, chords, pies; and the latest development in area fill: gradient fills.

9.1 BRUSHES

There are lots of attributes involved in painting an area with pixels: geometric shape, interpretation rules for geometric shape, raster operation, background mode, color, and pattern. The Windows graphics API groups color and pattern used in area fills to form a brush object. Surprisingly, a brush Windows graphic is simpler than a pen, because pens draw lines and curves with different widths and styles that the brushes do not have.

A brush's color determines the required color for the foreground pixels in an area fill. A brush's pattern creates different repetitive filling effects.

Logical Brush Objects

GDI provides several functions to create brush objects, or to be more precise, logical brush objects. A logical brush is a description for the application's requirements for an area fill, mainly color and pattern. It tells a device driver how an area fill should be implemented, but device drivers would use a different data structure to represent their interpretation of the brush, which is normally called a physical brush.

The internal data structure of logical brushes is managed by GDI, together with other objects like device contexts, logical brushes, logical fonts, etc. Once a logical brush is created, its handle is returned to the application, which serves as its future reference when using it. The generic type for GDI object handles is HGDIOBJ; HBRUSH is reserved for logical brush handles.

Each device context has a logical brush object attribute, which can be accessed using GetCurrentObject, SelectObject, GetObject, or EnumObjects. These functions, which we have discussed with pen objects in [Chapter 8](#), handle several GDI objects the same way.

Like other GDI objects, a logical brush object consumes user process resources, kernel resources, and at least one entry in the GDI handle table. So once the logical brush is no longer needed, it should be deleted by calling DeleteObject.

Stock Brushes

GDI defines seven predefined brush objects, called stock brushes, which can be used easily by applications. To get a stock brush handle, call GetStockObject with BLACK_BRUSH, DKGRAY_BRUSH, GRAY_BRUSH, LTGRAY_BRUSH, WHITE_BRUSH, NULL_BRUSH (same as HOLLOW_BRUSH), or DC_BRUSH. Black, dark gray, gray, light gray, and white stock brushes are solid brushes with different grayscale levels. White brush is the default brush selected in a device context. When the null stock brush (identical to the hollow stock brush) is used, the interior of an area will not be painted, similar to a null pen that does not draw lines. Note that as GetSelectObject is a generic function, its result should normally be cast to HBRUSH for stock brush objects.

The DC stock brush, as returned by GetStockObject(DC_BRUSH), is a new feature of Windows 98/2000. Like the DC pen, the DC brush is a pseudo GDI object which can change color after it's attached to a device context. For the DC brush, once it's selected into a device context, its color can be accessed using:

```
COLORREF GetDCBrushColor(HDC hDC);
COLORREF SetDCBrushColor(HDC hDC, COLORREF crColor);
```

GetDCBrushColor returns the current DC brush color; SetDCBrushColor sets a new color and returns the old color. These functions can be used even when the DC brush is not selected into a device context, but they are not effective until the DC brush is selected.

Stock objects are precreated by the operating system and shared by all processes running on the system. There is no need to delete stock object handles after an application is done with them. But it's completely safe to call DeleteObject on a stock object handle. DeleteObject returns TRUE, without doing anything.

Custom Brushes

Being able to paint only five grayscale levels using the stock brushes is boring. To create or retrieve more colorful brushes with interesting patterns, use any one of the following functions:

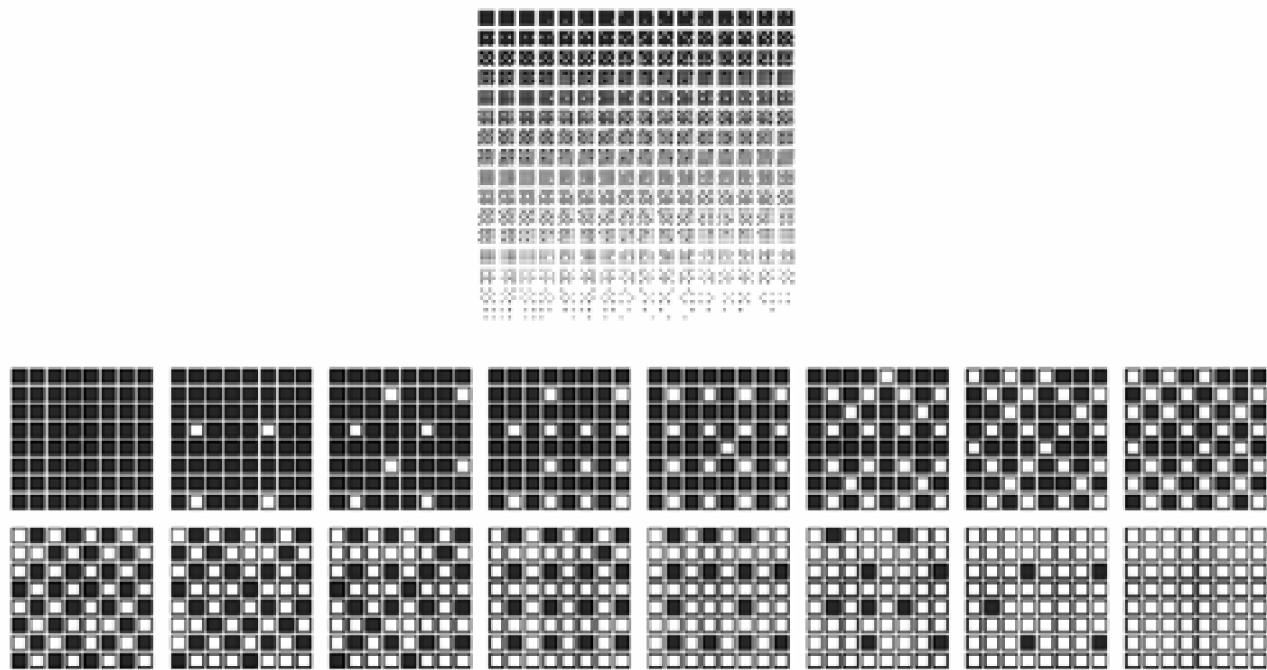
```
HBRUSH CreateSolidBrush(COLORREF crColor);
HBRUSH CreateHatchBrush(int fnStyle, COLORREF crRef);
HBRUSH CreatePatternBrush(HBITMAP hbmp);
HBRUSH CreateDIBPatternBrushPt(CONST VOID * lpPackedDIB,
    UINT iUsage);
HBRUSH CreateDIBPatternBrush(HGLOBAL hglbDIBPacked,
    UINT fuColorSpec);
HBRUSH GetSysColorBrush(int nIndex);
```

Solid Brushes

Solid color brushes are the easiest to create; only a color specifier is needed. Create SolidBrush creates only a logical brush. When its handle is selected into a device context, GDI and the device driver need to work out how the brush is going to be implemented. If the device context is non-palette-based, the color specifier is translated into RGB values without much trouble. On the other hand, for a palette-based device context, the color specifier needs to be translated to the palette index. If a match is found, the matching index is used in drawing; otherwise, the device simulates pixels of the required color using a combination of available colors, using the so-called dithering patterns. Dithering could produce more colors on 16-color or 256-color display cards, or even mimic grayscale on a black/white printer.

The code fragment shown below illustrates the usage of solid brush creation and selection. It paints an 8x8 rectangle for each of the 256 colors switching from blue to white, and displays zoomed views of 16 color rectangles on the diagonal line. If running on a 256-color display card, you will see dithering patterns as shown in [Figure 9-1](#).

Figure 9-1. Solid brush color dithering on palette-based devices.



```
// no outline for dither rectangle
SelectObject(hDC, GetStockObject(NULL_PEN));

for (int y=0; y<16; y++)
for (int x=0; x<16; x++)
{
    HBRUSH hBrush = CreateSolidBrush(RGB(y*16+x, y*16+x, 0xFF));
    HGDIOBJ hold = SelectObject(hDC, hBrush);

    Rectangle(hDC, 235+x*10, y*10, 235+x*10+9, y*10+9);

    if ( x==y ) // zoom in color on diagonal line
        ZoomRect(hDC, 235+x*10, y*10, 235+x*10+8, y*10+8,
                  80*(x%8), 180+80*(x/8), 6);

    SelectObject(hDC, hold);
    DeleteObject(hBrush);
}

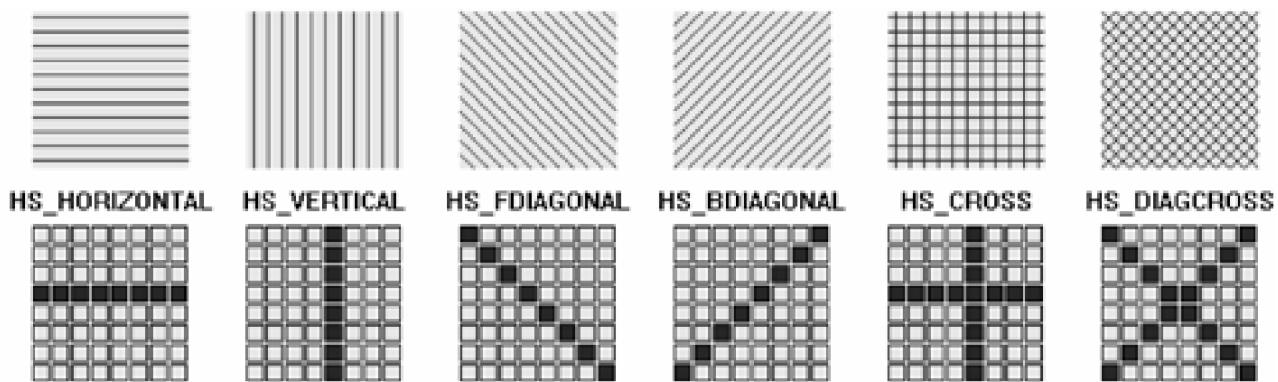
SelectObject(hDC, GetStockObject(BLACK_PEN));
```

For colors that are not in the current hardware palette, dithering creates patterns whose color averages are close to the original colors. For 8-by-8 dithering patterns, if two pure colors are used in them, there are at most 64 different levels of color. On low-resolution devices, dithering forms quite noticeable unpleasant patterns. But on a high-resolution device like a high-end printer, a device driver normally uses sophisticated halftoning algorithms to simulate continuous color tones. Combined with its tiny dots, a high-end printer can match the print quality of a finished photo.

Hatch Brushes

The CreateHatchBrush function creates logical brushes with one of the six predefined patterns and a color, which can be used to fill areas with uniform line patterns. The fnStyle parameter is the identifier for the hatch pattern, as illustrated in [Figure 9-2](#).

Figure 9-2. Hatch brush styles.



The upper part of the picture shows the result of using hatch brushes to fill rectangular areas. You can find out that small patterns are applied repetitively. The lower part of the figure zooms in to reveal these small patterns. Normally, device drivers use 8-by-8-pixel bitmaps to implement hatch brushes, but the DDI interface is designed to allow other implementation according to factors like resolution.

There are several things to notice when using hatch brushes: hatch pattern size, color, background mode, and hatch pattern alignment. Hatch brushes are normally implemented using 8-by-8 bitmaps in the device unit—that is, 8 pixels by 8 pixels. Unlike most other GDI features, the hatch brush pattern size is not defined in the logical unit. For example, for the screen display, hatch brushes are always seen to use 8-by-8 pixel patterns, independent of the logical-to-physical-coordinate transformation. It looks OK for the normal screen display, but it may look odd when the display zoom factor is really small, when a hatch pattern may look out of proportion. On the other hand, if an application uses hatch brushes to print to a high-resolution printer, area fills done with hatch brushes may not have a noticeable pattern at all. How large are 8 pixels on a paper printed by a 2400-DPI printer? They are 1/300 of an inch. To make hatch patterns have the same physical size as a 120-DPI screen display, a 2400-DPI printer needs to use 160-by-160-pixel hatch patterns. So if an application supports viewing documents in different zoom factors or printing, the predefined GDI hatch brushes should be avoided. Instead, alternatives that scale with device resolution and logical-to-device-coordinate mapping should be used.

Hatch brushes divide pixels into foreground pixels (shown in [Figure 9-2](#) in dark color), and background pixels (shown in light color). Foreground pixels are always painted, while background pixels are painted only when the background mode is OPAQUE. A device context's background mode attribute can be accessed using GetBkMode and SetBkMode. The second parameter crRef in CreateHatchBrush specifies the foreground color; the background color is controlled by the device context's background color attribute, accessible using GetBkColor and SetBkColor. For either foreground color or background color, GDI matches it to the closest pure color (color that is in the system palette), which will be used in drawing; no dithering is used for hatch brushes.

When multiple drawing objects use hatch brushes, or when scrolling is allowed, alignment of hatch patterns can become an issue. GDI aligns brushes per device context according to its brush origin attribute, which can be accessed using:

```
BOOL GetBrushOrgEx(HDC hDC, LPPOINT lppt);
```

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

Brush origin is a point (bx_0, by_0) in the device coordinate space that determines where a brush pattern's top leftmost pixel should be anchored; all other pixels follow accordingly. More specifically, point (x, y) in the device coordinate space should be painted with:

```
Pattern[(x-bx0) % pattern_width, (y-by0)% pattern_height]
```

The default brush origin is $(0, 0)$. To ensure brushes are properly aligned after transformation or mapping is changed, SetBrushOrgEx must be called. Here is an example to ensure brushes are aligned at $(0, 0)$ in logical coordinate space:

```
POINT P = { 0, 0 }; // origin  
LPtoDP(hDC, &P, 1); // map to device space  
SetBrushOrgEx(hDC, P.x, P.y, NULL);
```

Hatch brushes used to be very useful in business graphics like bar charts and pie charts to distinguish different data series. With better display cards having millions of colors and improved color printers, hatch brushes have become less useful, and even troublesome. If an application wants scalable patterns similar to GDI's hatch patterns, other drawing methods like lines or bitmaps could be used.

Bitmap Brushes

The six predefined hatch brushes are too restrictive, so GDI allows applications to create brushes based on application-supplied bitmaps. GDI supports two basic classes of bitmaps: device-dependent and device-independent. Both will be covered in detail in the next three chapters, so here we will just cover how to use bitmaps to create bitmap brushes and use them.

Given a device-dependent bitmap (DDB) handle, CreatePatternBrush creates a bitmap brush that tiles the bitmap to cover an area when used in area filling. GDI makes a copy of the bitmap, so its handle is no longer used by the logical brush after its creation. CreateDIBPatternBrushPt creates a brush from a pointer to a packed device-independent bitmap (DIB); CreateDIBPatternBrush creates a brush from a global memory handle to a DIB. In Win32 programming, a global memory handle (HGLOBAL) for a resource is actually a pointer in the 32-bit linear address space. But a global memory handle as returned by GlobalAlloc is different from the corresponding pointer that can be retrieved using GlobalLock. The distinction between memory handle and memory pointer is really inherited from the 16-bit programming paradigm.

The code below shows how to use these three functions to create bitmap brushes. First, we have to find a place to find bitmaps without creating our own resource file. If you have ever played Windows card games like Solitaire, you may remember those colorful poker cards. They are actually stored in a DLL, CARDS.DLL, for other applications to share. The code here uses LoadLibrary to load the DLL; it uses LoadBitmap to create a DDB handle, and FindResource and LoadResource to create a global memory handle.

```
HINSTANCE hCards = LoadLibrary("cards.dll");  
  
for (int i=0; i<3; i++)
```

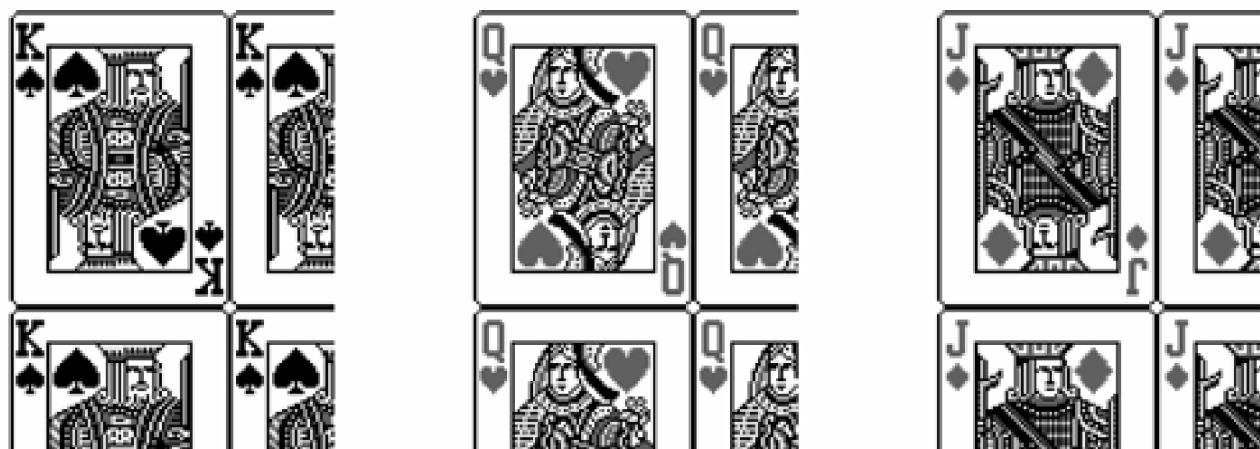
```
{  
HBRUSH hBrush;  
int width, height;  
  
switch ( i )  
{  
case 0:  
{  
HBITMAP hBitmap = LoadBitmap(hCards,  
MAKEINTRESOURCE(52));  
BITMAP bmp;  
GetObject(hBitmap, sizeof(bmp), & bmp);  
width = bmp.bmWidth; height = bmp.bmHeight;  
hBrush = CreatePatternBrush(hBitmap);  
DeleteObject(hBitmap);  
}  
break;  
  
case 1:  
{  
HRSRC hResource = FindResource(hCards,  
MAKEINTRESOURCE(52-14), RT_BITMAP);  
HGLOBAL hGlobal = LoadResource(hCards, hResource);  
  
hBrush = CreateDIBPatternBrushPt(  
LockResource(hGlobal), DIB_RGB_COLORS);  
width = ((BITMAPCOREHEADER *) hGlobal)->bcWidth;  
height = ((BITMAPCOREHEADER *) hGlobal)->bcHeight;  
}  
break;  
  
case 2:  
{  
HRSRC hResource = FindResource(hCards,  
MAKEINTRESOURCE(52-28), RT_BITMAP);  
HGLOBAL hGlobal = LoadResource(hCards, hResource);  
  
hBrush = CreateDIBPatternBrush(hGlobal, DIB_RGB_COLORS);  
width = ((BITMAPCOREHEADER *) hGlobal)->bcWidth;  
height = ((BITMAPCOREHEADER *) hGlobal)->bcHeight;  
}  
}  
  
HGDIOBJ hold = SelectObject(hDC, hBrush);  
  
POINT P = { i*150+20, 250 };  
LPtoDP(hDC, &P, 1);  
SetBrushOrgEx(hDC, P.x, P.y, NULL); // aligned with rectangle
```

```
Rectangle(hDC, i*150+20, 250,  
         i*150+20+width*3/2, 250+height*3/2);  
  
SelectObject(hDC, hOld);  
DeleteObject(hBrush);  
}
```

The code loops through three cases. It loads the King of Spades as a DDB and creates a DDB pattern brush. It then loads the Queen of Hearts and locks the resource to get a packed DIB pointer to create a DIB pattern brush. In the third iteration it loads the Jack of Diamonds and locks the resource to get a memory handle to create another DIB pattern brush. The brushes are used to fill three rectangle areas 1.5 times the dimension of the bitmap on each side to use the tiling of patterns. To make sure each bitmap is aligned with the top-left corner of the rectangle, LPtoDP is used to map the logical coordinate to the device coordinate, and SetBrushOrgEx is called to do the alignment.

[Figure 9-3](#) shows the result.

Figure 9-3. Bitmap pattern brushes and brush origin.



Note, on Windows 95/98, CARDS.DLL is a 16-bit DLL, which can't be loaded directly by Win32 applications.

There are a few things to remember about pattern brushes. First, at the GDI level, bitmap pattern brushes can't fully replace hatch brushes. For a bitmap pattern brush, every pixel is treated as a foreground pixel; there is no background pixel. Second, like hatch brushes, the bitmap pattern brush size is in device coordinate units, and it's always parallel to the axes of the device coordinate space. Patterns drawn using bitmap pattern brushes always have the same orientation and pixel size in the device coordinate space. To create patterns that scale with device resolution, an application has to use patterns of different resolutions. The third problem is the most critical: bitmap pattern brushes are restricted to 8 by 8 pixels on non-NT-based platforms. For example, on Windows 95/98, if a bitmap larger than that is used, only the top-left corner is used in drawing. Solutions to achieve the same effect with GDI bitmap functions are discussed in the next chapter.

One widely known trick with bitmap pattern brushes is to use them to draw vertical or horizontal true dotted lines. Recall that the so-called PS_DOT style line actually uses three pixels for a single dot; the true dotted-line style PS_ALTERNATE is supported only on NT-based platforms. Short of drawing pixel by pixel, the bitmap pattern brush offers a simple solution for vertical or horizontal lines. An application could create a pattern brush with a chessboard pattern and use it to draw single-pixel wide or high rectangles, or to draw an area that is clipped to be single-pixel wide or high.

The code below shows how to create a chessboard pattern brush and use it to draw a PS_ALTERNATE style-like rectangle perimeter. The pattern is created using CreateBitmap with an 8-by-8 black/white bitmap. A bitmap function PatBlt is used to draw the single-pixel line with the brush, which works in MM_TEXT mapping mode. For other mapping modes or advanced graphics mode, a clipping needs to be used to ensure only single-pixel lines are drawn.

The code also shows the second trick of the chessboard pattern brush: to draw semitransparent patterns. Assume in the current device context that black is the text color, white is the background color, pixel 0 is mapped to black (RGB(0, 0, 0)), and pixel 1 is mapped to white (RGB(255, 255, 255)). Raster operation R2_MASKPEN is used to "AND" brush color with the destination color. Black pixels in the brush are still black, but white pixels do not change destination. So only half the pixels get displayed, creating a semitransparent effect.

```
void Frame(HDC hDC, int x0, int y0, int x1, int y1)
{
    unsigned short ChessBoard[] = { 0xAA, 0x55, 0xAA, 0x55, 0xAA,
        0x55, 0xAA, 0x55 };

    HBITMAP hBitmap = CreateBitmap(8, 8, 1, 1, ChessBoard);
    HBRUSH hBrush = CreatePatternBrush(hBitmap);
    DeleteObject(hBitmap);

    HGDIOBJ hOld = SelectObject(hDC, hBrush);

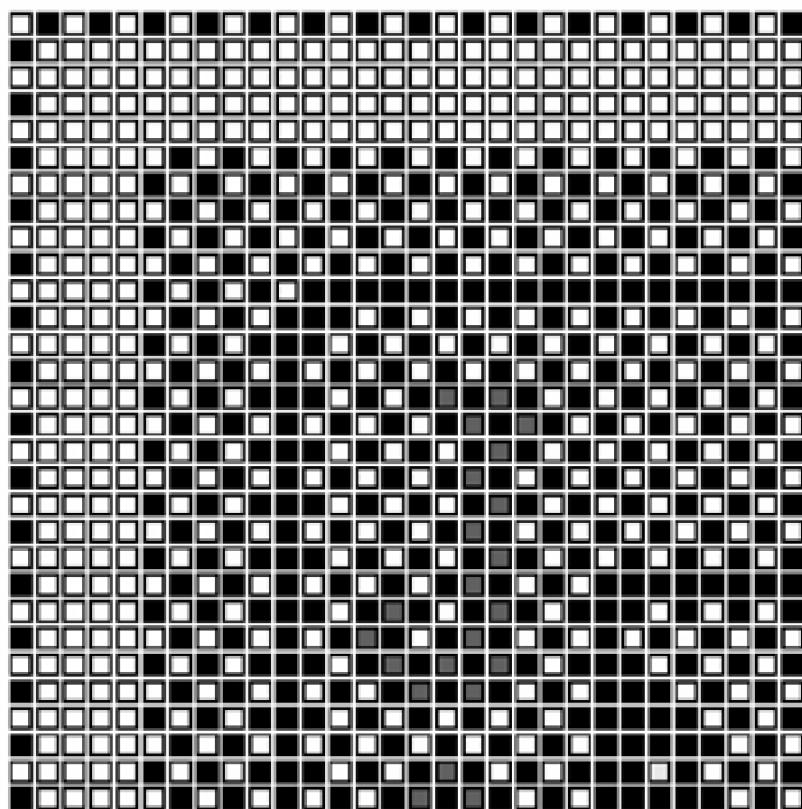
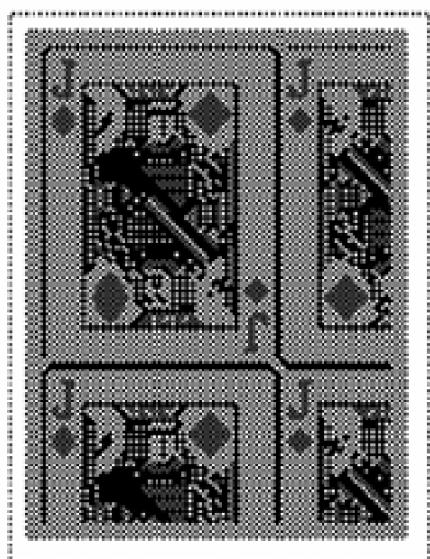
    // PS_ALTERNATE rectangle
    PatBlt(hDC, x0, y0, x1-x0, 1, PATCOPY);
    PatBlt(hDC, x0, y1, x1-x0, 1, PATCOPY);
    PatBlt(hDC, x0, y0, 1, y1-y0, PATCOPY);
    PatBlt(hDC, x1, y0, 1, y1-y0, PATCOPY);

    int old = SetROP2(hDC, R2_MASKPEN);
    Rectangle(hDC, x0+5, y0+5, x1-5, y1-5);
    SetROP2(hDC, old);

    SelectObject(hDC, hOld);
    DeleteObject(hBrush);
}
```

[Figure 9-4](#) demonstrates the effects of the chessboard pattern brush. Microsoft should have made it one of the hatch brushes.

Figure 9-4. Chessboard pattern brush usage: true dotted line and semitransparency.



System Color Brushes

The window management system uses dozens of colors to draw various parts of a window, like title bar, border, menu, scrollbar, buttons, etc. These colors are called system colors, configurable through the control panel display applet, or programmatically using the GetSysColor and SetSysColor API. For every system color, a stock brush is also created by the system. An application can retrieve the system color brushes using GetSysColorBrush function, given an index range from COLOR_SCROLLBAR to COLOR_GRADIENTINACTIVECAPTION.

System color brushes are useful to paint areas that need to have consistent colors as regions painted by the system. If a window handles customized painting of its nonclient area, system color brushes would be very handy. System color brushes are stock GDI objects, which do not need to be deleted after use. Calling DeleteObject on system color brushes will be ignored by the system.

System color brushes can be used as background brushes in registering window class. But it's all right to use a system color index in the format (HBRUSH) (COLOR_WINDOW+1). On certain systems, according to MSDN, GetSystemColorBrush may fail if the window management module USER32.DLL is loaded and unloaded multiple times. The reason is that whenever USER32.DLL is loaded, it builds a table of system color brushes; but when it unloads, it does not care to release these brushes. So if USER32.DLL is loaded and unloaded hundreds of times, it could overflow the GDI object table. This could only happen on a console application dynamically loading/unloading DLLs using USER32.DLL. For GUI-base application, USER32.DLL is always loaded, never unloaded and reloaded. For Windows NT/2000, system color brushes are created once and shared by all processes.

LOGBRUSH Structure

Let's briefly summarize what we've discussed so far. Brushes are used to paint the interior area of a region. GDI supports three types of brushes: solid brushes, hatch brushes, and pattern brushes. A solid brush is defined using a

color specifier, which could be dithered on a palette-based device. A hatch brush is defined using a color specifier and a hatch identifier. A hatch brush is special in that it divides pixels into foreground pixels and background pixels; the latter are drawn only in opaque background mode. The use of hatch brushes should be restricted to simple screen display, because their patterns normally do not scale with device resolution and display scale. A pattern brush is defined using either a device-dependent or a device-independent bitmap. The bitmap will be tiled in the area a pattern brush is used repetitively. GDI's implementation on Windows 95/98 limits the usable size of a pattern to be 8 by 8 pixels, which makes a nice feature much less practicable.

All three types of brushes can be specified using a LOGBRUSH structure, which can be passed to CreateBrushIndirect to create a logical brush. Here are the related definitions.

```
typedef struct tagLOGBRUSH {  
    UINT    lbStyle;  
    COLORREF lbColor;  
    LONG    lbHatch;  
};  
HBRUSH CreateBrushIndirect(CONST LOGBRUSH * lpb);
```

The tricky part of the LOGBRUSH structure is that, when specifying a BS_PATTERN style brush, lbHatch is the DDB handle; when specifying a BS_DIB PATTERN or BS_DIBPATTERNPT style brush, lbHatch is the DIB handle or pointer, and the LOWORD(lbColor) is either DIB_PAL_COLORS or DIB_RGB_COLORS.

The LOGBRUSH structure can be used in querying a GDI brush object using GetObject, as in:

```
LOGBRUSH logbrush;  
GetObject(hBrush, sizeof(LOGBRUSH), & logbrush);
```

For the LOGBRUSH structure returned by GetObject, do not expect to find a valid handle or pointer to a pattern brush's bitmap. When creating a pattern brush, GDI makes a copy of the bitmap in the internal GDI data structure, which will be hidden from the user application.

The LOGBRUSH structure is also used in creating extended pens. A geometric pen actually uses a brush to draw lines and curves, so dithering, hatches, and bitmaps can also be used in it.

A logical brush object is an object managed by GDI. On Windows NT/2000, a logical brush object has a user mode object to optimize frequent solid brush creation and destruction, and a kernel mode object that stores all the information about a logical brush. For example, the kernel mode object keeps foreground color, background color, more detailed style flags, bitmap, mask bitmap, etc. The mask bitmap is needed to implement a hatch brush, which may skip the background drawing. On the DDI level, the graphics engine allows a device driver to supply bitmaps for hatch brush implementation when a physical device is enabled, so as to produce more visible hatch patterns. A special entry point is defined to allow the device driver to realize a logical brush—that is, to create its own internal interpretation of a logical brush, which will be later used in the drawing related to the logical brush.

Except for pattern brushes, a logical brush consumes very little memory. A pattern brush uses an extra GDI handle for the bitmap, regardless of its being a DDB or DIB pattern brush, and space for a copy of the bitmap.

9.2 RECTANGLES

A rectangle is a basic geometric shape that the Windows API deals with. It's used in a defined window, in a client area, in defining various shapes with a rectangular bounding box, in formatting text, and even in clipping. Win32 defines the structure and API for handling a rectangle as a data structure, and painting rectangle areas in various ways.

Rectangle as a Data Structure

The Win32 API defines a rectangle using a RECT structure, on which a dozen or so operations are defined. Here are some related definitions.

```
typedef struct _RECT {
    LONG left;
    LONG top;
    LONG right;
    LONG bottom;
}
BOOL SetRect(LPRECT lprc, int xLeft, int yTop,
             int xRight, int yBottom);
BOOL SetRectEmpty(LPRECT lprc);
BOOL IsRectEmpty(CONST RECT * lprc);
BOOL EqualRect(CONST RECT *lprc1, CONST RECT *lprc2);
BOOL CopyRect(LPRECT lprcDst, CONST RECT * lprcSrc);
BOOL OffsetRect(LPRECT lprc, int dx, int dy);
BOOL PtInRect(CONST RECT * lprc, POINT pt);

BOOL InflateRect(CONST LPRECT lprc, int dx, int dy);
BOOL IntersectRect(CONST LPRECT lprcDst, CONST RECT * lprcSrc1,
                   CONST RECT * lprcSrc2);
BOOL SubtractRect(LPRECT lprcDst, CONST RECT * lprcSrc1,
                  CONST RECT * lprcSrc2);
BOOL UnionRect(LPRECT lprcDst, CONST RECT *lprcSrc1,
               CONST RECT * lprcSrc2);
```

A rectangle is defined by its minimum and maximum coordinates on both axes, which correspond to its top left and bottom right corners in the device coordinate space. When dealing with functions using the RECT structure, it's always assumed that left is no more than right and top is no more than bottom. The reason is that these functions are provided by the window manager to manage window and client area rectangles, which use screen device coordinates, not GDI drawing. An application must normalize the data before they are passed to those functions; otherwise, unexpected results may be generated. The RECT pointers passed are also assumed to be valid pointers. Their implementation does limited checking for bad pointers, apparently for performance reasons.

SetRect sets all of the four fields in a RECT structure with a new value, mainly for initializing a new rectangle.

SetRectEmpty sets all four fields to zero, so the area it covers is empty. IsRectEmpty checks if a rectangle is empty, which is defined as its width or height being zero or negative. EqualRect checks if two rectangles have exactly the

same four fields. CopyRect copies the source rectangle into a destination rectangle. OffSetRect translates a rectangle—that is, moving its left and right by *dx* and top and bottom by*dy*. PtInRect checks if a point is within a rectangle. The test is top left inclusive and bottom right exclusive. A point on its left or top side is considered in the rectangle, while a point on its right or bottom side is considered outside.

InflateRect expands a rectangle *dx* units horizontally and *dy* units vertically. If the values are negative, the rectangle is actually shrunk. IntersectRect finds the overlapping part of two rectangles, which is either empty or a rectangle. SubtractRect removes a rectangle from another rectangle. But we all know that generally subtracting a rectangle from a rectangle generates a nonrectangular area, which needs to be described using three rectangles. The Win32 API defines the result of SubtractRect to be the bounding rectangle of the result area. So if the overlapping area of rectangles A and B has the same width or height as rectangle A, the overlapping area is considered removed in the subtraction result A - B; otherwise, A stays the same. UnionRect returns the bounding rectangle of the area covered by either of the two rectangles.

All these functions on RECT have very simple implementation. If performance is critical, an application should consider implementing them using in-line functions, instead of the Win32 API calls. For example, calling SetRect with five parameters is at least six instructions per call, while using an in-line function may need only four instructions.

A RECT structure has the same layout as an array of two POINT structures. If function LPtoDP or DPtoLP needs to be called for coordinate transformation, a RECT structure can be cast as an array with two POINT structures. But, once a transformation or mapping is applied to a RECT structure, care should be taken to see if the rectangle is still normalized and parallel to the two axes.

Drawing Rectangles

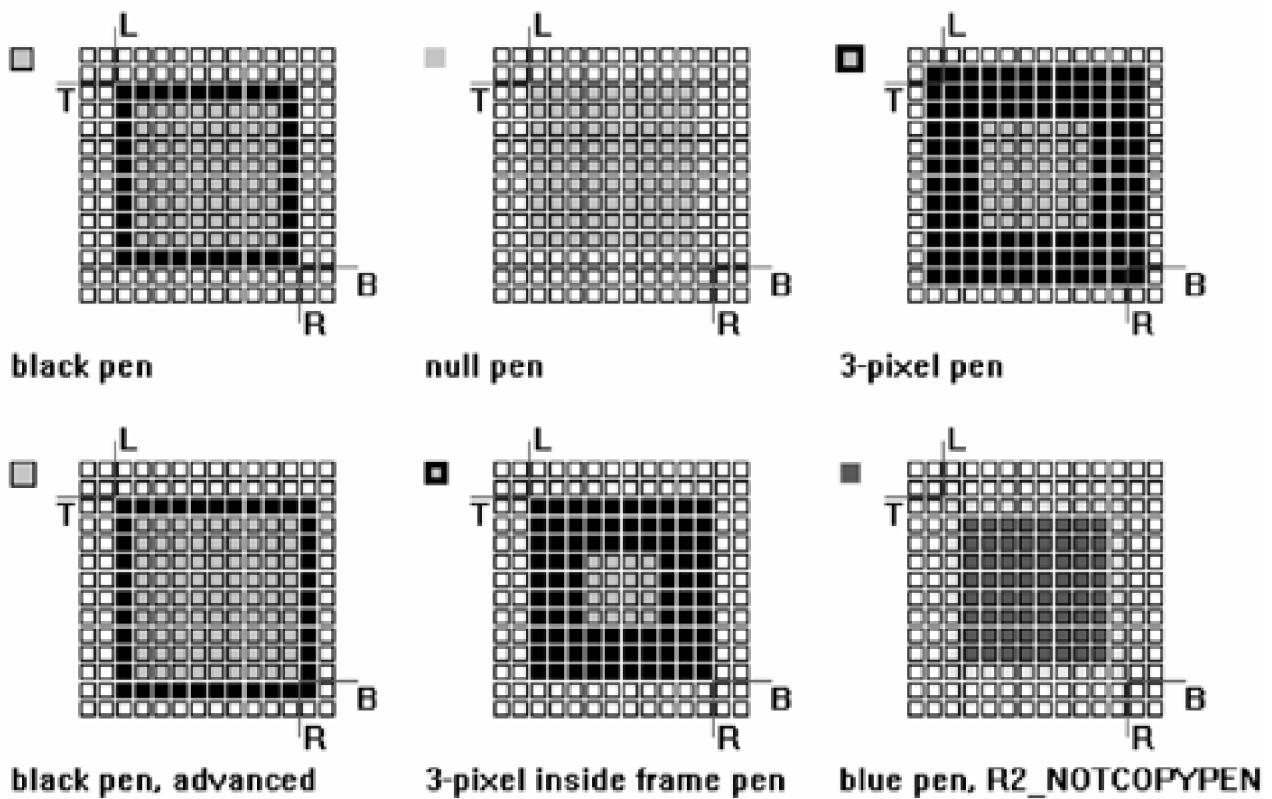
Win32 provides several functions to fill the interior of a rectangle using a brush, or stroke its perimeter using a pen, or do both:

```
BOOL Rectangle(HDC hDC, int nLeftRect, int nTopRect,  
    int nRightRect, int nBottomRect);  
int FillRect(HDC hDC, CONST RECT * lprc, HBRUSH hbr);  
int FrameRect(HDC hDC, CONST RECT * lprc, HBRUSH hbr);  
BOOL InvertRect(HDC hDC, CONST RECT * lprc);  
BOOL DrawFocusRect(HDC hDC, CONST RECT * lprc);
```

Rectangle

The Rectangle function draws a rectangle as defined by its four sides. Quite a few attributes in a device context affect the drawing of a rectangle. Because Rectangle is such a basic GDI function, we will explain it in more detail. [Figure 9-5](#) illustrates calling the Rectangle function with different device context settings.

[Figure 9-5. Different styles of rectangles.](#)



If the device context is in compatible graphics mode, the rectangle's bottommost edge and rightmost edge in the device coordinate space is not drawn, following the traditional inclusive, exclusive rule. Note that the rightmost edge may not correspond to nBottomRect; it's the maximum value of nTopRect and nBottomRect when mapped to the device coordinate space. But if the device context is in advanced graphics mode, all four sides of the rectangle are included in the drawing, as shown in the lower picture in the first column. This inconsistency may be necessary because it's hard to define right and bottom vs. left and top when an arbitrary affine transformation is allowed in the advanced graphics mode. The current pen object in the device context is used to stroke the perimeter of the rectangle. If the pen width is a single pixel in the device space, the pixels on the perimeter of the rectangle are painted. If the pen is n device pixels in width, one pixel is drawn on the centerline, $(n-1)/2$ outside the rectangle, and $(n-1)/2$ inside the rectangle. If an inside frame pen is used, one pixel is drawn on the centerline, and 1 pixels are drawn inside the rectangle. Another special case is the null pen, which does not draw the perimeter of a rectangle. When it's used, the rectangle is one pixel smaller in both height and width.

The current brush object in the device context is used to paint what's not painted by the pen, or in the case of the null pen, the whole rectangle one pixel smaller. The brush usage is also affected by background mode, background color, and brush origin.

The current raster operation in the device context influences both the perimeter and interior. For example, setting the raster operation to be R2_NOP makes the Rectangle function a no operation.

[FillRect](#)

The FillRect function fills a rectangle defined by a RECT structure with a brush. The rightmost and bottommost edges in the device coordinate space are always excluded, even in advanced graphics mode. Unlike calling Rectangle with a null pen that draws a smaller rectangle, the FillRect function paints the whole rectangle. It does not use the binary raster operation attribute in the device context.

The brush parameter to FillRect could also be a system color index, wrapped in the format (HBRUSH) (system color index + 1). The reason why FillRect is so different from Rectangle is that its implementation is based on GDI bitmap operations. FillRect calls an undocumented GDI function PolyPatBlt, which is based on PatBlt, to be described in the

next chapter.

FrameRect

The FrameRect function fills the perimeter of a rectangle with a brush, not a pen. The dimension of the actual drawing is the same as for FillRect. The width of the perimeter is 1 unit in logical coordinate space, whose actual pixel width is determined by world transformation and mapping.

Drawing the perimeter of a rectangle can generate interesting effects that could be difficult to do with a GDI pen. Using a brush allows dithered color to be used, without creating a geometric pen. A true dotted-line rectangle can be drawn using a pattern brush with the chessboard pattern. FrameRect is implemented using the same PolyPatBlt.

InvertRect

The InvertRect function inverts the color of each pixel in a rectangle, in the same way a pen in R2_NOT mode inverts pixels on a line. In a palette-based device, the palette indexes are inverted, and their colors will be determined by the color arrangement in the palette. In a non-palette-based device, black turns to white, white turns to black, and the RGB value of each pixel is inverted. Calling InvertRect twice with the same parameters restores the original contents of the device context.

InvertRect is implemented using PatBlt, so it follows the same inclusive-exclusive rule as FrameRect and FillRect.

DrawFocusRect

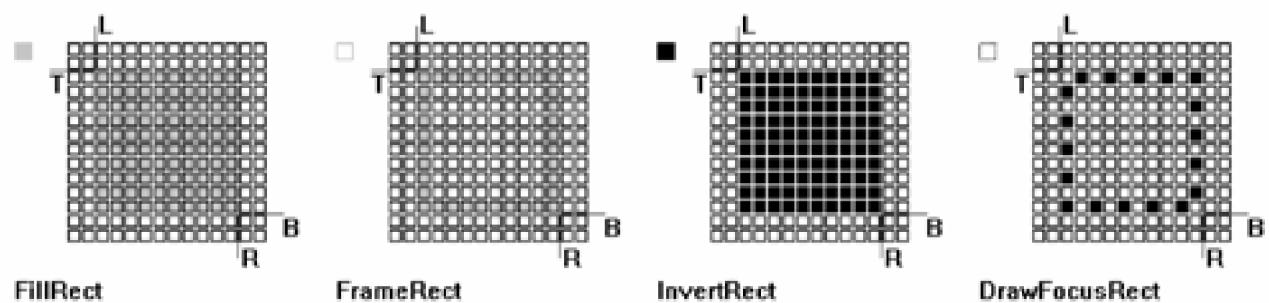
The DrawFocusRect function is similar to FrameRect. It draws the perimeter of a rectangle with a chessboard pattern brush using an exclusive-OR raster operation. Like InvertRect, calling DrawFocusRect twice restores the original contents of a device context.

DrawFocusRect is implemented using PolyPatBlt with a pattern brush and an exclusive-OR raster operation.

The name DrawFocusRect comes from its usage in the window management module. For example, a dialog box calls DrawFocusRect to draw a dotted rectangle on the button having the keyboard focus. When the keyboard focus goes to another button, DrawFocusRect is called a second time to remove the dotted rectangle, and is then called again to draw on the button having the focus. DrawFocusRect can also be used to draw a rubber rectangle. If DrawFocusRect is used for these user-interaction situations, care should be taken to maintain the correctness of the rectangle. The MSDN library overstates the problem by warning programmers that rectangles drawn using DrawFocusRect can't be scrolled. Actually, scrolling is not a problem, because the dirty region after the scrolling includes only the newly exposed areas, so calling DrawFocusRect in WM_PAINT message processing will not erase the rectangle. But if you use GetDC to retrieve a device context, which does not have the proper system region setup, erasing the rectangle before scrolling is an easier solution.

[Figure 9-6](#) illustrates FillRect, FrameRect, InvertRect, and DrawFocusRect. All these functions are not actually GDI functions, although they use GDI functions for the actual drawing. They are used and provided by the Window Manager (USER32.DLL).

Figure 9-6. Rectangle drawing functions used by the Window Manager.



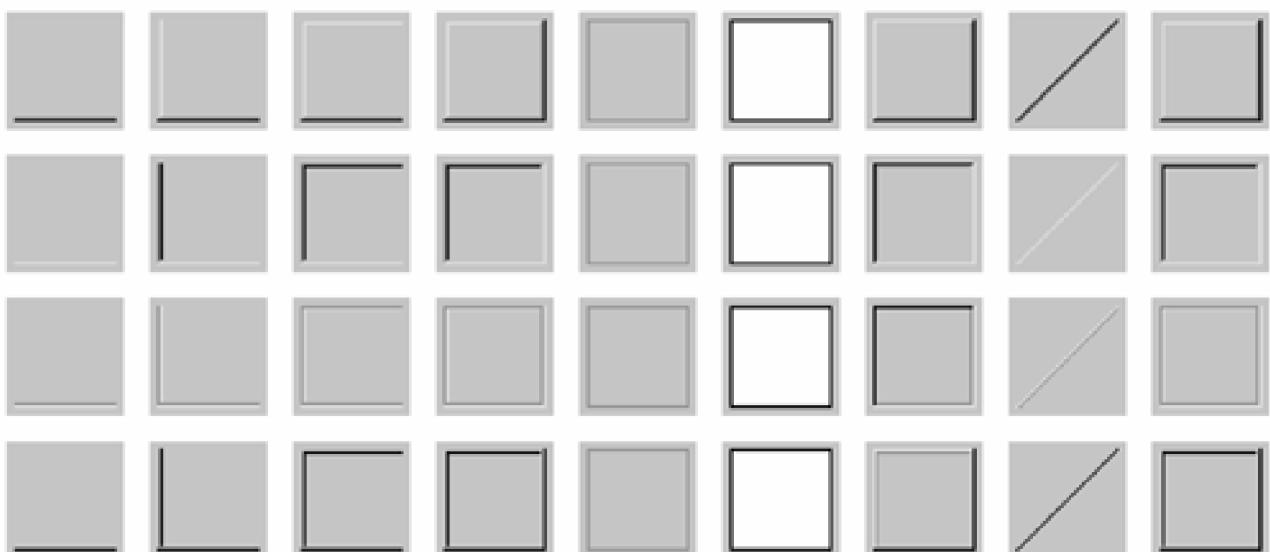
Drawing Edges and Controls

The Win32 API shares several functions the Window Manager uses to draw assorted edges and controls, which are closely related to rectangle drawing. They can be used to implement owner draw controls, take over non-client-area painting, or simulate window and control. Here are the prototypes for two major functions: DrawEdge and DrawFrameControl.

```
BOOL DrawEdge(HDC hDC, LPRECT lprc, UINT edge, UINT grFlags);  
BOOL DrawFrameControl(HDC hDC, LPRECT lprc, UINT uType,  
                      UINT uState);
```

Both functions take a device context, a RECT structure specifying an area to draw, and two flags. The available flags and their meaning are left to the MSDN library documentation. We will just show some sample usage here. The code below shows how to draw various edges, as illustrated in [Figure 9-7](#).

Figure 9-7. Using DrawEdge to draw various edges.

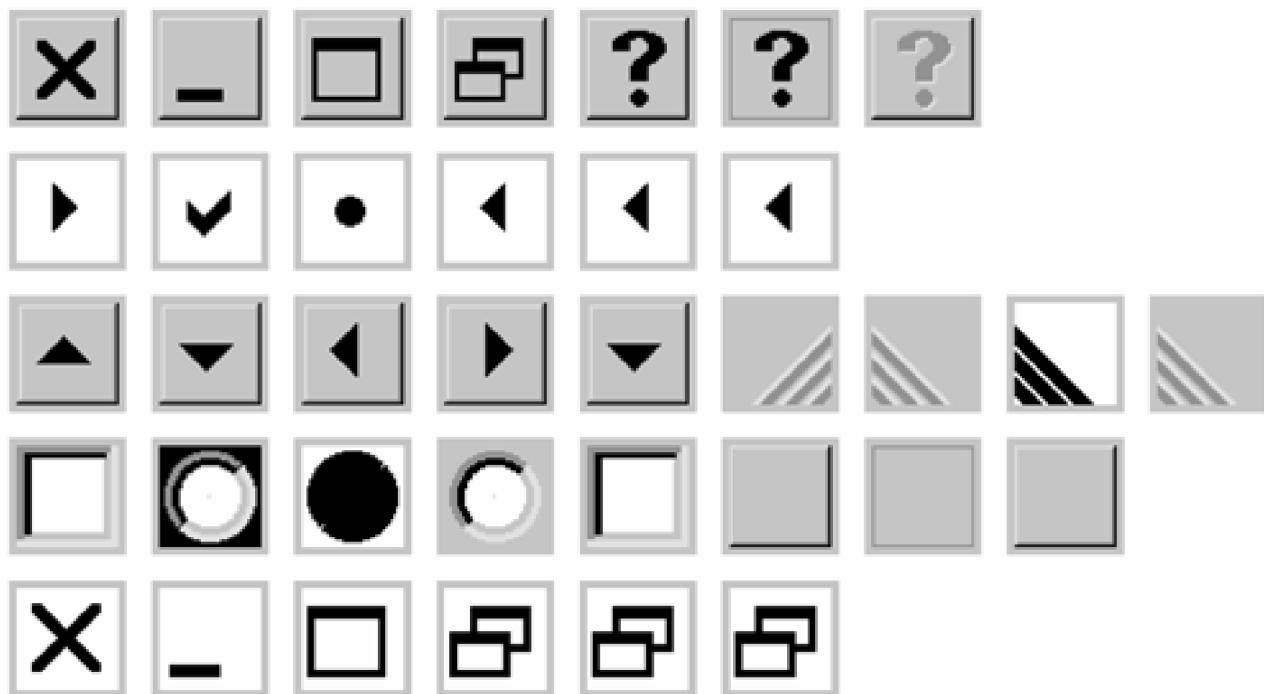


```
BF_MIDDLE | BF_RECT,
BF_MIDDLE | BF_RECT | BF_FLAT,
BF_MIDDLE | BF_RECT | BF_MONO,
BF_MIDDLE | BF_RECT | BF_SOFT,
BF_MIDDLE | BF_RECT | BF_DIAGONAL,
BF_MIDDLE | BF_RECT | BF_ADJUST );

for (int f=0; f<sizeof(Flag)/sizeof(Flag[0]); f++)
{
    RECT rect = { f*56+20, e*56 + 20, f*56+60, e*56+60 };
    InflateRect(&rect, 3, 3); // bigger background
    FillRect(hDC, & rect, GetSysColorBrush(COLOR_BTNFACE));
    InflateRect(&rect, -3, -3); // restore size
    DrawEdge(hDC, & rect, Edge[e], Flag[f]);
}
}
```

The DrawFrameControl function draws various controls commonly seen on title bars, menus, scrollbars, buttons, and popup menus. [Figure 9-8](#) shows a few examples. Refer to the MSDN library to complete the explanation.

[Figure 9-8. Using DrawFrameControl to draw various controls.](#)



[Figure 9-8](#) shows controls drawn into 40-by-40-pixel rectangles, much bigger than the size the system normally uses. The controls are displayed without jagged edges, which are normal if a small bitmap is scaled to draw the controls. You may have never guessed that Windows uses characters in the Marlett TrueType font to draw these crosses, arrows, question marks, etc., which makes them totally scalable.

9.3 ELLIPSES, CHORDS, PIES, AND ROUNDED RECTANGLES

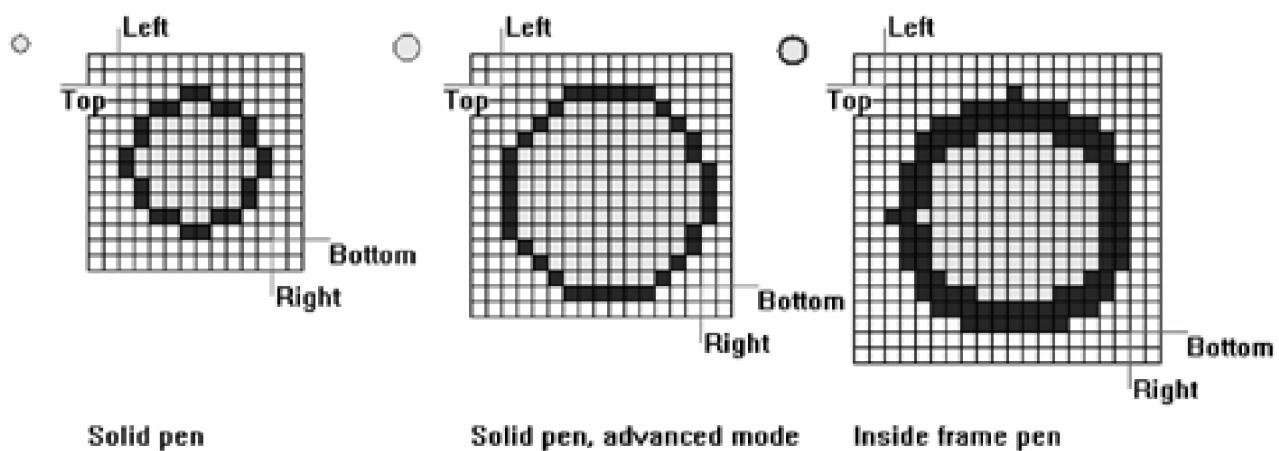
GDI provides several functions to draw a fully filled ellipse or part of it, and even a mixture between an ellipse and a rectangle, a rounded rectangle. Here are the related function prototypes.

```
BOOL Ellipse(HDC hDC, int nLeftRect, int nTopRect,  
             int nRightRect, int nBottomRect);  
BOOL Chord(HDC hDC, int nLeftRect, int nTopRect,  
            int nRightRect, int nBottomRect,  
            int nXRadial1, int nYRadial1,  
            int nXRadial2, int nYRadial2);  
BOOL Pie(HDC hDC, int nLeftRect, int nTopRect,  
         int nRightRect, int nBottomRect,  
         int nXRadial1, int nYRadial1,  
         int nXRadial2, int nYRadial2);  
BOOL RoundRect(HDC hDC, int nLeftRect, int nTopRect,  
               int nRightRect, int nBottomRect,  
               int nWidth, int nHeight);
```

These four functions have the same bounding box as the Rectangle function described above. Their drawing also has the same characteristics as drawing a rectangle. The current pen object in the device context is used to stroke the outline, while the interior is filled with the current brush. In compatible graphics mode, if the pen width in the device space is a single pixel, the rightmost and bottommost edges in the device space are not drawn on, following the inclusive-exclusive rule. But in advanced graphics mode, all four edges are drawn on. Microsoft documentation mentions only the fact that rectangles are rendered differently in the two graphics modes. Inside frame pens draw lines within the bounding box.

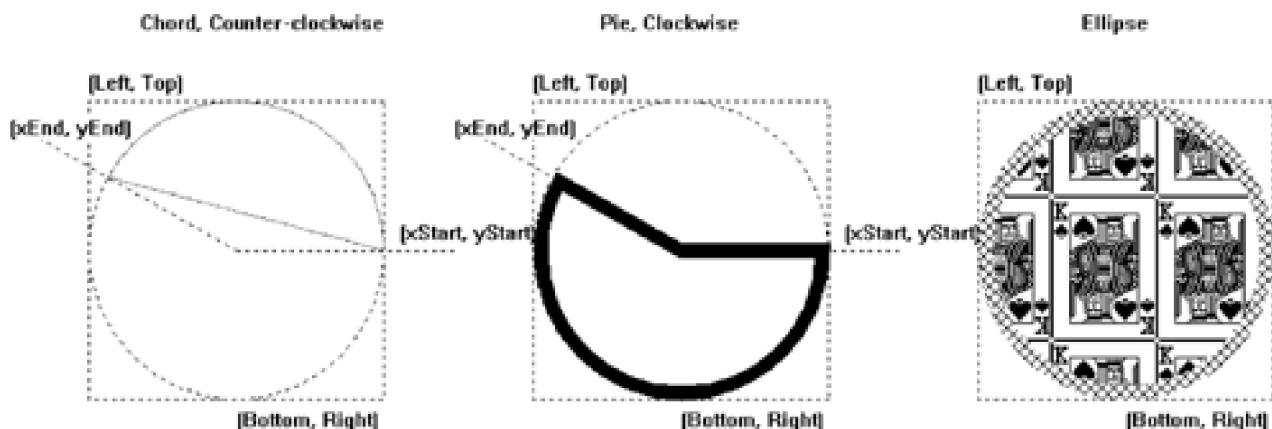
[Figure 9-9](#) looks at the pixel-level details of an ellipse drawing. The first ellipse is drawn using a single-pixel solid pen in compatible graphics mode; the bottom and right edges are not drawn on. The second ellipse is drawn in advanced graphics mode; the bottom and right edges are included. The third ellipse is drawn using a two-pixel-wide inside frame pen, which generates an ugly nonsymmetrical shape. The nonsymmetrical shape may be due to the Bezier curve approximation and drawing Bezier curves using lines. Although we mentioned that the approximation has a very tiny relative error, for a small figure as shown here, the single-pixel displacement does make a difference.

[Figure 9-9. Ellipse drawing details.](#)



[Figure 9-10](#) compares the Ellipse, Pie, and Chord functions. The Ellipse function draws a full ellipse perimeter with the current pen, and its interior with the current brush. A circle is a special case of an ellipse when its width and height are the same in physical units. The Pie function draws a pie-shaped wedge enclosed by a part of the ellipse perimeter and two radials. The Chord function draws a chord shape enclosed by a part of the ellipse perimeter and a secant line, which is the line linking two points on the perimeter. Both Pie and Chord use two points to specify their start angle and end angle, in the same way as the Arc function does. So the actual shape is also affected by the device context's arc direction parameter, if running in compatible graphics mode. In advanced graphics mode, arc direction is always counterclockwise in logical coordinate space.

[Figure 9-10. Ellipse, pie, and chord.](#)



The curves drawn by these functions are all closed curves. So if a geometric pen is used, its join attribute is applied at all intersections, while its end cap attribute will not be used. The perimeters they draw can also be stored in a path object. The Chord example here uses a cosmetic alternative pen. The Pie sample uses a wide geometric inside frame pen with a miter join. Note that the arc part is fully inside the bounding box, but the two radial lines have pixels evenly distributed on both sides. If an application uses the Pie function to draw a pie chart using different pen colors, the radial lines will show a different width. The Ellipse sample in [Figure 9-10](#) demonstrates the hatch pattern geometric pen and pattern brush.

Here is a generic routine for a simple pie chart drawing:

```
void DrawPieChart(HDC hDC, int x0, int y0, int x1, int y1,
    double data[], COLORREF color[], int count)
{
    double sum = 0;
```

```
double angle = 0;

for (int i=0; i<count; i++)
    sum += data[i];

for (i=0; i<count; i++)
{
    double a = data[i] * 2 * 3.14159265358 / sum;

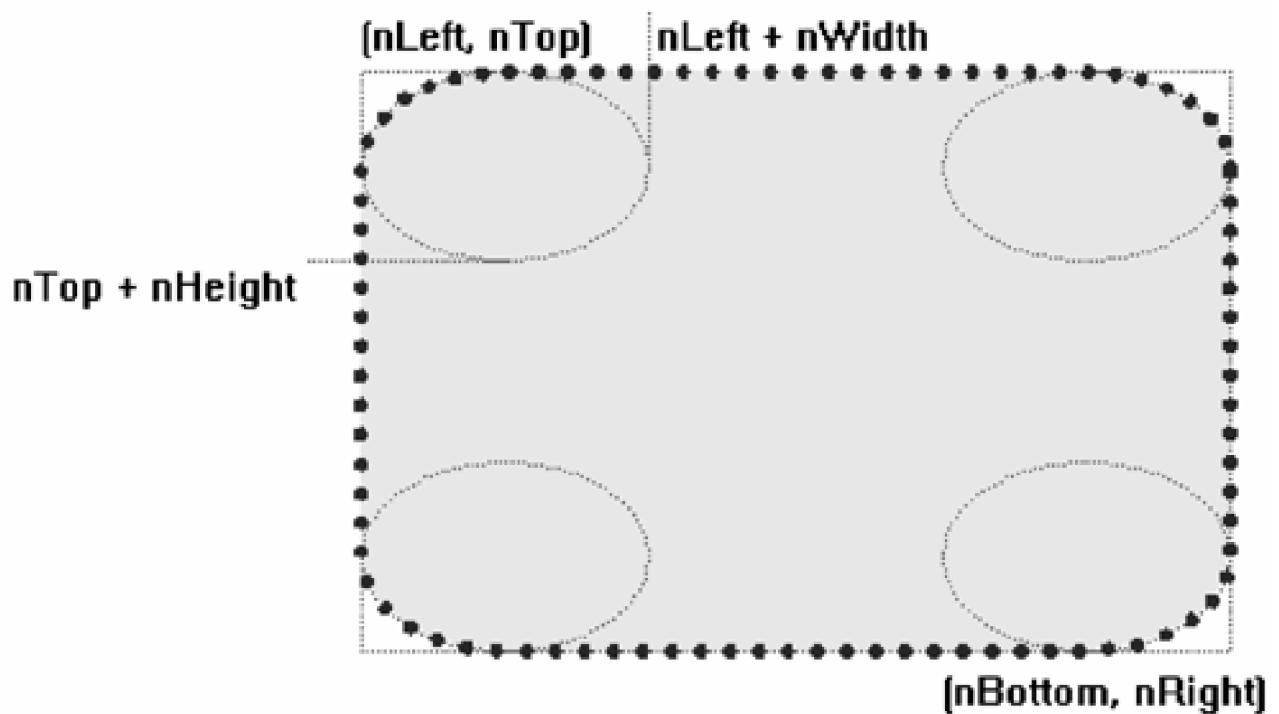
    HBRUSH hBrush = CreateSolidBrush(color[i]);
    HGDIOBJ hOld = SelectObject(hDC, hBrush);

    Pie(hDC, x0, y0, x1, y1,
        (int) ((x1-x0) * cos(angle)),
        - (int) ((y1-y0) * sin(angle)),
        (int) ((x1-x0) * cos(angle+a)),
        - (int) ((y1-y0) * sin(angle+a)));

    angle += a;
    SelectObject(hDC, hOld);
    DeleteObject(hBrush);
}
}
```

The RoundRect function can draw a rectangle, an ellipse, and anything between them, because it draws rectangles with rounded corners. It accepts the same bounding box parameters as the Rectangle and Ellipse functions. Its last two parameters, nWidth and nHeight, determine the dimension of the round corners. The round corners can be seen as four quadrants of an ellipse nWidth in width and nHeight in height. They are then linked using straight lines. If nWidth and nHeight are both zeros, RoundRect draws a rectangle. If nWidth equals the width of the bounding box, and nHeight is the same as its height, RoundRect draws an ellipse. In case an application passes a negative number to nWidth or nHeight, GDI uses their absolute values. The dimension of the bounding box also limits the size of the corners. [Figure 9-11](#) illustrates the Round Rect function.

[Figure 9-11. RoundRect: from rectangle to ellipse.](#)



[< BACK](#) [NEXT >](#)

9.4 POLYGONS

A rectangle is a special case of a polygon, which is defined as a closed shape having two or more vertices connected by straight lines. So a polygon can be a triangle, a parallelogram, a rectangle, a square, an octagon, and so on.

A polygon is defined by an array of vertices, or in the GDI API an array of POINT structures each representing the coordinate of a vertex. GDI supports drawing a single polygon or multiple polygons in a single call. Multiple polygons are stored in two arrays, an array of vertex count for each polygon and an array of all the vertices. Here are the GDI functions for polygon drawing.

```
int GetPolyFillMode(HDC hDC);
int SetPolyFillMode(HDC hDC, int iPolyFillMode);
BOOL Polygon(HDC hDC, CONST POINT * lpPoints, int nCount);
BOOL PolyPolygon(HDC hDC, CONST POINT * lpPoints,
    CONST INT * lpPolyCounts, int nCount);
```

Drawing a polygon using the `Polygon` function is similar to drawing a polyline using the `Polyline` function. The `lpPoints` parameter points to an array of `POINT` structures where the vertex coordinates are stored. The `nCount` parameter is the number of vertices in the array; at least two vertices are required. The `Polygon` function automatically closes the figure to ensure a closed figure and applies the `join` attribute to every vertex if a geometric pen is used. The polygon outline is stroked with the current pen, and its interior is painted using the current brush.

By the same token, drawing a multiple polygon using the `PolyPolygon` function is similar to drawing a multiple polyline using the `PolyPolyline` function. The `nCount` parameter is the number of polygons. The `lpPolyCounts` parameter points to an array of vertex count for each polygon, which should be no less than two. The `lpPoints` parameter points to an array of all the vertices put together consecutively. Each polygon is automatically closed by the `PolyPolygon` function. The combined outline of the polygons is stroked with the current pen, and their interiors are painted using the current brush.

Unlike functions with a rectangular bounding box, for example `Rectangle`, `Ellipse`, and `Arc`, which are rightmost, bottommost edge exclusive in compatible graphics mode, `Polygon` and `PolyPolygon` draw to all their vertices. They behave the same way in compatible and advanced graphics modes. So using a rectangle's four corners to draw a polygon is slightly different from drawing a rectangle in compatible mode. This explains why the `Rectangle` draws differently in two graphics modes. An affine transformation could rotate or shear a rectangle to make it a nonrectangle or not parallel to the axes, so the GDI graphics engine can't draw a transformed rectangle using the original rectangle support.

Polygon Fill Mode

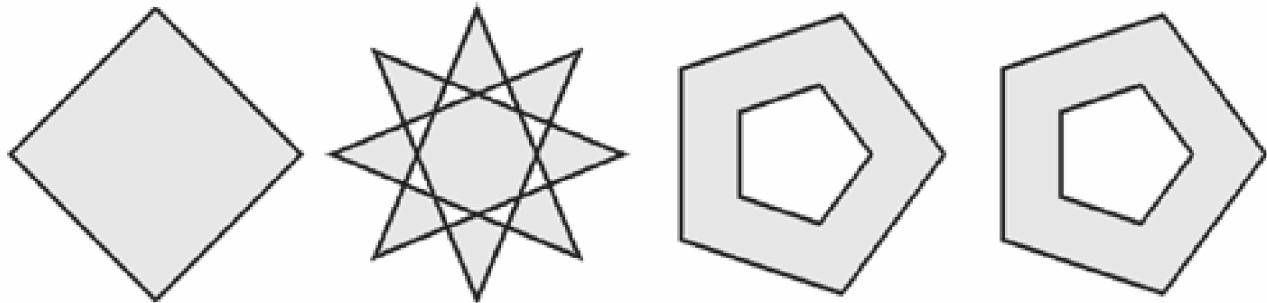
For a simple polygon—for example, a convex polygon—its interior is well defined. But a concave polygon could possibly overlap with itself, which makes the definition of its interior not so intuitive. Windows GDI allows two rules to define the interior of a polygon. In GDI's terminology, this is called polygon fill mode.

`Polygon` fill mode is an attribute of a device context object, which can be accessed using `GetPolyFillMode` and `SetPolyFillMode`. The two valid options are `ALTERNATE` and `WINDING`.

`ALTERNATE` polygon fill mode, which is the default setting for a device context, is very simple and intuitive. To test

whether a point is inside a polygon in ALTERNATE polygon fill mode, a ray is drawn from that point to infinity in any direction, and the number of intersections between the ray and the polygon outline is counted. If the number of intersections is odd, the point is inside; if even, the point is outside. [Figure 9-12](#) illustrates ALTERNATE polygon fill mode.

Figure 9-12. ALTERNATE polygon fill mode.



The first picture is a simple diamond shape, a convex polygon. Every scan line has two intersections with its perimeter; the points between them are the interior of the polygon. Linking the vertices of an octagon together to form a star shape, which overlaps with itself, generates the second polygon. Each scan line can have up to six intersections with the polygon, so anything between the second and third, fourth and fifth intersections is not considered interior to the polygon. The last two figures both contain two polygons; one bigger polygon contains a smaller polygon. The two cases are drawn in the same way in the ALTERNATE fill mode.

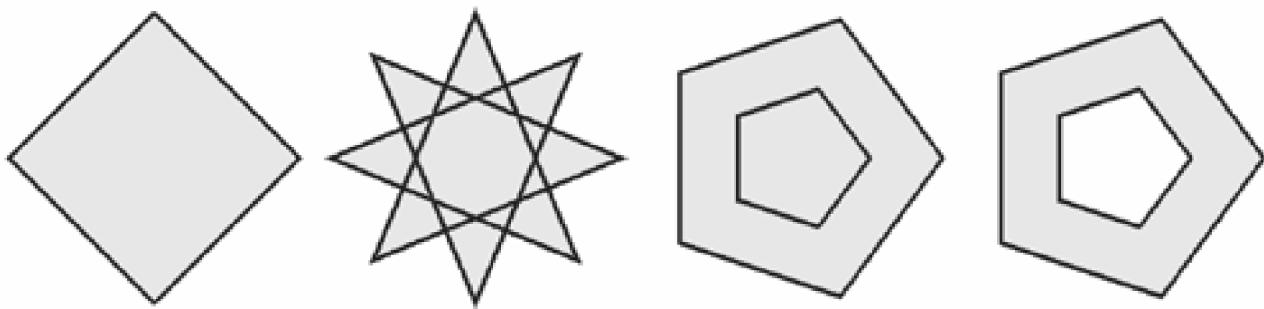
In real implementation, the bounding box for the polygon is calculated, a series of scan lines $y = y_{\min} + 0.5, y_{\min} + 1.5, \dots, y_{\max} - 0.5$ are tested for intersection with the outline. For each scan line, the number of intersections is always an even number. A point between the first and second, third and fourth, \dots , intersections is considered inside in the ALTERNATE fill mode.

The main problem with the ALTERNATE fill mode is in its handling of overlapping polygons. Part of the overlapping portion may be excluded from the painting, which is unfortunately quite common in computer graphics. In the last chapter, we saw that the paths generated by the WidenPath call contain spikes, which are actually overlapping areas. When multiple polygons meet together, overlapping is also very common. To solve this problem, GDI defines a more sophisticated WINDING polygon fill mode.

WINDING fill mode adds a sense of curve direction. To determine whether a point is within a polygon, the same ray is drawn from that point to infinity, and its intersections with the outline are examined. A count is kept for each ray, which starts from zero. Each intersection with a clockwise outline increments the count, whereas each intersection with a counterclockwise outline decrements the count. If the count is nonzero, the point is considered inside; otherwise, it's considered outside.

[Figure 9-13](#) shows the same polygons drawn in the WINDING polygon fill mode. All the polygons' curves here go clockwise, except the small inner polygon in the last picture, which goes counterclockwise. So the second and third pictures are now painted differently from ALTERNATE fill mode.

Figure 9-13. WINDING polygon fill mode.



Here is the code that generates both [Figures 9-12](#) and [9-13](#):

```
for (int t=0; t<2; t++)
{
    logbrush.lbColor = RGB(0, 0, 0xFF);
    KGDIObject pen (hDC, ExtCreatePen(PS_GEOMETRIC | PS_SOLID |
        PS_JOIN_MITER, 3, &logbrush, 0, NULL));

    if ( t==0 )
        SetPolyFillMode(hDC, ALTERNATE);
    else
        SetPolyFillMode(hDC, WINDING);

    for (int m=0; m<4; m++)
    {
        SetViewportOrgEx(hDC, 120+m*220, 120+t*220, NULL);

        const int s0[] = { 4,4, 100,0, 100,1, 100,2, 100,3 };
        const int s1[] = { 8,8, 100,0, 100,3, 100,6, 100,1,
            100,4, 100,7, 100,2, 100,5 };
        const int s2[] = { 10,5, 100,0,100,1,100,2,100,3,100,4,
            50,0, 50,1, 50,2, 50,3, 50,4 };
        const int s3[] = { 10,5, 100,0,100,1,100,2,100,3,100,4,
            50,4, 50,3, 50,2, 50,1, 50,0 };
        const int * spec[] = { s0, s1, s2, s3 };
        const double PI2 = 2 * 3.1415927;

        POINT P[10];
        int n = spec[m][0]; // number of points
        int d = spec[m][1]; // number of vertexes for each polygon
        const int * s = spec[m]+2; // radius, vertex index
        for (i=0; i<n; i++)
        {
            P[i].x = (int) ( s[i*2] * cos(s[i*2+1] * PI2 / d ) );
            P[i].y = (int) ( s[i*2] * sin(s[i*2+1] * PI2 / d ) );
        }

        if ( m<2 )
            Polygon(hDC, P, n);
    }
}
```

```
else
{
    int V[2] = { 5, 5 };
    PolyPolygon(hDC, P, V, 2);
}
}
```

[< BACK](#) [NEXT >](#)

9.5 CLOSED PATHS

A path is a GDI object that can hold multiple sequences of lines, arcs, and Bezier curves. Once a path is built by enclosing line and curve drawing calls between BeginPath and EndPath, it can be stroked using a pen, painted using a brush, or both stroked and painted. [Chapter 8](#) covers drawing the outline of a path using a pen; here we are revisiting paths for painting its interior using a brush. Here are the two related functions.

```
BOOL FillPath(HDC hDC);  
BOOL StrokeAndFillPath(HDC hDC);
```

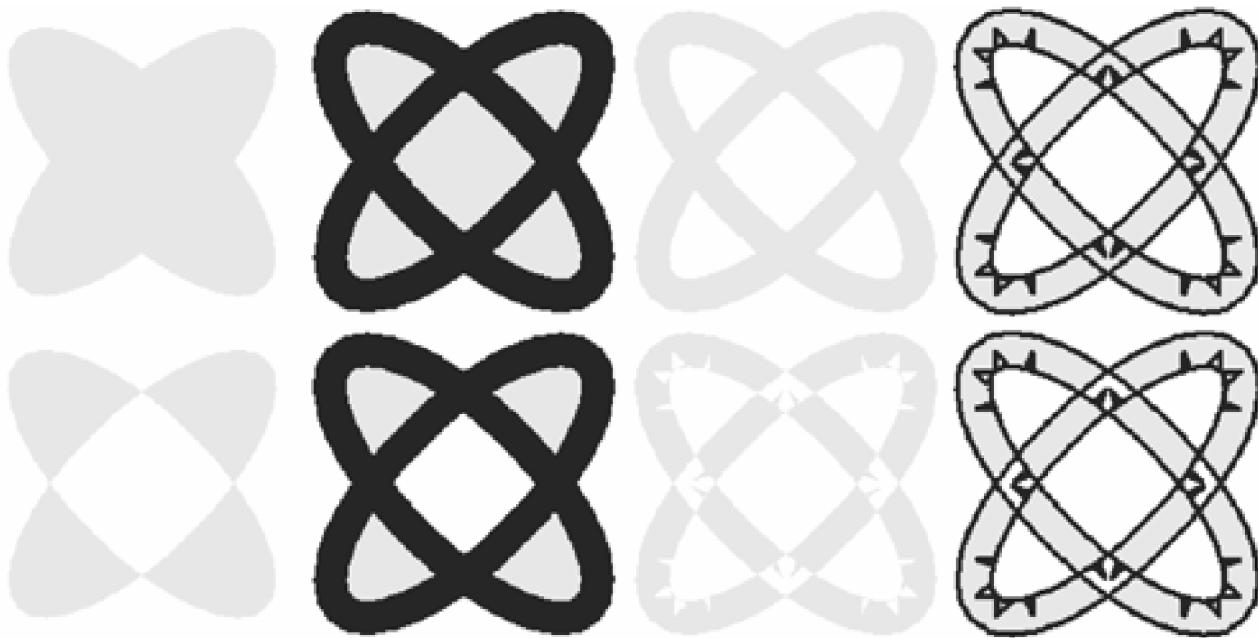
The FillPath function closes all open figures in the current path implicitly attached to the device context, and fills their interior with the current brush selected in the device context. Recall that a path is made up of one or multiple sequences of lines and curves. If the curves are all approximated using lines, a path is just like multiple polygons. The drawing behavior of FillPath is just like the PolyPolygon function with a null pen. Polygon fill mode determines whether a point is inside the painting area in the same way as PolyPolygon. Before FillPath returns to its caller, the path object in the device context is released.

The StrokeAndFillPath function closes all open figures in the current path, fills their interior with the current brush, and draws the outlines with the current pen. So StrokeAndFillPath is very similar to the PolyPolygon function. Again, before StrokeAndFillPath returns to its caller, the path object in the device context is released. Note that StrokeAndFillPath can't easily be replaced with calling StrokePath and FillPath—for two reasons. First, either call releases the path, so the next call will fail unless the device context is saved and restored. Second, calling StrokePath and FillPath separately generates overlapping areas, which may result in different effects when certain raster operations are used.

A polygon or multiple polygons can always be converted to a path, without losing any accuracy. A path without any curves can be converted to multiple polygons. A path with curves can be approximated using lines by calling FlattenPath and then converted to multiple polygons. But approximating curves using lines loses accuracy and generates more data for GDI. So path functions should be preferred over polygon functions when possible.

[Figure 9-14](#) illustrates FillPath, FillAndStrokePath, and their interaction with WidenPath and polygon fill modes.

Figure 9-14. Fillpath, StrokeAndFillPath, and polygon fill modes.



We are examining eight different cases here. The path in each case is made up of two overlapping ellipses rotated 45 degrees. The first row is generated in the WINDING fill mode, the second row in the ALTERNATE fill mode. The first column is drawn using FillPath with a light color brush, the second column using StrokeAndFillPath with a dark thick pen. The third column is obtained by calling WidenPath with the thick pen, and then FillPath. The last column uses WidenPath with the same thick pen, followed by StrokeAndFillPath with a thin pen. Recall our mentioning in [Chapter 8](#) that WidenPath generates a totally new path that represents the outline of the original path, and spikes in the new path are the result of connecting lines and curves.

Here is the code that generates [Figure 9-14](#). A path with two ellipses is generated once, retrieved using GetPath, rotated 45 degrees, and then played back to the device context to generate the paths that are used by the actual drawing calls.

```
void KMyCanvas::TestFillPath(HDC hDC)
{
    const int nPoint = 26; // 13 points per ellipse
    POINT Point[nPoint];
    BYTE Type[nPoint];

    // constructing a path with two ellipses
    BeginPath(hDC);
    Ellipse(hDC, -100, -40, 100, 40);
    Ellipse(hDC, -40, -100, 40, 100);
    EndPath(hDC);

    // retrieve path data and rotate 45 degrees
    GetPath(hDC, Point, Type, nPoint);

    for (int i=0; i<nPoint; i++)
    {
        double x = Point[i].x * 0.707;
        double y = Point[i].y * 0.707;
```

```
Point[i].x = (int) (x - y);
Point[i].y = (int) (x + y);
}

KGDIObject brush(hDC, CreateSolidBrush(RGB(0xFF, 0xFF, 0)));
KGDIObject pen (hDC, CreatePen(PS_SOLID, 19, RGB(0, 0, 0xFF)));

for (int t=0; t<8; t++)
{
    SetViewportOrgEx(hDC, 120+(t%4)*180, 120+(t/4)*180, NULL);

    // construct the rotated ellipses
    BeginPath(hDC);
    PolyDraw(hDC, Point, Type, nPoint);
    EndPath(hDC);

    if ( t>=4 )
        SetPolyFillMode(hDC, ALTERNATE);
    else
        SetPolyFillMode(hDC, WINDING);
    switch ( t % 4 )
    {
        case 0: FillPath(hDC); break;
        case 1: StrokeAndFillPath(hDC); break;
        case 2: WidenPath(hDC); FillPath(hDC); break;
        case 3: WidenPath(hDC);

        {
            KObject thin(hDC, CreatePen(PS_SOLID, 3,
                RGB(0, 0, 0xFF)));
            StrokeAndFillPath(hDC);
        }
    }
}
SetViewportOrgEx(hDC, 0, 0, NULL);
}
```

The motto of the story is, WidenPath followed by FillPath in the WINDING fill mode is the same as StrokePath.

9.6 REGIONS

In [Chapter 7](#), we had a simple discussion on regions, while focusing on a region's usage in clipping. In Win32 GDI, a region is both a very important data structure and a drawing mechanism. We will take a detailed look at regions and their usage in this section.

Regions are extremely important in Windows programming. They are being used in the following areas, some of which we have already covered in [Chapter 7](#).

1. Defining window shape: SetWindowRegion.
2. Maintaining areas on windows that need repainting: InvalidateRgn, Get UpdateRgn.
3. Clipping: SelectClipRgn, SetMetaRgn.
4. Drawing: a region can be painted directly.
5. Geometric testing: a region can be used to represent a geometric shape.
6. DirectDraw: IDirectClipper uses region data structure.

A region in GDI is a set of points in a coordinate space. The set can be empty, it can cover the whole coordinate space, it can cover a rectangle area, or it can be any irregular shape. A region object is an object managed by GDI that represents a region. As with other GDI objects, after a region object is created, a user application gets only its handle, which can be passed back to GDI for future reference. The data type for GDI region handles is HRGN. The internal data structure of a region object is quite complicated, and potentially quite big in size. After a region object is no longer needed, it should be deleted with the DeleteObject function.

Region Object Creation

The following functions are provided to create region objects from scratch:

```
HRGN CreateRectRgn(int nLeftRect, int nTopRect,  
                    int nRightRect, int nBottomRect);  
HRGN CreateRectRgnIndirect(CONST RECT * lprc);  
HRGN CreateRoundRectRgn(int nLeftRect, int nTopRect,  
                        int nRightRect, int nBottomRect,  
                        int nWidthEllipse, int nHeightEllipse);  
HRGN CreateEllipticRgn(int nLeftRect, int nTopRect,  
                      int nRightRect, int nBottomRect);  
HRGN CreateEllipticRgnIndirect(CONST RECT * lprc);  
HRGN CreatePolygonRgn(CONST POINT * lppt, int cPoints,  
                      int fnPolyFillMode);
```

```
HRGN CreatePolyPolygonRgn(CONST POINT *lppt, CONST INT *
    IpPolyCounts, int nCount, int fnPolyFillMode);
HRGN PathToRegion(HDC hDC);
```

All the functions in this group, except the last PathToRegion function, do not depend on a device context, a pen, or a brush. A region object is an independent object used to represent a geometric shape. When it's used in a different context, coordinates in a region are interpreted either as logical coordinates or device coordinates.

The CreateRectRgn function creates a region made of all the points within a rectangle area, defined normally by its top left corner and bottom right corner. The two points defining a rectangle do not need to be well-ordered; GDI will reorder them to make sure that left is smaller than right and top is smaller than bottom for GDI's internal representation of a region. CreateRectRgnIndirect is a simple variant of CreateRectRgn that passes parameters through a RECT structure. On Windows NT/2000, CreateRectRgnIndirect implementation simply calls CreateRectRgn.

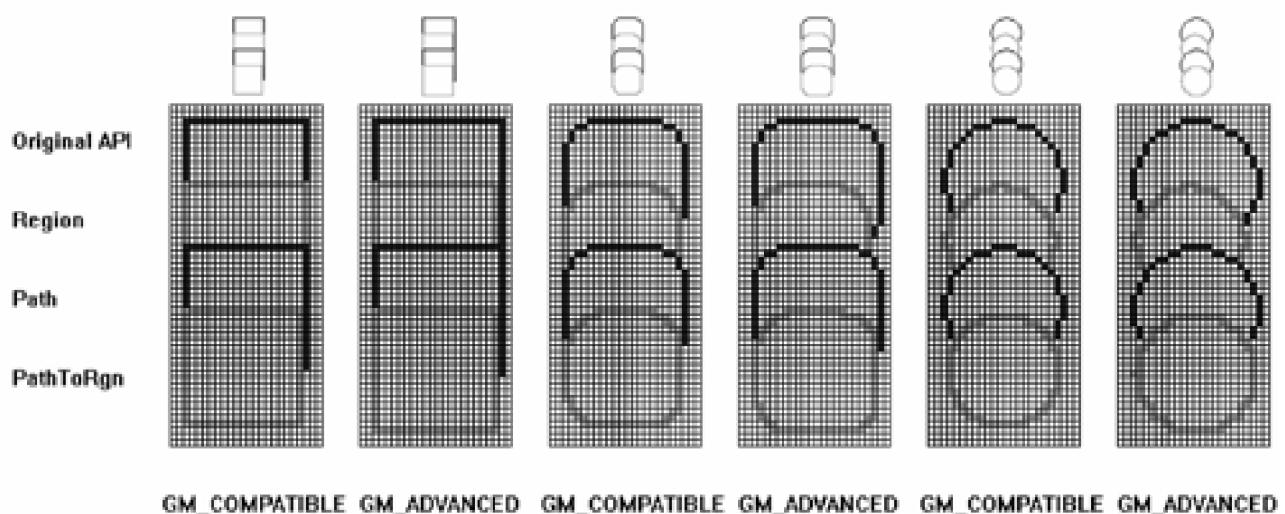
When a region object is used, it's always interpreted as top, left edge inclusive and bottom, right edge exclusive. This is true in both the compatible and advanced graphics modes. For example, CreateRectRgn(0, 0, 0, 0) creates an empty region, instead of a region with a single point (0, 0). If used in the device coordinate space, CreateRectRgn(0, 0, 1, 1) creates a region made up of a single point (0, 0), which is equivalent to CreateRectRgn(1, 1, 0, 0).

The CreateRoundRectRgn creates a region made of all the points within a rectangle with rounded corners. Each of the four corners fits one quarter of an ellipse nWidth Ellipse by nWidthHeight in size. For some unknown reason, when a rounded rectangle region is created, its bottom edge and right edge are excluded in the internal data structure representing the region. Note here the difference with a rectangle region, which includes its bottom edge and right edge in its internal representation. So when a rounded rectangle region is used in the device coordinate space, 2-pixels-wide bottom and right edges are excluded. If it's used in the logical coordinate space, the excluded edges are 1 logical unit plus 1 device unit in width.

The CreateEllipticRgn function creates a region made of all the points within an ellipse. The CreateEllipticRgnIndirect is a simple variant, which passes the call to it. Just like CreateRoundRectRgn, CreateEllipticRgn removes the bottom, right edges in the region's internal representation. So when an elliptic region is used, its bottom, right edges are double-excluded.

Microsoft Knowledge Base, in the Windows Development section under Win32 SDK, has an article on CreateEllipticRgn's exclusiveness problem (Q83807). It mentions that Ellipse() includes the lower-right corner of the bounding rectangles in its calculations, while the CreateEllipticRgn function excludes the lower-right point. We are seeing something different from what *Microsoft Knowledge Base* says. According to [Figure 9-8](#), Ellipse does exclude the right, bottom edges in compatible graphics mode. [Figure 9-15](#) shows that CreateEllipticRgn always excludes 1 logical unit more than Ellipse() in compatible graphics mode.

Figure 9-15. Regions covered by CreateRectRgn, CreateRoundRectRgn, and CreateEllipticRgn.



[Figure 9-15](#) shows a rectangle, a rounded rectangle, and an ellipse drawing using four different methods, and compares their pixel-level details. Drawing these three basic shapes can use direct GDI API calls, create a region first and then draw the region, convert to a path and then draw the path, or convert to a path then to a region and then draw the region. The tests are done both in compatible and advanced graphics mode.

[Figure 9-15](#) shows that `Rectangle`, `Ellipse`, and `RoundRectangle` are right-bottom exclusive in compatible mode, but right-bottom inclusive in advanced mode. `CreateRectRgn` is always right-bottom exclusive. But `CreateRoundRectRgn` and `CreateEllipticRgn` are double right-bottom exclusive. What's not shown here is that the shape generated by `CreateEllipticRgn` is not the same shape painted using the `Ellipse` function, even if you increase its size carefully by one device unit. If consistency is critical—for example, when the created region is used to clip `Ellipse` function call—Microsoft suggests using the `Region` function.

The `CreatePolygonRgn` function creates a region made up of all the points within a polygon. As we mentioned in [Section 9.4](#), polygon fill mode determines whether a point is within a polygon, and GDI supports two polygon fill modes. To be independent of device contexts, `CreatePolygonRgn` accepts a polygon fill mode parameter as its last parameter. The `CreatePolyPolygonRgn` function creates a region made up of multiple polygons. Both functions include all the coordinates in their internal representation of regions. But when the regions are painted, the right bottom edges are excluded. So the area covered by `CreatePolygonRgn`'s result is smaller than the area covered by calling `Polygon` with the same parameters.

The last function to create a region is `PathToRegion`, which converts the current path in a device context to a region. A path object is a strange GDI object in that it's always attached to a device context at the GDI API level. So GDI takes the liberty to store path objects in device coordinates instead of logical coordinates. The `PathToRegion` function closes all open figures in the path and converts them to a region according to the current polygon fill mode in the device context. Note that the region created uses the device coordinate space of the device context, unlike the `GetPath` function, which uses inverse transformation to convert path data from the device coordinate space back to the logical coordinate space. Although `PathToRegion` uses all the original coordinates in generating the region, when the region is used, the right, bottom edges are excluded.

To summarize, there are three reasons why the regions created by the region creation function provided by GDI do not cover exactly the same areas covered by the corresponding GDI drawing function. First, `CreateRectRgn`, `CreateRectRgnIndirect`, `CreatePolygonRgn`, `CreatePolyPolygonRgn`, and `PathToRegion` use the original coordinates in generating the internal representation of a region, but `CreateRoundRectRgn`, `CreateEllipticRgn`, and `CreateEllipticRgnIndirect` deduct one unit on the right and bottom edges of the bounding box when generating the region's internal representation. The latter case should be considered a GDI defect instead of a design decision or a feature. Second, when a region is used in a device context, either for clipping or for painting, its rightmost and

bottommost edges are always excluded. Third, region creation functions behave the same way in both graphics modes, while rectangle, rounded rectangle, and ellipse drawing functions include bottom and right edges in advanced graphics mode.

Operations on Region Objects

A region is a set of points in a two-dimensional coordinate space, so naturally set-related operations can be easily defined on region objects. GDI provides quite a wealthly set of functions to query, move, transform, reset, and combine regions. Here are the related functions.

```
BOOL PtInRegion(HRGN hrgn, int X, int Y);
BOOL RectInRegion(HRGN hrgn, CONST RECT * lprc);
BOOL EqualRgn(HRGN hSrcRgn1, HRGN hSrcRgn2);
int GetRgnBox(HRGN hrgn, LPRECT lprc);

int CombineRgn(HRGN hrgnDest, HRGN hrgnSrc1, hrgnSrc2,
    int fnCombineMode);

int OffsetRgn(HRGN hrgn, int nXOffset, int nYOffset);
DWORD GetRegionData(HRGN hRgn, DWORD dwCount,
    LPRGNDATA lpRgnData);
HRGN ExtCreateRegion(CONST XFORM * lpXForm, DWORD nCount,
    CONST RGNDATA * lpRgnData);
```

Querying a Region

The PtInRegion checks if a point (x, y) is within the set of points in a region. The right and bottom edges of a region are considered out of the region. For example, for the empty region created by CreateRectRgn(0, 0, 0, 0), PtInRegion always returns FALSE; for the single-point region created by CreateRectRgn(0, 0, 1, 1), PtInRegion only returns TRUE for point (0, 0).

The PtInRegion function is very useful for implementing fancy style buttons or clickable areas that change color when the mouse cursor is moved onto them to indicate that they are clickable. An application just needs to create a region object corresponding to the clickable area and call PtInRegion in WM_MOUSEMOVE message handling to change the area display accordingly. Similar processing needs to be added to mouse-click event handling to take the necessary actions when a certain clickable area is hit.

[Listing 9-1](#) shows a generic class KButton for implementing clickable buttons and two derived classes for rectangular and elliptic buttons. The DefineButton routine specifies the bounding box of a button. The DrawButton virtual function creates a region object and draws the button according to whether the mouse is on the button. The IsOnButton function uses PtInRegion to test whether point (x, y) is on the button. The UpdateButton function updates the button display according to the current mouse position.

Listing 9-1 KButton class: Clickable Areas

```
class KButton
```

```
{  
protected:  
    HGN m_hRegion;  
    bool m_bOn;  
    int m_x, m_y, m_w, m_h;  
  
public:  
  
    KButton()  
    {  
        m_hRegion = NULL;  
        m_bOn = false;  
    }  
  
    virtual ~KButton()  
    {  
    }  
  
    void DefineButton(int x, int y, int w, int h)  
    {  
        m_x = x; m_y = y; m_w = w; m_h = h;  
    }  
  
    virtual void DrawButton(HDC hDC)  
    {  
    }  
  
    void UpdateButton(HDC hDC, LPARAM xy)  
    {  
        if ( m_bOn != IsOnButton(xy) )  
        {  
            m_bOn = ! m_bOn;  
            DrawButton(hDC);  
        }  
    }  
  
    bool IsOnButton(LPARAM xy) const  
    {  
        return PtInRegion(m_hRegion, LOWORD(xy), HIWORD(xy))!=0;  
    }  
};  
  
class KRectButton : public KButton  
{  
public:  
    void DrawButton(HDC hDC)  
    {  
        RECT rect = { m_x, m_y, m_x+m_w, m_y+m_h };  
    }  
};
```

```
if ( m_hRegion == NULL )
    m_hRegion = CreateRectRgnIndirect(& rect);

InflateRect(&rect, 2, 2);
FillRect(hDC, & rect, GetSysColorBrush(COLOR_BTNFACE));
InflateRect(&rect, -2, -2);

DrawFrameControl(hDC, &rect, DFC_CAPTION,
    DFCS_CAPTIONHELP | (m_bOn ? 0 : DFCS_INACTIVE));
}

};

class KEllipseButton : public KButton
{
public:
    void DrawButton(HDC hDC)
    {
        RECT rect = { m_x, m_y, m_x+m_w, m_y+m_h };

        if ( m_hRegion == NULL )
            m_hRegion = CreateEllipticRgnIndirect(& rect);

        if ( m_bOn )
        {
            FillRgn(hDC, m_hRegion,
                GetSysColorBrush(COLOR_CAPTIONTEXT));
            FrameRgn(hDC, m_hRegion,
                GetSysColorBrush(COLOR_ACTIVEBORDER), 2, 2);
        }
        else
        {
            FillRgn(hDC, m_hRegion,
                GetSysColorBrush(COLOR_INACTIVECAPTIONTEXT));
            FrameRgn(hDC, m_hRegion,
                GetSysColorBrush(COLOR_INACTIVEBORDER), 2, 2);
        }
    }
};

};

The sample code shown below displays a client area with two clickable buttons that change color depending on whether the mouse is on them. When the mouse is clicked on either of them, a message box is displayed.
```

```
LRESULT KMyCanvas::WndProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    // KRectButton rbtn, KEllipseButton ebtn defined in class
    switch( uMsg )
```

```
{  
    case WM_CREATE:  
        rbtn.DefineButton(10, 10, 50, 50);  
        ebtn.DefineButton(10, 70, 50, 50);  
        return 0;  
  
    case WM_PAINT:  
    {  
        PAINTSTRUCT ps;  
        HDC hDC = BeginPaint(m_hWnd, &ps);  
        rbtn.DrawButton(hDC);  
        ebtn.DrawButton(hDC);  
        EndPaint(m_hWnd, &ps);  
    }  
    return 0;  
  
    case WM_MOUSEMOVE:  
    {  
        HDC hDC = GetDC(hWnd);  
        rbtn.UpdateButton(hDC, IParam);  
        ebtn.UpdateButton(hDC, IParam);  
        ReleaseDC(hWnd, hDC);  
    }  
    return 0;  
  
    case WM_LBUTTONDOWN:  
    if ( rbtn.IsOnButton(IParam) )  
        MessageBox(hWnd, "Button1 Clicked", NULL, MB_OK);  
    if ( ebtn.IsOnButton(IParam) )  
        MessageBox(hWnd, "Button2 Clicked", NULL, MB_OK);  
    return 0;  
  
    default:  
        return DefWindowProc(hWnd, uMsg, wParam, IParam);  
    }  
}
```

The next function, RectInRegion, checks whether any part of a rectangle specified by the lprc parameter, excluding its right, bottom edges, is within a region. Note here that the function does not check if the whole rectangle is within a region, so maybe the function should be renamed as RectTouchRegion.

The EqualRegion compares two regions to see if they have the same set of points. If the two parameters are exactly the same region object handle, of course they are the same regions. But if they are not the same handles, it's still possible they represent the same set of points. For example, CreateRectRgn(0, 0, 0, 0) and CreateRectRgn(1, 1, 1, 1) both create empty regions, which are considered equal by the EqualRegion function. From the fact that the EqualRegion function is provided, you can be pretty sure that region objects are uniquely represented internally, in that one set of points has only one representation. Otherwise, EqualRegion will be very slow in performance.

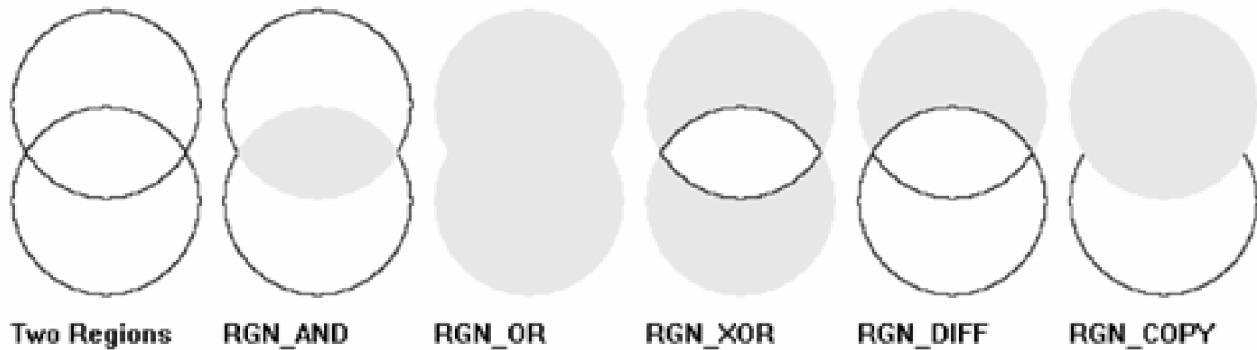
The GetRgnBox function returns the bounding box of a region. For an empty set, its bounding box is always {0, 0, 0,

0). For other rectangular regions, its bounding box is the original defining rectangle normalized to make the left smaller than the right and the top smaller than the bottom. For an elliptic region or a rounded rectangle region, we mentioned that GDI removes one unit from the bottom edge and right edge, so the bounding box is smaller than the defining bounding box. For example, CreateElliptic Rgn(10, 10, 1, 1) returns a region whose bounding box is {1, 1, 9, 9}. The region bounding box is a quick way to test if a point or area touches a region, which is especially important if performance is important. The rectangle returned by GetRgnBox allows the application to do a quick test without involving GDI call and user-mode to kernel-mode transition. The rcPaint rectangle within the PAINT STRUCT structure filled by Begin Paint is the bounding rectangle for the window's system region. It's used by most applications to test if a certain object needs to be redrawn in the WM_PAINT message handling.

Set Operations on Regions

The CombineRgn function provides several useful set-like operations on region objects. The function accepts three region objects, hrgnDest, hrgnSrc1, hrgnSrc2, and an integer parameter, fnCombineMode. When calling CombineRgn, hrgnDest must be a valid region object handle. The function replaces the region object represented by the handle with the region object generated by the function call. The fnCombine Mode parameter specifies how regions represented by hrgnSrc1 and hrgnSrc2 should be combined together, using either copy, set intersection, union, difference, or symmetric difference. [Table 7-1](#) summarizes the five region combine modes [Figure 9-16](#) illustrates these region operations.

Figure 9-16. Region combine modes.



Both GetRgnBox and CombineRgn return an integer result to indicate the complexity of the region generated, or an error condition. [Table 9-1](#) summarizes the possible results.

If a region is a set of points, what is its universal set—that is, the set of all points? For Win32 GDI, logical coordinates are limited to 32-bit integer values, device coordinates on NT-based systems are limited to 27-bit nonnegative integer values, and non-NT-based systems further restrict the coordinates to 16-bit integer values. So a rectangular region with the bounding box [0x80000000, 0x80000000, 0x7FFFFFFF, 0x7FFFFFFF] should be the universal set for regions. But GDI on an NT-based system seems to truncate the numbers to 28-bit signed integer values, which limit the universal set's bounding box to be $[-(1 \ll 27), -(1 \ll 27), (1 \ll 27) - 1, (1 \ll 27) - 1]$. Another undocumented restriction, if you're using region functions in logical coordinate spaces, is that they are restricted to using 28-bit signed integers instead of 32-bit signed integers.

Table 9-1. GetRgnBox, CombineRgn Results

Value	Meaning
NULLREGION	The region is an empty region.
SIMPLEREGION	The region can be described using a single rectangle.
COMPLEXREGION	The region can be described using multiple rectangles.
ERROR	Error occurs: invalid input parameters or out of memory. No region generated.

Set operations on regions are very useful for geometric calculations. If you want to test whether two closed paths overlap with each other, one way to do that is to convert the paths for the paths to polygons using FlattenPath, and implement a polygon overlapping test algorithm. But it is not that simple. With a region, you can convert paths to regions and calculate their intersection using CombineRgn(RGN_AND). If their intersection is not empty, the original two paths overlap with each other. This collision test is essential for game programming, where actions are triggered when a flying object touches a target. Set operations can also be used to test if a region fully contains another region. We mentioned that the RectInRegion function actually tests if a rectangle touches a region. The following function really tests if a rectangle is fully contained by a region. It uses CombineRgn to find the union of a region and a rectangle, and uses EqualRgn to check if the combined region is the same as the original region.

```
BOOL RectContainedInRegion(HRGN hrgn, CONST RECT * lprc)
{
    HRGN hCombine = CreateRectRgnIndirect(lprc);
    CombineRgn(hCombine, hrgn, hCombine, RGN_OR);
    BOOL rslt = EqualRgn(hCombine, hrgn);
    DeleteObject(hCombine);
    return rslt;
}
```

Transforming Region Data

GDI supports translation, reflection, and scaling between the page coordinate space and the device coordinate system. NT-based GDI supports more general affine transformation between the world coordinate space and the page coordinate space, which adds rotation and shearing. All these transformations are supported on region objects, with the expected exception that rotation and shearing are only supported directly on NT-based systems.

The OffsetRgn function provides the most basic transformation: translation. It accepts an offset on the x-axis and an offset on the y-axis, adds them to every coordinate in the region object, and returns a region complexity number. It can be used to draw objects repetitively on a device surface, either using the region to paint or using it to clip painting. An application can use the region once, move it to another place using OffsetRgn, and use it again. It's also useful to keep track of moving objects in a game or an animation program. If a region is used to represent a car in a racing game, when the car moves so should its region, so that a collision can be detected using the region.

A pair of functions provides the more general transformations: GetRegionData and ExtCreateRegion. GetRegionData converts the internal data structure for a region into a user-accessible data structure, RGNDATA structure. ExtCreateRegion accepts a RGNDATA structure and an XFORM structure for an affine transformation, transforms the data, and creates a new region. Here is the critical RGNDATA structure.

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize; // sizeof(RGNDATAHEADER)
    DWORD iType; // RDH_RECTANGLES
    DWORD nCount; // no of rectangles in the region
    DWORD nRgnSize; // total region size
    RECT rcBounds; // bounding rectangle
} RGNDATAHEADER;
```

```
typedef struct _RGNDATA {
    RGNDATAHEADER rdh;
    char     Buffer[1]; // variable length
} RGNDATA;
```

There are several interesting things to note about this RGNDATA structure. Just for the record, RGNDATA is not the internal data structure used by GDI to represent a region. NT-based systems use a more memory-efficient dynamic array structure, REGIONOBJ, to represent a region, which contains an array of SCAN structures. A SCAN structure represents the intersection of a region with the area between two scan lines, $y = \text{top}, y = \text{bottom}$, under the restriction that the x -coordinates of all the intersections of the region boundary and all the scan lines between top and bottom are constants. A SCAN structure contains an array of x -coordinates for these intersections, always even in number. There is no evidence to verify that the region is represented using a trapezoid, as Microsoft documentation claims. A SCAN-based data structure for the region object shares the y -coordinates for possible multiple intersections, which saves memory. It also enforces top-down order in the array of a SCAN structure, and left-to-right order with a SCAN structure, for unique representation of regions and efficient region operations. For example, if CombineRgn is used to combine multiple small regions into a big region, its internal representation should be independent of the order in which the regions are combined. For details, refer to [Section 3.7](#). Non-NT-based systems use 16-bit coordinates instead of 32-bit coordinates.

The memory requirement for a REGIONOBJ structure depends on the complexity of a region. Suppose a region can be divided into N scan lines, and the average number of intersections per scan line is M . Note that a scan line is at least one unit in height, but it could be multiple units in height. The size of a REGIONOBJ can be expressed as:

$$\text{sizeof(REGIONOBJ)} = (4 * M + 16) * (N + 2) + 40$$

For a rectangle region, one scan line covers the whole rectangle, so $N = 1$, $M = 2$, only 112 bytes. GDI could only store its bounding rectangle and set a special flag saying it's a simple region. For an elliptic region, the number of scan lines needed is close to ? the height of the ellipse, $M = 2$. If you're creating a region for a full-page ellipse on a 600-dpi printer, $N = ? * 11 * 600 = 4400$, $\text{sizeof(REGIONOBJ)} = 111$ KB.

An application could use a region to implement color keying for a bitmap by creating a region from the bitmap with all pixels not of the same color as the color key. Such a region can be used as a clipping region for the bitmap drawing, so that pixels with the same color as the color key are not drawn or are transparent. In the worst case, N is the height of the bitmap, M is half the width of the bitmap, and the size of the REGION OBJ is 2 bytes for each pixel in the bitmap. An application using regions extensively should be aware of these memory costs to the system.

The graphics engine actually allocates more memory than is needed to represent the region to accommodate possible region growth. This is the same strategy dynamic array uses to minimize dynamic memory allocation and memory copy.

The REGIONOBJ structure is basically a two-dimensional structure; the first dimension is an array of SCAN structures sorted according to y , and the second dimension is an array of x -coordinates within the SCAN structures

sorted according to x. Such a design allows moderate performance in region operation. For example, to check if a point is within a region, if it passes the bounding-box test, the point's y-coordinate can be compared with the y-coordinate of each SCAN structure using a linear search. Each SCAN structure keeps its size at both ends, so it's quite easy to move from one SCAN structure to the next one. After the right SCAN is found, another linear search can be conducted using the x-coordinate. So the complexity of PtInRegion is $O(N) + O(M)$, using the big O notation. The best algorithm should be $O(\log(N)) + O(\log(M))$ using binary search, but this requires a more complicated data structure. GDI is trying to use multiple small data structures with pointers linking them together in order to minimize dynamic allocation. CombineRgn for region union, intersection, and difference will conceivably have similar complexity in terms of the number of comparisons needed. Extra time is needed to copy data for the new region. So if you are using CombineRgn to combine n regions together, the worst-case time complexity is $O(n^2)$. Every time the number of regions is doubled, the time needed quadruples. Such algorithms should be avoided, if possible.

The RGNDATA structure is GDI's way of providing a unified interface to Win32 applications running on different platforms. The RGNDATA structure is also used by DirectDraw IDirectDrawClipper interface.

It has a fixed size header with size, type, and the bounding rectangle, and an array of RECT structure. The RGNDATA structure is a flattened version of GDI's internal two-dimensional data structure into a one-dimensional data structure. For a region with N scan lines, and M average intersections per scan line, $M/2 * N$ rectangles are needed. The total memory needed is:

$$\text{sizeof(RGNDATA)} = 8 * M * N + 32$$

For a region with a single convex figure—for example, a single rectangle, rounded rectangle, or one ellipse— $M = 2$, so a RGNDATA structure is about ? the size of a REGIONOBJ structure. When M becomes quite big, a RGNDATA structure is about double the size of a REGIONOBJ structure.

Like the REGIONOBJ structure, the RGNDATA structure generated by GDI is always well-ordered. The RECT structure in it comes in left-to-right, top-down order. Each RECT structure always has left smaller than right, and top smaller than bottom.

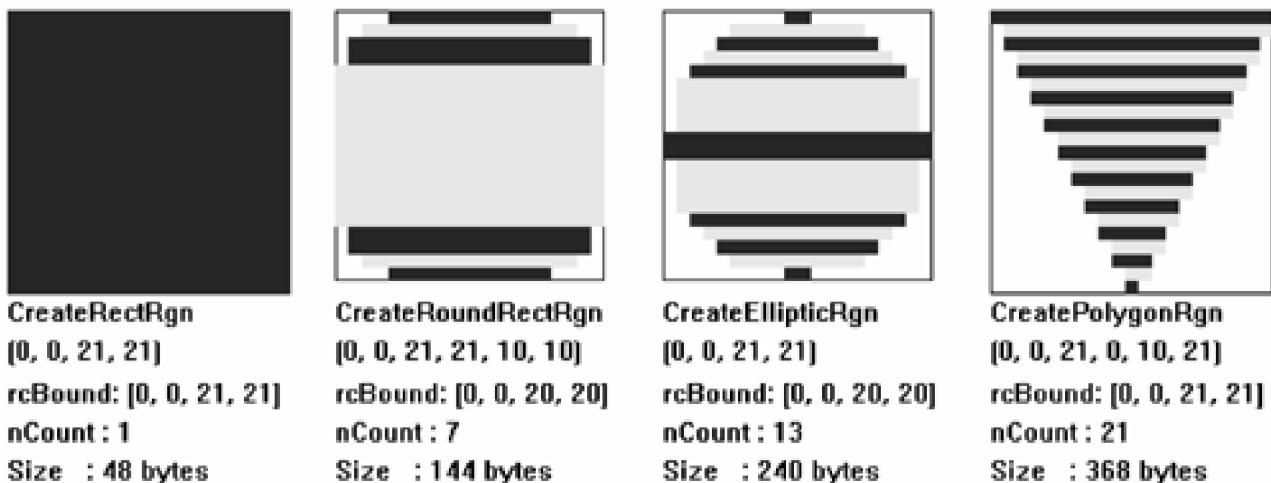
Because RGNDATA is a linear array of rectangles, certain algorithms could be made faster, if an application wants to implement them. For example, to check if a rectangle touches a region represented using a RGNDATA structure, you can do a binary search on the rectangle array, instead of a linear search, reducing the time complexity to $O(\log(M * N))$. But set operations using CombineRgn still involve data copy that is linear to the size of data in time complexity.

The GetRegionData function fills a user-supplied buffer with a RGNDATA structure. But before calling GetRegionData, an application normally does not know the exact size of the structure. An application can make a guess on the size of the RGNDATA that could cover 80% of the cases, allocate a buffer of that size (possibly on the stack), and call the GetRegionData with the buffer size and pointer. If the buffer is big enough, it will be filled with a RGNDATA structure, and the buffer size will be returned as a function result. If the buffer is too small, GetRegionData returns 0; RGNDATA is not generated. Then the application should call GetRegionData with 0 as dwCount, and NULL as lpRgnData; GDI will return the size of the actual buffer needed. The application does the allocation, mostly from a heap, and calls GetRegionData again with the buffer size and pointer to get the real data. An application can skip the first try to call, going directly to call GetRegionData to get the size. But minimum heap allocation is a good practice of programming.

[Figure 9-17](#) illustrates RGNDATA for rectangle, rounded rectangle, ellipse, and triangle regions; all have the same defining bounding box {0, 0, 21, 21} . The rectangle region contains a single rectangle; the rounded rectangle region contains 7 rectangles, mainly for the rounded corners; the elliptic region has 13 rectangles; and the triangle region has 21 rectangles. The bounding boxes for RGNDATA are painted in black pen. The RECT array within is painted

with alternative dark and light color bars.

Figure 9-17. RGNDATA for different kinds of regions.



The ExtCreateRegion function allows creating a region object from a RGNDATA structure, with a possible affine transformation applied to the region object. The RGNDATA structure can either be obtained directly from GDI using GetRegionData, a modified version of region data gotten from GDI, or generated by an application from scratch. When calling ExtCreateRegion, each RECT structure should have the left smaller than the right and the top smaller than the bottom. The RGNDATA's rcBounds rectangle should be the bounding box of the rectangles. Otherwise, inconsistent regions could be generated or region creation could fail.

The first parameter to the ExtCreateRegion function is a pointer to an affine transformation matrix, although non-NT-based systems do not allow shearing and rotation in the transformation. Transforming a region could be very useful for applications. For example, the region returned by PathToRegion is in the device coordinate space. If an application wants to paint the region instead of using it for clipping, it needs to be transformed back to the logical coordinate space. The OffsetRgn function can handle a simple translation, but more general transformations need ExtCreateRegion.

A RGNDATA structure represents a region using integer coordinates linked by lines instead of curves, much like a flattened path. So any scaling up should expect jagged edges. If a region can be defined using a path, transforming the path and converting to a region could generate a more accurate result.

It has been reported that ExtCreateRegion can't handle more than 4000 rectangles at a time on non-NT-based systems. The workaround is to divide a big RGNDATA into multiple smaller ones, have each call ExtCreateRegion once, and then combine the results using CombineRgn.

[Listing 9-2](#) shows a simple wrapping class around the GetRegionData and ExtCreateRegion functions.

Listing 9-2 KRegion Class: Access Region Data

```
class KRegion
{
public:
    int    m_nRegionSize;
```

```
int      m_nRectCount;
RECT *   m_pRect;
RGNDATA * m_pRegion;

KRegion()
{
    m_nRegionSize = 0;
    m_nRectCount = 0;
    m_pRegion   = NULL;
    m_pRect     = NULL;
}

void Reset(void)
{
    if ( m_pRegion )
        delete [] (char *) m_pRegion;
    m_pRegion   = NULL;
    m_nRegionSize = 0;
    m_nRectCount = 0;
    m_pRect     = NULL;
}

~KRegion()
{
    Reset();
}

BOOL GetRegionData(HRGN hRgn);
HRGN CreateRegion(XFORM * pXForm);
};

BOOL KRegion::GetRegionData(HRGN hRgn)
{
    Reset();

    m_nRegionSize = ::GetRegionData(hRgn, 0, NULL);
    if ( m_nRegionSize==0 )
        return FALSE;

    m_pRegion = (RGNDATA *) new char[m_nRegionSize];
    if ( m_pRegion==NULL )
        return FALSE;

    ::GetRegionData(hRgn, m_nRegionSize, m_pRegion);

    m_nRectCount = m_pRegion->rdh.nCount;
    m_pRect     = (RECT *) & m_pRegion->Buffer;

    return TRUE;
}
```

```
}

HRGN KRegion::CreateRegion(XFORM * pXForm)
{
    return ExtCreateRegion(pXForm, m_nRegionSize, m_pRegion);
}
```

The GetRegionData and ExtCreateRegion functions also open the door for applications to generate and transform the RGNDATA structure on their own and pass to GDI to create regions. This could be useful if an application wants to implement region rotation or shearing on non-NT-based systems, or if to overcome the $O(N^2)$ performance bottleneck the CombineRgn function has to combine N regions.

Painting Using Regions

Given a region object handle, GDI provides several functions to paint the area covered by the region.

```
BOOL FillRgn(HDC hDC, HRGN hrgn, HBRUSH hbr);
BOOL PaintRgn(HDC hDC, HRGN hrgn);
BOOL FrameRgn(HDC hDC, HRGN hrgn, HBRUSH hbr,
    int nWidth, int nHeight);
BOOL InvertRgn(HDC hDC, HRGN hrgn);
```

All these functions take a device context handle and a region handle. The coordinates in the region are in the logical coordinate space, instead of in the device coordinate space as clipping regions are. So a region handle returned by PathToRegion can't be passed to these functions directly, unless the logical and device coordinate spaces are identical or it is intentional. The graphics engine transforms the region object into the device coordinate space, where the right, bottom edges exclusion rule is enforced.

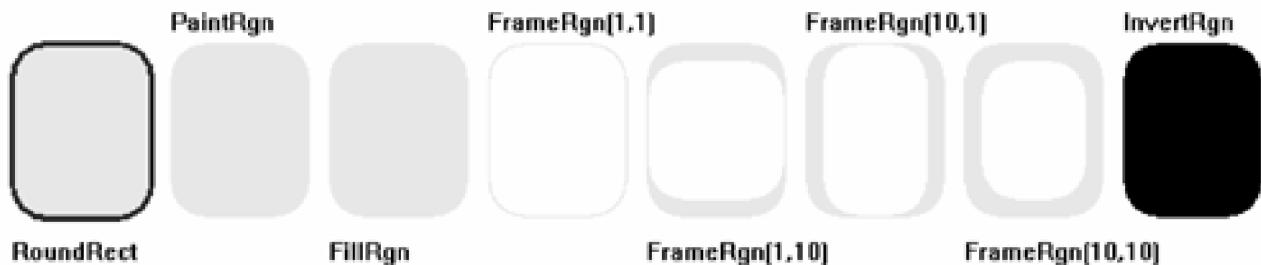
The FillRgn function fills the set of points in the region with the brush specified by the hbr parameter. The PaintRgn does the same filling using the current brush object in the device context. The two functions share the same implementation in GDI.

The FrameRgn function draws the border of a region using a brush, with an adjustable brush width and height. It could generate some interesting nonuniform borders, more than what's possible with a normal pen, which always generates lines with uniform width. The FrameRgn function treats its "pen" as a rectangle parallel to the axes, which always draws inside the region, never out.

The InvertRgn function inverts the pixels in the device's frame buffer in the same way as the raster operation R2_NOT does. It's similar to the InvertRect function.

The first three functions, FillRgn, PaintRgn, and FrameRgn, use the current binary raster operation and follows the background mode. InvertRgn uses R2_NOT to invert the whole region. [Figure 9-18](#) illustrates their effects on a rounded rectangle region, in comparison with the RoundRect function.

Figure 9-18. PaintRgn, FillRgn, FrameRgn with different width, and InvertRgn.



Although these functions look similar to other GDI functions, region functions have their hidden cost in region object creation and manipulation, both in time and memory usage, especially when the region gets complicated. If the shape of the region can be covered easily by direct GDI drawing functions like rectangle, ellipse, rounded rectangle, and path drawing functions, these functions should be preferred. The region painting function should be used to replace more expensive functions—for example, pixel-drawing API or flood fills. For example, suppose an application draws two overlapping circles on the screen, and would like to paint the overlapping area using a certain brush. Calculating the exact two arcs forming the boundary of the overlapping area is not so easy with current GDI support, but calculating the intersection of two circular regions is very easy with GDI.

9.7 GRADIENT FILLS

Up until now, area fill supported by GDI has allowed applications to fill an area with a single-color solid-color brush, a dual-color hatch brush, or a bitmap brush with as many colors as are in the bitmap. As display cards and printers with high color depth become more pervasive, applications tend to use more colors to achieve more compelling effects. Gradient fills are simple color effects that fill an area with a wide range of colors generated following certain patterns. Gradient fills started from professional applications like Photoshop, CorelDraw, Microsoft Office Suite, and now are part of GDI after Windows 98 and Windows 2000.

Win32 GDI now provides a single function supporting gradient fills, which depends on three new structures:

```
typedef struct _TRIVERTEX {
    LONG x;
    LONG y;
    COLOR16 Red;
    COLOR16 Green;
    COLOR16 Blue;
    COLOR16 Alpha;
} TRIVERTEX, * PTRIVERTEX, * LPTRIVERTEX;

typedef struct _GRADIENT_TRIANGLE {
    ULONG Vertex1;
    ULONG Vertex2;
    ULONG Vertex3;
} GRADIENT_TRIANGLE, *PGRADIENT_TRIANGLE, *LPGRADIENT_TRIANGLE;
typedef struct _GRADIENT_RECT {
    ULONG UpperLeft;
    ULONG LowerRight;
} GRADIENT_RECT, *PGRADIENT_RECT, *LPGRADIENT_RECT;

BOOL GradientFill(HDC hDC, CONST PTRIVERTEX pVertex,
    DWORD dwNumVertex, CONST PVOID pMesh,
    DWORD dwNumMesh, DWORD dwMode);
```

The GradientFill function has several new things in terms of API design. First, the “long” or “far” prefixes before the pointer types that are inherited from the Win16 API are dropped. Second, there is a sense that the traditional 3-channel RGB color format is not enough; now an alpha channel is added. Third, 8 bits per channel is felt to be insufficient, so a 16-bit color channel is used. All these are signs of progress toward a better API.

The GradientFill function fills either one or multiple rectangles, or one or multiple triangles, which is determined by the last parameter dwMode. Currently, dwMode has three valid values, as shown in [Table 9-2](#).

Each rectangle or triangle is called a mesh, a jargon term from computer game graphics. The number of meshes is held in dwNumMesh; pMesh points to either an array of GRADIENT_RECT structures, or an array of GRADIENT_TRIANGLE structures. Each GRADIENT_RECT structure holds the indexes of a rectangle's up per left corner and bottom right corner vertices. Each GRADIENT_TRIANGLE holds the indexes of a triangle's three vertices. The vertex indexes are indexes into an array of TRIVERTEX structures, pointed to by the pVertex

parameter. The dw Num Vertex parameter is the number of TRIVERTEXs in the array.

Table 9-2. GradientFill Modes

dwMode Value	Meaning
GRADIENT_FILL_RECT_H	Fill rectangles with colors varying gradually from left to right, while from top to bottom colors stay constant.
GRADIENT_FILL_RECT_V	Fill rectangles with colors varying gradually from top to bottom, while from left to right colors stay constant.
GRADIENT_FILL_RECT_TRIANGLE	Fill triangles with colors interpolated from their three vertices.

So for each rectangle or triangle, for each of its vertices, there is a TRIVERTEX structure that defines its location and color. Its location is specified in the logical coordinate space using 32-bit values. Its color is specified in four 16-bit channels, green, blue, and alpha.

For a horizontal gradient rectangle fill, if (x_0, y_0) is the top-left vertex, (x_1, y_1) is the bottom-right, then point (x, y) 's color is determined by:

$$C(x,y) = (C(x_1,y_1) * (x-x_0) + C(x_0,y_0) * (x_1-x)) / (x_1-x_0)$$

where $C(x, y)$ denotes one of the color channels at point (x, y) .

The vertical gradient rectangle uses a similar formula that depends on the y -coordinate:

$$C(x,y) = (C(x_1,y_1) * (y-y_0) + C(x_0,y_0) * (y_1-y)) / (y_1-y_0)$$

A triangle gradient fill is a little more complicated. If (x_0, y_0) , (x_1, y_1) , and (x_2, y_2) are its three vertices, for point (x, y) within the triangle, we can draw three lines from point (x, y) to the vertices to partition the triangle into three subtriangles. Let a_i be the area of the triangle opposite to (x_i, y_i) , color at (x, y) is defined by:

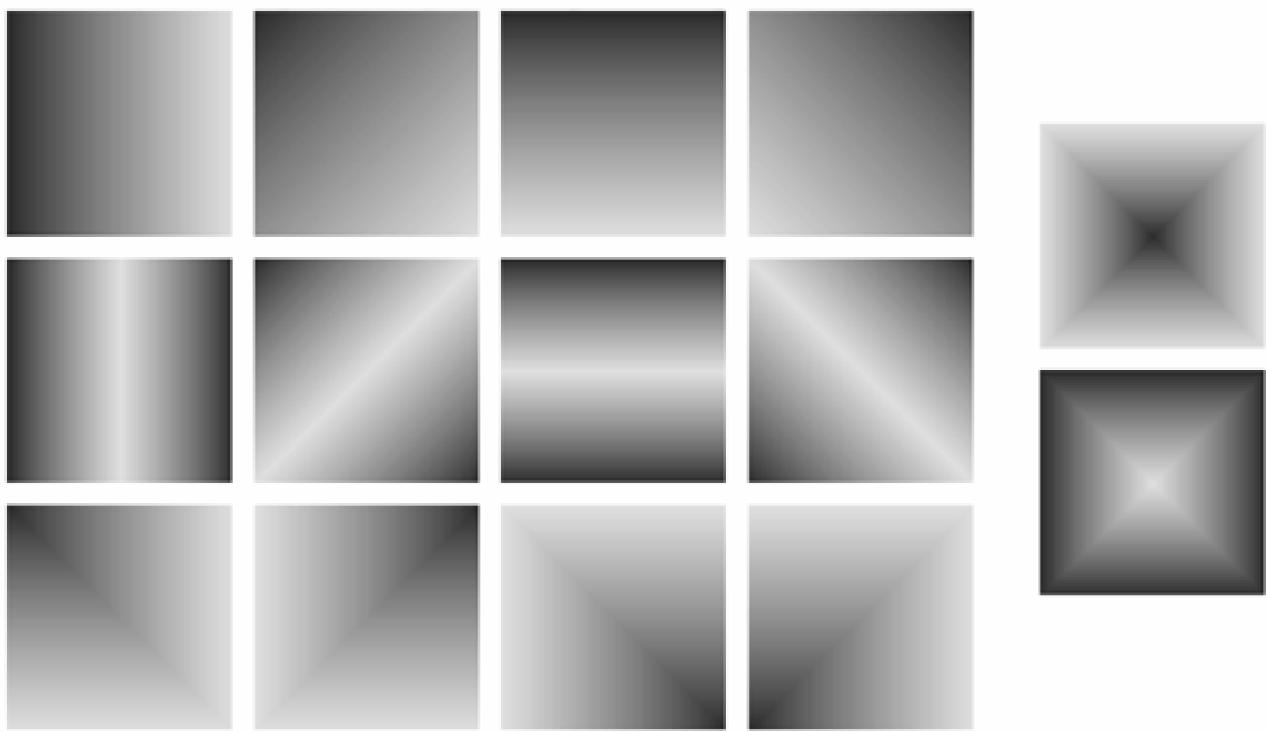
$$C(x,y) = (C(x_0, y_0) * a_0 + C(x_1, y_1) * a_1 + C(x_2, y_2) * a_2) / (a_0 + a_1 + a_2)$$

The color interpolation formulas for rectangle fills are based on distance, so naturally the color interpolation formula for triangle fills is based on area. For a rectangle fill, color forms a straight line in the 2D space formed by either of the axes and each of the color channels. For a triangle fill, color forms a plane in the 3D space formed by the x , y -axes and each of the color channels.

Gradient Fill of Rectangles

As a sample usage of the GradientFill function, let's try to implement a variety of gradient fills for a single rectangle region. How many can you think of? [Figure 9-19](#) shows 14 commonly seen combinations.

Figure 9-19. Gradient fills of a rectangle area.



All these gradient fills use a light color and a dark color to fill a rectangle. The first four fills in a row go in a single direction, either left or right, top to down, or diagonal. The next four fills divide the rectangle into two parts, and fill from two opposites to the center. The next four fill from one corner to the whole rectangle. The last two on the right side fill from the outside to the center. [Listing 9-3](#) shows part of the implementation code.

[Listing 9-3](#) Gradient fill of rectangle areas.

```
inline COLOR16 R16(COLORREF c) { return GetRValue(c)<<8; }
inline COLOR16 G16(COLORREF c) { return GetGValue(c)<<8; }
inline COLOR16 B16(COLORREF c) { return GetBValue(c)<<8; }

inline COLOR16 R16(COLORREF c0, COLORREF c1)
{ return ((GetRValue(c0)+GetRValue(c1))/2)<<8; }
inline COLOR16 G16(COLORREF c0, COLORREF c1)
{ return ((GetGValue(c0)+GetGValue(c1))/2)<<8; }
inline COLOR16 B16(COLORREF c0, COLORREF c1)
{ return ((GetBValue(c0)+GetBValue(c1))/2)<<8; }

BOOL GradientRectangle(HDC hDC, int x0, int y0,
    int x1, int y1, COLORREF c0, COLORREF c1, int angle)
{
    TRIVERTEX vert[4] = {
        { x0, y0, R16(c0), G16(c0), B16(c0), 0 },
        { x1, y1, R16(c1), G16(c1), B16(c1), 0 },
        { x0, y1, R16(c0, c1), G16(c0, c1), B16(c0, c1), 0 },
        { x1, y0, R16(c0, c1), G16(c0, c1), B16(c0, c1), 0 }
    };
    ULONG Index[] = { 0, 1, 2, 0, 1, 3};
    switch ( angle % 180 )
    {
```

```
case 0:
    return GradientFill(hDC, vert, 2, Index, 1,
        GRADIENT_FILL_RECT_H);

case 45:
    return GradientFill(hDC, vert, 4, Index, 2,
        GRADIENT_FILL_TRIANGLE);

case 90:
    return GradientFill(hDC, vert, 2, Index, 1,
        GRADIENT_FILL_RECT_V);

case 135:
    vert[0].x = x1;
    vert[3].x = x0;
    vert[1].x = x0;
    vert[2].x = x1;
    return GradientFill(hDC, vert, 4, Index, 2,
        GRADIENT_FILL_TRIANGLE);
}

return FALSE;
}

BOOL CornerGradientRectangle(HDC hDC, int x0, int y0, int x1,
    int y1, COLORREF c0, COLORREF c1, int corner)
{
    TRIVERTEX vert[] = {
        { x0, y0, R16(c1), G16(c1), B16(c1), 0 },
        { x1, y0, R16(c1), G16(c1), B16(c1), 0 },
        { x1, y1, R16(c1), G16(c1), B16(c1), 0 },
        { x0, y1, R16(c1), G16(c1), B16(c1), 0 }
    };

    vert[corner].Red = R16(c0);
    vert[corner].Green = G16(c0);
    vert[corner].Blue = B16(c0);

    ULONG Index[] = { corner, (corner+1)%4, (corner+2)%4,
        corner, (corner+3)%4, (corner+2)%4 };

    return GradientFill(hDC, vert, 4, Index, 2,
        GRADIENT_FILL_TRIANGLE);
}
```

The GradientRectangle routine handles the four fills in the first row, of which the first and third are simple single rectangle fills. The rest are implemented using two triangle fills. The CornerGradientRectangle routine handles the four fills in the third row; all of them use two triangles. Several inline functions are used here to convert the 8-bit RGB into the 16-RGB needed by the TRIVERTEX structure, and calculate colors between two colors. Note that we

are not using the GRADIENT_RECTANGLE and GRADIENT_TRIANGLE structures here; instead, arrays of unsigned long indexes are used directly. Another thing to notice: Although constant points are used in on-line documentation, the GDI header file stills refuses constant points for the GradientFill function.

Gradient Fills to Make 3D Buttons

Gradient fills can be used multiple times to create interesting effects. If combined with clipping, gradient fills can be applied to nonrectangular areas. One sample usage is making 3D buttons. [Figure 9-20](#) shows three 3D buttons made using the Gradient Rectangle routine.

[Figure 9-20. Using gradient fills to make 3D buttons.](#)



The first rectangle button is made from a gradient fill from dark to light color, followed by another gradient fill in a smaller area from the light color to the dark color. The gradient gives an impression of curved surfaces. The next two buttons just add clipping to rounded rectangles and ellipses. Actually, all three buttons are clipped to rounded rectangle regions with different roundness. Here is the button maker.

```
// Make 3D buttons: w: axis for round corners,  
// d: distance between two rings  
void RoundRectButton(HDC hDC, int x0, int y0, int x1, int y1,  
                     int w, int d, COLORREF c1, COLORREF c0)  
{  
    for (int i=0; i<2; i++)  
  
    {  
        POINT P[3] = { x0+d*i, y0+d*i, x1-d*i, y1-d*i,  
                      x0+d*i+w, y0+d*i+w };  
  
        LPtoDP(hDC, P, 3);  
  
        HRGN hRgn = CreateRoundRectRgn(P[0].x, P[0].y,  
                                         P[1].x, P[1].y, P[2].x-P[0].x, P[2].y-P[0].y);  
        SelectClipRgn(hDC, hRgn);  
        DeleteObject(hRgn);  
  
        if ( i==0 )  
            GradientRectangle(hDC, x0, y0, x1, y1, c1, c0, 45);
```

```
    else
        GradientRectangle(hDC, x0+d, y0+d, x1-d, y1-d,
                          c0, c1, 45);
    }

    SelectClipRgn(hDC, NULL);
}
```

The routine uses a loop to do two gradient fills, each with different dimensions. The clipping part is a little bit tricky because the routine has to find the location and size in the device coordinate space to create the right clipping region. The benefit of this extra effort is that the routine can be used in any logical coordinate space.

[< BACK](#) [NEXT >](#)

9.8 AREA FILLS IN REALITY

Area fill is an important feature of any graphics application, from a word processor and image editor to a sophisticated graphics design package. GDI provides three basic tools to do area fills: brushes and gradient that define the fill color and pattern, area fill functions that fill simple geometric shapes directly, and clipping that restricts the fill region to allow more control over the shapes.

Area fill features provided by graphics applications are quite close to what GDI is offering:

Single-color solid fill, with possible semitransparency.

- Gradient fill
- Texture fill
- Pattern fill
- Bitmap fill

Semitransparent Fill

Single-color solid fill can easily be implemented using GDI's solid brush. Semitransparent area fill fills every other pixel with a certain color, leaving the rest of the destination pixels unchanged. It can be achieved using the chessboard pattern brush, with two raster operations. The following routine shows how to draw a semitransparent rectangle:

```
void SemiFillRect(HDC hDC, int left, int top, int right,
                  int bottom, COLORREF color)
{
    int nSave = SaveDC(hDC);

    const unsigned short ChessBoard[] = { 0xAA, 0x55, 0xAA, 0x55,
                                         0xAA, 0x55, 0xAA, 0x55 };

    HBITMAP hBitmap = CreateBitmap(8, 8, 1, 1, ChessBoard);
    HBRUSH hBrush = CreatePatternBrush(hBitmap);
    DeleteObject(hBitmap);
    HGDIOBJ hOldBrush = SelectObject(hDC, hBrush);
    HGDIOBJ hOldPen = SelectObject(hDC, GetStockObject(NULL_PEN));

    SetROP2(hDC, R2_MASKPEN);
    SetBkColor(hDC, RGB(0xFF, 0xFF, 0xFF)); // keep
    SetTextColor(hDC, RGB(0, 0, 0)); // black
    Rectangle(hDC, left, top, right, bottom);

    SetROP2(hDC, R2_MERGEPEN);
    SetBkColor(hDC, RGB(0x0, 0x0, 0x0)); // keep
```

```
SetTextColor(hDC, color);           // color
Rectangle(hDC, left, top, right, bottom);

SelectObject(hDC, hOldBrush);
SelectObject(hDC, hOldPen);
DeleteObject(hBrush);

RestoreDC(hDC, nSave);
}
```

In the SemiFillRect routine, a pattern brush is created with the chessboard pattern. The first call to the Rectangle function using the R2_MASKPEN raster operation clears the foreground pixels to black (0), and leaves the background pixels untouched. The second call to the Rectangle function using R2_MERGEPEPEN colors the foreground pixels to the specified color, still leaving the background pixels unharmed. With the ternary raster operation we are going to discuss in the next chapter, this can be reduced to a single drawing call with the same brush.

Portable Gradient Fill in HLS Color Space

For Windows 98 and Windows 2000, GDI provides quite good support for gradient fills. But there are still several problems. It has been reported that the gradient fill implemented on Windows 98 has a resource leak, so it's not good for long-time use. GDI support for gradient fill is still not available on Windows NT 4.0 and Windows 95 machines. The only type of gradient fill supported is linear interpolation in RGB space. There is still a need for applications to implement their own gradient fills for all these reasons. Gradient fill for triangles is better implemented using bitmap techniques, but rectangle gradient fill can easily be simulated using multiple rectangle fills using gradually changing colors. The following routine, HLSGradientRectangle, shows how to do a rectangle gradient fill in HLS (hue lightness saturation) space. It does not rely on GDI's GradientFill function.

```
void HLSGradientRectangle(HDC hDC, int x0, int y0, int x1, int y1,
    COLORREF cref0, COLORREF cref1, int nPart)
{
    KColor c0(cref0); c0.ToHLS();
    KColor c1(cref1); c1.ToHLS();

    for (int i=0; i<nPart; i++)
    {
        KColor c;
        c.hue    = (c0.hue*(nPart-1-i) + c1.hue*i) / (nPart-1);
        c.lightness = (c0.lightness*(nPart-1-i) + c1.lightness*i) /
            (nPart-1);
        c.saturation= (c0.saturation*(nPart-1-i) + c1.saturation*i)/
            (nPart-1);
        c.ToRGB();

        HBRUSH hBrush = CreateSolidBrush(c.GetColorRef());

        RECT rect = {x0+i*(x1-x0)/nPart,y0,x0+(i+1)*(x1-x0)/nPart,y1};
        FillRect(hDC, & rect, hBrush);
        DeleteObject(hBrush);
    }
}
```

```
}
```

The routine uses the KColor class to translate colors between RGB and HLS space. Color interpolation is done in HLS space instead of RGB space. A rectangle is equally divided into a number of partitions and filled with solid color brush. In HLS color space, it's very easy to gradually adjust a color's lightness without changing its hue and saturation or to shift a color's hue without changing its lightness and saturation. Both of them are unnatural to express in RGB color space.

Radial Gradient Fill

A radial gradient fill gradually changes color according to the distance, normally from the center of a circle. It can be used to simulate light effects on a 3D-sphere surface. Although GDI does not support radial gradient fills, it can be implemented by dividing a circle into multiple triangles and gradually filling color from a center point to the perimeter of an approximating polygon. Here is a routine that does that.

```
BOOL RadialGradientFill(HDC hDC, int x0, int y0, int x1, int y1,
    int r, COLORREF c0, COLORREF c1, int nPart)
{
    const double PI2 = 3.1415927 * 2;

    TRIVERTEX * pVertex = new TRIVERTEX[nPart+1];
    ULONG     * pMesh   = new ULONG[(nPart+1)*3];

    pVertex[0].x    = x1;
    pVertex[0].y    = y1;
    pVertex[0].Red  = R16(c0);
    pVertex[0].Green = G16(c0);
    pVertex[0].Blue  = B16(c0);
    pVertex[0].Alpha = 0;

    for (int i=0; i<nPart; i++)
    {
        pVertex[i+1].x = x0 + (int) (r * cos(PI2 * i / nPart));
        pVertex[i+1].y = y0 + (int) (r * sin(PI2 * i / nPart));
        pVertex[i+1].Red  = R16(c1);
        pVertex[i+1].Green = G16(c1);
        pVertex[i+1].Blue  = B16(c1);
        pVertex[i+1].Alpha = 0;

        pMesh[i*3+0] = 0;
        pMesh[i*3+1] = i+1;
        pMesh[i*3+2] = (i+1) % nPart+1;
    }

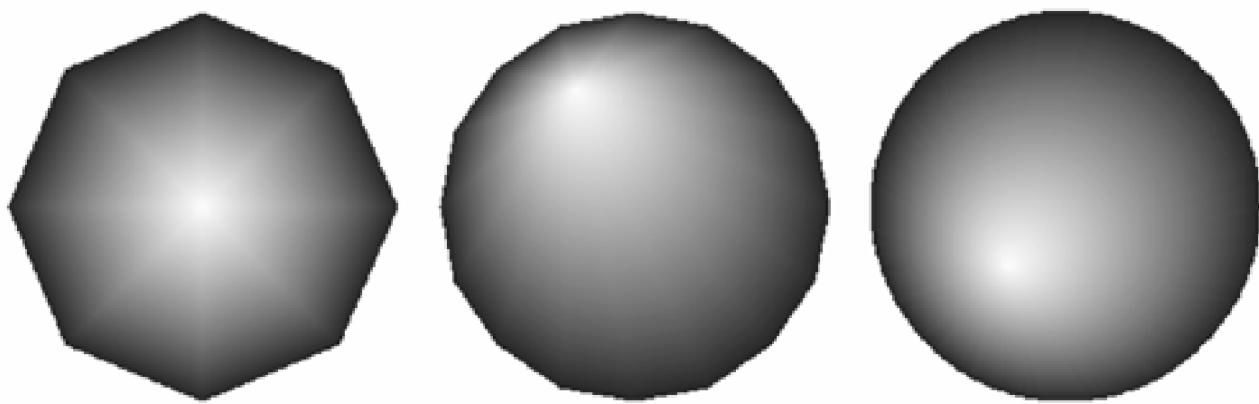
    BOOL rsIt = GradientFill(hDC, pVertex, nPart+1, pMesh,
        nPart, GRADIENT_FILL_TRIANGLE);
```

```
delete [] pVertex;
delete [] pMesh;

return rslt;
}
```

The routine approximates a circle by a polygon, and divides it into a number of triangles to be filled by GradientFill. Point (x_0, y_0) is the real center of the circle, while point (x_1, y_1) is the starting point of all gradient triangles to simulate different viewing points. [Figure 9-21](#) shows examples using 8-, 16-, and 256-sided polygon partitions.

[Figure 9-21. Radial gradient fill using multiple triangles.](#)



This routine is good enough to make a few buttons with curved surfaces, but it definitely does not match the professional 3D quality that would be generated by Direct 3D or OpenGL. The routine uses uniform color on the vertices, and we probably still do not have enough triangles.

Texture and Bitmap Fills

Texture fill means filling an area with bitmaps representing the texture of certain materials—for example, paper, marble, granite, sand, or wood. GDI's pattern brush would have been a good tool to implement texture fills, but it has two problems. First, pattern brushes are limited to 8-by-8 pixels on non-NT-based systems. Second, the GDI pattern dimension is in the device coordinate space, which does not scale with device resolution. Using 8-by-8 bitmaps is only good enough for very fine grain textures for screen display. A texture bitmap good for 96-dpi screen display would be too small for 1200-dpi printers. For example, the wood texture pattern is quite sensitive to device resolution.

Bitmap fills just means stretching a bitmap to fill an area. Both texture and bitmap fills involve bitmaps, which will be covered in depth in the next chapter, so we will leave this topic until later.

Pattern Fills

GDI provides several predefined hatch patterns to fill areas with dual color patterns with possible transparency support. The hatch patterns were originally designed for screen display. Although the NT graphics DDI allows printer drivers to provide scaled-up bitmaps for hatch brush implementation, not many device drivers actually use this customization feature. If an application uses hatch brushes for screen display, the hatch patterns will not scale between 72-, 96-, or 120-dpi display modes. They will not scale when the display scale is changed. If an application uses hatch brushes for a printing job, the hatch patterns would be visible only under a microscope.

Pattern fills are quite easy to implement using GDI's line drawing features, which can have scalable pen widths and

coordinates expressed in the logical coordinate space. Shown below is a simple routine that does rotated brick pattern fill using lines.

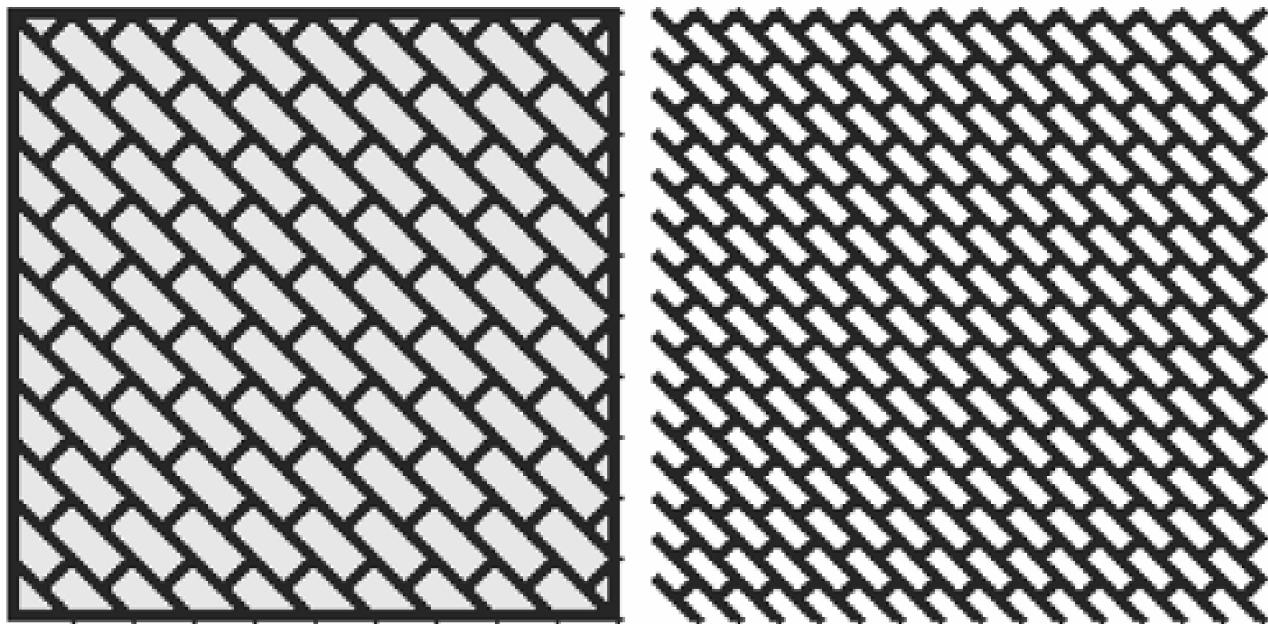
```
void BrickPatternFill(HDC hDC, int x0, int y0, int x1, int y1,
    int width, int height)
{
    width = abs(width);
    height = abs(height);

    if ( x0>x1 ) { int t = x0; x0 = x1; x1 = t; }
    if ( y0>y1 ) { int t = y0; y0 = y1; y1 = t; }

    for (int y=y0; y<y1; y += height )
        for (int x=x0; x<x1; x += width )
    {
        MoveToEx(hDC, x,      y, NULL);
        LineTo(hDC, x+width,  y+height);
        MoveToEx(hDC, x+width, y, NULL);
        LineTo(hDC, x+width/2, y+height/2);
    }
}
```

The routine uses the logical coordinate space only, so the pattern can be transformed easily. It does not handle clipping to the defining rectangle, opaque background filling, and brush origin alignment. [Figure 9-22](#) shows a sample output.

Figure 9-22. Using lines to implement pattern fills.



9.9 SUMMARY

In this chapter we covered GDI's area filling features—that is, brushes, area filling function, regions, and the latest gradient fills.

Unlike pens that do have their own geometric dimension, brushes specify only how color patterns are used to fill the interior of an area. We went to quite some length to cover the limitation of GDI brushes when faced with the reality of modern application requirements and operating-system incompatibility issues, and we offered some ideas, solutions, and suggestions.

Brushes as created by GDI are logical specifications for the actual brushes that are used by the device drivers to do the drawing, which are normally device-dependent. When a new logical brush is first used, the graphics engine calls the device driver to realize a logical brush, or in other words, to generate a physical brush data structure based on the logical brush. After that, a realized brush object is passed to the device driver entry points for those functions that use the brush. [Chapter 3, Section 3.7](#), of this book covers more details about internal brush data structure, together with other GDI objects.

GDI provides quite a few functions to paint simple geometric shapes like rectangles, rounded rectangles, ellipses, and polygons with brushes for interior and pens for outline. More complicated shapes can be defined using paths that combine different types of curves. On the DDI level, almost all outlines are converted to paths, and internal implementation for `StrokeAndFillPath` is used to implement most of the area filling functions. Even polygons and polypolygons are paths with straight lines only. One exception is a rectangle under compatible graphics mode, which uses a simpler DDI entry point.

GDI is a basic graphic programming interface, so not enough geometric manipulation features are provided. When shapes get complicated, calculating their exact outlines becomes much harder, if not impossible. The easy way out, using GDI, is regions. Regions allow a quite complete suite of set-like operations that combine regions to make new regions, which can be used for either painting or clipping. Regions do have their cost in memory usage and CPU time, and even output quality if zoomed to a larger scale. GDI provides a pair of APIs to let an application manipulate the region's underlying representation and the application's transformation of it. This opens the door for lots of interesting applications such as adding perspective transformations to region data.

The latest implementation of the Win32 GDI API adds some more color-depth to GDI's 2D API, the gradient fill. The gradient fill provides a way to simulate light effects on different kinds of surface. It will be seen more and more often in applications.

So far, we have covered pixel drawing, line and curve drawing, and area filling. So we have covered zero-, one-, and two-dimensional geometric shapes. Starting from the next chapter, we will cover various kinds of bitmaps supported by GDI and their ever-expanding usage.

Further Reading

Filling an area using a solid color or a brush on a raster-based device is normally referred to as scan conversion in computer graphics literature. There is lots of information on this topic. For example, *Graphics Gems*, Volume I, edited by Andrew S. Glassner, contains an article on fast scan conversion of arbitrary polygons and antialiasing polygon scan conversion.

The new GradientFill function uses rectangles and triangles to define basic geometric shapes and uses the TRIVERTEX structure to define the attributes of a vertex. You will find that the design of GradientFill looks like OpenGL or other game programming API like Direct3D. The *OpenGL Programming Guide*, written by the Open GL Architecture Review Board, is a good reference on OpenGL.

Sample Program

There is only one big sample program for this chapter, “[Areas](#)” (see [Table 9-3](#)). It demonstrates all the topics covered in this chapter and provides all the figures in this chapter.

Table 9-3. Sample Program for Chapter 9

Directory	Description
Samples\Chapt_09\Area	The Test menu contains dozens of options to demonstrate color dithering, hatch brushes, pattern brushes, system color brushes, rectangle, frame control drawing, ellipse, chord, pie, rounded rectangle, polygon, poly polygon, region, path, and gradient fill.

[< BACK](#) [NEXT >](#)

Chapter 10. Bitmap Basics

Pixels, lines, and areas, as we discussed in the last three chapters, can be used to synthesize a financial chart, an engineering drawing, a textile pattern design, or a cartoon scene. Synthetic graphics programming follows precise mathematical formulas to use geometrical objects to build up a picture, similar to a graphic artist who uses brushes and paints to draw a picture on a canvas. This branch of computer graphics is often referred to as *vector graphics*, or *line art*. Another important branch of computer graphics deals with digitized images taken from the real world, which is called *bitmap graphics*.

An image is a rectangular array of picture elements, or pixels, each representing a color. Images are also referred to as *sampled images*, because they are often the results of finite digital sampling of a real-world scene, using a scanner, digital camera, or a video camera.

Windows operating systems choose to use the term *bitmap* instead of *image*, which is a word made from concatenating *bit* and *map*. This may hint at the black/ white original of Windows graphics programming, but the term *bitmap* is actually used to refer to generic images.

The topic of bitmaps is too big for a single chapter, so we divide it into three chapters. This chapter is going to discuss image data format, displaying images onto a graphics device. We will cover three different bitmap formats supported by GDI: DIB (device-independent bitmap), DIB section, and DDB (device-dependent bitmap). In the next chapters, we will discuss how to display bitmaps transparently, blend alpha with background, fade bitmap to darkness, lighten up bitmap, rotate bitmap, do image processing, etc.

10.1 DEVICE-INDEPENDENT BITMAP FORMATS

When capturing an image from real-life sources, image data needs to be converted to a digitized format, which can then be stored in computer memory or secondary storage, or transmitted to a remote device. The formats in which images are stored in computers today are quite messy. Different operating systems, hardware vendors, and even applications store images in different formats. The commonly seen bitmap formats include:

- JPEG, developed by Joint Photographic Experts Group, a 24-bit color image format with lossy compression.
- TIFF, developed by Aldus, a very versatile image format supporting many sub-formats, both MAC and PC binary order, and LZW compression.
- GIF, developed by CompuServe, an image format designed for small images with no more than 256 colors, supporting progressive display and transparency.
- PNG (www.cdrom.com/pub/png/), Portable Network Graphics, an emerging image format that has a long list of fancy features, and as yet nonpatented algorithms and free source code.

The image format native to Microsoft Windows operating systems is the BMP image format. Compared with other image formats, the BMP is a very simple image format mainly designed for easy graphics programming within an application, or among several applications. Its color depth support is quite adequate, with support for 1-, 2-, 4-, 8-bit indexed color images, and 16-, 24-, and 32-bit true color RGB images.

Images in BMP format tend to be very big in size, because the format only supports the simplest form of run-length encoding in its 4-and 8-bit indexed color formats. For example, a 1024-by-768, 24-bit true color BMP image is 2.25 MB, which can normally be compressed to about 200 KB as a JPEG image. It's not a good idea to save such images on hard disk or transform them through the Internet.

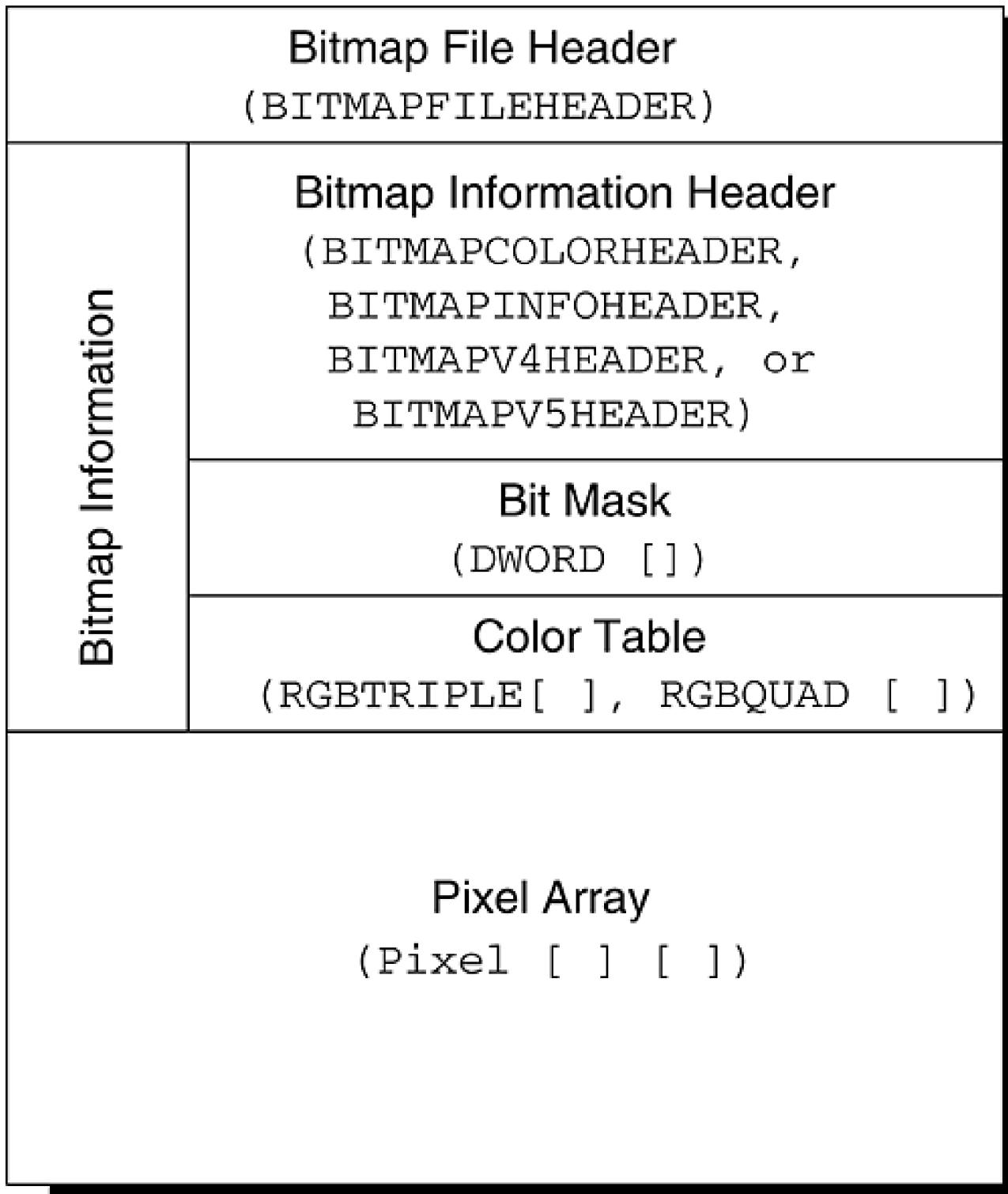
BMP File Format

A bitmap in the BMP format is commonly known as a device-independent bitmap (DIB). The phrase "device-independent" here means the format contains complete information about a bitmap, such that it can be reproduced on various devices. It is not supposed to mean that the bitmap is encoded in a device-independent color space, although the newer versions of Microsoft operating systems are adding color profiling data in the BMP format to compensate for color device dependency. Device-independent bitmaps contrast with another format of images the Windows graphics system uses internally during runtime, the device-dependent bitmaps, or DDBs. In this section we will first discuss DIBs, the fundamental Windows image format, then cover DDB and yet another image format, the DIB section, later.

When stored in a disk file, a device-independent bitmap, or a DIB, contains three pieces of information: a bitmap file header, a bitmap information block, and an array of pixels. The bitmap information block can be further divided into a bitmap information header, an array of bit masks, and a color table, depending on the color depth of the image.

[Figure 10-1](#) illustrates the on-disk format of a DIB image.

Figure 10-1. BMP file format.



Bitmap File Header

BMP file's bitmap file header is a simple file header that enables applications to identify a BMP file. It stores three pieces of basic information: a BMP file signature, file length field, and an offset to the pixel array. It is defined in the BITMAPFILEHEADER structure.

```
typedef struct tagBITMAPFILEHEADER {  
    WORD bfType;      // BMP file signature  
    DWORD bfSize;     // size of whole file  
    WORD bfReserved1; // 0  
    WORD bfReserved2; // 0  
    DWORD bfOffBits;  // offset from file start to pixel array  
} BITMAPFILEHEADER;
```

The signature field bfType is formed by two ASCII characters, “B” and “M”, for BMP file, so it must be 0x4D42, or “M” * 256 + “B”. The size field is the total size length of the image file, a nice feature for downloading. The last field of the structure is an offset to the pixel array from the starting of the image file.

Note that the BITMAPFILEHEADER structure was initially designed for 16-bit Windows, so it's only WORD aligned, not DWORD aligned. The size of it is 14 bytes long, which makes the bitmap information header after that not DWORD aligned either. This will cause trouble when you try to use a memory-mapped bitmap file to back up a DIB section.

Bitmap Information Header

The bitmap file header just tells an application that a file is a BMP file. The exact image information is kept in the bitmap information block that follows. The first part of the bitmap information block is a bitmap information header.

While the bitmap file header has survived several generations of Windows operating systems without a single change, the bitmap information header has changed a lot and is still changing. A bitmap information header contains the bitmap image format, dimension, compression scheme, color table size, color profiling information, etc. To date, there are four versions. The easiest version is the BITMAPCOREHEADER structure, originally designed for the OS/2 operating system.

```
typedef struct tagBITMAPCOREHEADER {  
    DWORD bcSize;      // sizeof(BITMAPCOREHEADER)  
    WORD bcWidth;     // pixel width of bitmap  
    WORD bcHeight;    // pixel height of bitmap + orientation  
    WORD bcPlanes;    // number of planes, must be 1  
    WORD bcBitCount;  // number of bits per pixel  
} BITMAPCOREHEADER;
```

The most widely used bitmap information header is the BITMAPINFOHEADER structure, a much-expanded version from the OS/2 version:

```
typedef struct tagBITMAPINFOHEADER {  
    DWORD biSize;      // sizeof(BITMAPINFOHEADER)  
    LONG biWidth;     // pixel width  
    LONG biHeight;    // pixel height + orientation  
    WORD biPlanes;    // number of planes, must be 1
```

```
WORD biBitCount; // number of bits per pixel
DWORD biCompression; // compression scheme
DWORD biSizeImage; // size of pixel array
LONG biXPelsPerMeter; // horizontal resolution
LONG biYPelsPerMeter; // vertical resolution
DWORD biClrUsed; // number of colors in the color table
DWORD biClrImportant; // number of entries needed to display
} BITMAPINFOHEADER;
```

The BITMAPINFOHEADER structure is normally referred to as version 3 of the bitmap information structure. Anything in the Win32 API that dates back to Windows 3.1 is referred to as version 3, new features added by Windows 95 and Windows NT are referred to as version 4, and newer features added by Windows 98 and Windows 2000 are referred to as version 5.

Windows 95 and Windows NT 4.0 add a new bitmap information header structure BITMAPV4HEADER; Windows 98 and Windows 2000 add another BITMAP V5HEADER structure. The top parts of these two new structures match the BITMAP INFOHEADER structure field by field, except that the value of the size field should be sizeof(BITMAPV4HEADER) or sizeof(BITMAPV5HEADER). The version 4 bit map information header structure adds RGBA color masks, color space, color end points, and gamma, designed for ICM 1.0 support. The version 5 bitmap information header structure adds new color space types, rendering intent, and color profile data, designed for ICM 2.0 support. Refer to MSDN Library for detailed descriptions of these structures.

Ironically, when the Win98 imaging component generates BMP files using the V5 information header, it's considered a defect, not a feature, because even Visual Basic can't read the new BMP files. But a well-behaved application should at least expect BMP files in all four bitmap information headers, even if it ignores the new fields of the V4 and V5 headers and treats them as the BITMAPINFOHEADER structure.

In the bitmap information header structures, the first field is the size of the structure, which is the only indicator of which version of the header is actually used. If it equals sizeof(BITMAPCOREHEADER), then an application has to use the OS/2 DIB format. The size field also serves as an offset to find the DIB color table.

The next two fields are the width and height of the DIB in pixels. Note that the OS/2 DIB format stores them in 16-bit WORD, while the later versions store them as 32-bit LONG. The height of a DIB is a positive value normally, but it could be negative. The sign indicates the order of the scan lines in the pixel array. Positive DIB height corresponds to a bottom-up order of scan lines, in which the first pixel in the pixel array is the first pixel of the last scan line in the picture. Negative DIB height signals that the scan lines are in a more intuitive, top-down order. Most BMP files are in the traditional bottom-up order in terms of scan lines.

The plane and bit-count fields determine the format of scan lines in the pixel array. For a two-color image, black/white images being the special case, one bit is needed to represent a pixel. For a 256-color image, 8 bits are needed to represent a pixel. Different graphics devices can store a scan line differently, using either a single plane or multiple planes. The DIB format only supports single-plane images, so the plane field must be one. The bit-count field now solely determines the storage size of each pixel and the number of colors represented by a pixel. [Table 10-1](#) outlines supported bit count.

Table 10-1. Supported DIB Bit Count: Bits per Pixel

Value	Maximum Colors	Pixel Size	Description
0	Depends on embedded image		Windows 98/2000 only; embedded JPEG or PNG image.

1	$2 (2^1)$? byte	Monochrome image.
2	$4 (2^2)$	$\frac{1}{4}$ byte	Four-color image used on WinCE.
4	$16 (2^4)$	$\frac{1}{2}$ byte	16-color image.
8	$256 (2^8)$	1 byte	256-color image.
16	$32,768 (2^{15})$, or $65,536 (2^{16})$	2 bytes	High-color image.
24	$166,777,216 (2^{24})$	3 bytes	True color image.
32	$166,777,216 (2^{24})$	4 bytes	True color image.

The image with bit count “ n ” is commonly denoted in ‘ n -bpp’ format, meaning n bits per pixel. When bit count is 8 or less, a color table is attached after the bitmap information header structure.

The BITMAPCOREHEADER structure used for OS/2 ends after the bit-count field, while the other structures have more information to specify. The OS/2 DIB format should be interpreted to use the default values for the unspecified fields.

The biCompression field holds the compression method used in the pixel array. [Table 10-2](#) summarizes the supported values.

The most common DIB compression method is no compression when the biCompression field is BI_RGB. Visual C++ Studio and Microsoft-provided image editors only generate BMP files in uncompressed format. When uncompressed, each scan line in a DIB is a packed array of pixels. For each pixel, a 1-bpp pixel occupies ? of a byte, a 2-bpp pixel occupies $\frac{1}{4}$ of a byte, and a 4-bpp pixel occupies one nibble or $\frac{1}{2}$ of a byte. For these three cases, one byte can hold several pixels, starting from the most significant bits to the least significant. For images with higher bit counts, each pixel is stored in biBitCount/8 bytes. Each scan line in a device-independent bitmap is always rounded to the closest double-word boundary, padded with zero if needed.

Table 10-2. Supported DIB Compression Method

Value	Description
BI_RGB	Uncompressed image.
BI_RLE8	8-bpp image using run-length encoding compression. Bottom-up DIB only.
BI_RLE4	4-bpp image using run-length encoding compression. Bottom-up DIB only.
BI_BITFIELDS	16-, 32-bpp uncompressed image, three-bit masks provided to specify how to extract RGB components from a pixel.
BI_JPEG	Pixel array is an embedded image in JPEG format. Windows 98/2000 only.
BI_PNG	Pixel array is an embedded image in PNG format. Windows 98/2000 only.

For 4-bpp and 8-bpp DIBs, an optional run-length encoding can be used to reduce the size of a bitmap. Run-length encoding for 8-bpp images looks at a number of continuous bytes having the same value, replacing them with two bytes: a repeat count and byte to repeat. Special escape groups are designed to deal with cases where bytes are not the same, end of line, or end of image. In the best case, run-length encoding is able to reduce image size by a factor of 128 if each scan line has the same pixel value. In the worst case, run-length encoding could generate an image larger than the original uncompressed image.

Here is the format of an image compressed using the BI_RLE8 compression method, using BNF (Backus-Normal-Form) notation:

```
<image> ::= <pixelrun> { <pixelrun> }
<pixelrun> ::= <encodedrun> | <rawrun>
<encodedrun> ::= <endofline> | <endofimage> | <delta> |
    <repeatrun>
<endofline> ::= 0 0
<endofimage> ::= 0 1
<delta> ::= 0 2 dx dy
<rawrun> ::= 0 raw_count { byte }
<repeatrun> ::= repeat_count byte_to_repeat
```

An RLE compressed image is encoded as a sequence of pixel runs, which has five forms. If the first byte of a pixel run is not zero, it's a repeat count from 1 to 255, which tells how many times the next byte should be repeated. If the first byte is zero, the next byte could either be 0 for end of line, 1 for end of image, 2 for delta move, or 3 to 255 for raw uncompressed pixels. The delta run moves the current cursor within the decoded image by relative values on both the x-and y-axes. It permits quick jumping over multiple scan lines. The end-of-scan line and end-of-image pixel run also allow cut scan line or image short, but the values of these skipped-over pixels are not defined. So in practical usage, the delta run is not normally encountered; end-of-scan line and end-of-image are added after the last pixel of the scan line or whole image. In other words, each scan line is normally encoded separately and no pixels are skipped to avoid ambiguity. The raw pixel run takes care of nonrepetitive pixel sequences. It starts with 0 and a raw byte count between 3 and 255, because 0 to 2 are already used for end of line, end of image, and delta. The exact number of bytes follows the count. Each raw run must contain an even number of bytes, so the raw byte count must be an even number. This ensures that each pixel run within an encoded image is always on WORD boundary, making encoding and decoding easier and more efficient.

If an RLE-encoded image does not skip pixels, its encoding syntax can be simplified as:

```
<image> ::= <scanline> { <scanline> } [ <endofimage> ]
<scanline> ::= <pixelrun> { <pixelrun> } [ <endofline> ]
<pixelrun> ::= <repeatrun> | <rawrun>
```

For 4-bpp image using the 4-bit BI_RLE4 compression, the basic syntax for the image is the same as for BI_RLE8 compression. But now each pixel is 4 bits only, or a nibble. The raw count is a count of pixels, so the data following it should be $(\text{raw_count}+1)/2$ bytes. The repeat count is also a pixel count; the byte after that contains two pixels that will be used alternatively to fill the required number of pixels.

For 16-bit and 32-bit DIBs, the biCompression field could be BI_BITFIELDS. This is not a real compression method; it's a way to tell the order and size of red, green, and blue components in a pixel. When the biCompression field is BI_FIELDS, three DWORDs are added after the bitmap information header and before the color table. They are the masks to take red, green, and blue components out from 16-or 32-bit packed pixels. So it sounds like these masks allow great flexibility in creating strange DIB formats. Actually there are lots of restrictions. NT-based systems restrict the masks for each channel to be contiguous and non-overlapping. But it's possible to create a DIB with an unusual RGB channel order on NT-based systems. Non-NT-based systems only support three variations of bit masks. For 16-bpp images, only 5-5-5 RGB and 5-6-5 RGB formats are supported. In the 5-5-5 RGB format, the blue mask is 0x1F, the green mask is 0x3E0, and the red mask is 0x7C00, so each channel is 5 bits. In the 5-6-5

RGB format, the blue mask is 0x1F, the green mask is 0x7E0, and the red mask is 0xF800; only the green channel is 6-bit. For 32-bpp images, only the 8-8-8 format is supported, where the blue mask is 0xFF, the green mask is 0xFF00, and the red mask is 0xFF0000.

The bit masks are really designed to solve the compatibility issue caused by the different implementation of high color display modes. The *PC 99 System Design Guide* specifies that a display card should support a 5-5-5 or 5-6-5 16-bit frame buffer, or both. If only 5-5-5 is supported, it should be reported as 16 bit instead of 15 bit, in case certain applications get confused. For a 32-bit frame buffer, an 8-8-8-8 frame buffer is the requirement, where the MSB 8-bit is the alpha channel. Strangely, the alpha channel is not mentioned in the bitmap information header documentation.

The BI_JPEG and BI_PNG compression types may raise some hope that GDI is finally addressing the issue of huge uncompressed high color or true color DIB images. But the solution provided is half-hearted. The two compression modes are valid under Windows 98 and Windows 2000 only under limited conditions. GDI and the graphics engine underneath do not provide any decoding for JPEG and PNG images. Video display drivers do not support them. Only printer drivers can optionally support them. An application should query a printer device context using the ExtEscape function to check for JPEG or PNG support; only after getting an affirmative reply can an application send a JPEG or PNG compressed bitmap wrapped as a DIB to the device driver. In this case, GDI simply passes the data to a printer driver. This only provides a partial solution to the huge memory footprint that is required by large-size high color or true color DIB images. We will discuss BI-JPEG in [Chapter 17](#).

Now let's return back to the bitmap information header structures. After the compression flag comes the biSizeImage field, which holds the size of the image pixel array. For BI_RGB compression, the biSizeImage field can be 0; GDI can calculate image size based on width, height, and bit count. But if RLE, JPEG, or PNG compression is used, it must be the actual size of the image data.

The last two fields of a BITMAPINFOHEADER structure hold information about the color table. The biClrUsed field stores the number of entries in the color table. For DIBs with no more than 256 colors, a zero for biClrUsed field means the maximum number of entries, or $2^{(\text{bit count})}$. The on-disk format of a DIB must contain a full-size color table with $2^{(\text{bit count})}$ entries. Only one in-RAM DIB format can have an incomplete color table. The biClrImportant field specifies the number of entries that are actually required to display the bitmap. Again zero means all are important instead of none.

Each entry in the color table is usually 4 bytes unless it's in the old OS/2 format, which adds up to 1024 bytes for an 8-bpp DIB. That's clearly something to optimize in the Win16 world, which has a 64 KB GDI heap. For Win32 programming, 1024 bytes are not considered a big deal unless you have hundreds or thousands of images. The biClrImportant field is really a hint to the application program as to how many colors are actually used in the image. It can be used to generate a perfect-size palette to display the image in a palette-based frame buffer.

A high color or true color DIB image does not need a color table, because each pixel has full RGB components. But it could have nonzero biClrUsed and biClrImportant fields, and a color table. A color table for high color or true color images can be used to generate a palette for displaying the image on a palette-based device. [Chapter 13](#) of this book covers palette.

Bit Masks

For a 16-or 32-bpp DIB image, if its biCompression field is BI_BITFIELDS, the bit masks are stored after the bitmap information header as an array of DWORDs. There are always three masks in the RGB order.

GDI does not provide any data structure or API to access the bit masks.

Color Table

For DIB images with no more than 256 colors, each pixel in the pixel array is an index into a color table that translates the indexes into RGB values. High color or true color images can also have color tables for creating logical palette on palette-based systems.

The number of entries in the color table is specified in the bitmap information header's biClrUsed field. For the OS/2 DIB format, or when it's zero, the maximum number of entries is assumed.

The entries in the color table have three forms. For the OS/2 DIB format, each entry is represented using a RGBTRIPLE structure. For other DIB formats, each entry is represented using a RGBQUAD structure. For an in-memory DIB, each entry could be a 16-bit WORD that is yet another index. Both RGBTRIPLE and RGB QUAD specify color using 8-bit RGB values. They differ only by a reserved field, as shown below.

```
typedef struct tagRGBTRIPLE {
```

```
    BYTE rgtBlue;  
    BYTE rgtGreen;  
    BYTE rgtRed;  
} RGBTRIPLE;
```

```
typedef struct tagRGBQUAD {
```

```
    BYTE rgbBlue;  
    BYTE rgbGreen;  
    BYTE rgbRed;  
    BYTE rgbReserved;  
} RGBQUAD;
```

GDI defines two more data structures that combine bitmap information header and the color table, BITMAPCOREINFO and BITMAPINFO:

```
typedef struct tagCOREINFO {
```

```
    BITMAPCOREHEADER bmciHeader;  
    RGBTRIPLE      bmciColors[1];  
} BITMAPCOREINFO;
```

```
typedef struct tagBITMAPINFO {
```

```
    BITMAPINFOHEADER bmiHeader;  
    RGBQUAD       bmiColors[1];  
} BITMAPINFO;
```

Consider these data structures dangerous and take care in using them. Both structures only have space for one color table entry. So an application needs to allocate more space than their size to hold the bitmap information header and color table. The bmiHeader field in the BITMAPINFO structure could either be BITMAP INFO HEADER, or BITMAPV4HEADER, or BITMAPV5HEADER, so the offset of the bmiColors field is not reliable. Even if you're only using a BITMAPINFOHEADER structure, no space is reserved for bit masks for 16-bpp and 32-bpp DIBs. An

application should not rely on the BITMAPINFO structure to locate the color table of a bitmap; instead, it should calculate the offset to the color table based on the bitmap information header during runtime.

Pixel Array

The pixel array is where the real pixels of an image are stored. An image is normally a sequence of scan lines rounded to the closest 32-bit boundary. The order of the scan lines is bottom-up by default, unless the biHeight field in the bitmap information header structure is negative. Bottom-up means the first pixel in the pixel array is really the first pixel of the last scan line when displayed on the screen in MM_TEXT mode.

Within each scan line, pixels are packed together to save space. Extra bits are added at the end of a scan line to make it a multiple of double words. One important value to calculate for a DIB is the number of bytes per scan line. It can be calculated using the following function:

```
int inline ScanlineSize(int width, int bitcount)
{
    return (width * bitcount + 31)/32;
}
```

For DIBs in BI_RGB compression mode, accessing individual pixels with a pixel array is quite easy and can be implemented quite efficiently. Direct pixel access is very important to implement imaging algorithms on bitmaps, or to enhance GDI's drawing capability. It's also a critical skill in DirectDraw programming, where a drawing surface is basically a DIB. We will cover the details later.

Packed Device-Independent Bitmap

The BMP file format is the format in which a DIB is stored in an independent disk file. We mentioned that a BMP file contains a file header, a bitmap information block, and a pixel array. The file header serves an information purpose during loading of a DIB into memory. But once in memory, the file header is no longer needed. The file header can be reconstructed from the bitmap information header.

A DIB without a file header is called a packed DIB. The word *packed* here does not mean the way in which pixels are packed in a scan line. It conveys that the remaining pieces of a DIB are next to each other in a continuous memory.

A packed DIB starts with a bitmap information header, followed by a mask array, color table, and pixel array. Win32 API commonly use a pointer to BITMAPINFO structure to denote a packed DIB pointer. Although the structure has no reference to the mask array and pixel array, at least it tells you there is a color table.

Quite a few Win32 API return and accept packed DIBs. When DIBs are included in executable files as resources, FindResource, LoadResource, and LockResource can be used to return a pointer to a packed DIB. CreateDIBPatternBrushPt uses a packed DIB to create a pattern brush. The Window clipboard also uses packed DIBs.

Within an in-memory packed DIB, the color table could be a table of indexes to a logical palette. For example, if the iUsage parameter to CreateDIBPatternBrushPt is DIB_PAL_COLORS, the color table is an array of indexes. But such a DIB should not be passed to other applications or written to disk, unless you're only dealing with willing partners. There is no flag in the DIB file format to tell that the color table is relative to an unknown palette.

Divided Device-Independent Bitmap

The information a packed DIB carries can be divided into two groups: pixel array and bitmap format information. The latter consists of the bitmap information header, masks, and color table. It's a burden to carry them along together. For example, an image editor may support multiple levels of undoing by saving multiple intermediate DIBs; all have the exactly the same information except the pixel arrays. An application may accept images in other formats like PCX, TIFF, or GIF. It may want to construct a BIT MAPINFO structure in memory, a pixel array from the decompressed data, and then hope to pass these two pieces as a DIB.

Quite a few GDI drawing calls are flexible enough to drop the requirement for a packed DIB. Instead a DIB is passed to them in two parameters, a pointer to the BIT MAPINFO structure, and a pointer to the pixel array. This gives great flexibility in constructing a DIB in RAM.

A divided device-independent bitmap, or a nonpacked DIB, can have an incomplete color table to save space. For example, a 64-grayscale image in DIB format only needs to allocate space for 64 entries in its color table instead of 256.

[< BACK](#) [NEXT >](#)

10.2 A DIB CLASS

Although the BMP image format is a simple image format, the description given in the last section does not look so simple. The complexity of a BMP file is the result of continually evolving the format to cater to new needs, and the constant struggling between performance and memory usage.

Graphics programming with device-independent bitmaps is not that easy. The GDI API does not provide enough support to make that job easier. For example, there is no function to return the number of colors in the color table for a packed DIB, or return the pointer to the pixel array in a packed DIB. Programmers have to write their own code to parse different versions of the bitmap information structures to figure them out. More complicated problems like calculating the address of a pixel (x, y) in the pixel array or converting a color image to a grayscale image are also not supported.

A C++ class can best capture the DIB processing. But even Microsoft Foundation Class shies away from providing a DIB class. Several sources have provided their own implementation of DIB class. In this section, we are starting to build a nontrivial DIB class. Here are the design goals.

- Be able to read and display all valid DIB formats. Input images to an application could come from various sources, so being able to accept them and display them is the minimum requirement.
- Be able to access various DIB information efficiently. Most of the DIB information is kept in the bitmap information header, which has four flavors, with different default values. A good DIB class needs to access that information in the shortest time without going through lots of conditional statements.
- Provide access to individual pixels in the uncompressed pixel array. Direct DIB pixel accessing is key to implementing lots of image-processing algorithms.

The basic idea is to provide a DIB class that can read and display all valid DIB formatted images and also provide efficient imaging algorithms on uncompressed true color images. [Listing 10-1](#) shows the class declaration for the DIB class, KDIB.

Listing 10-1 KDIB Class Declaration

```
typedef enum
{
    DIB_1BPP,      // 2 color image, palette-based
    DIB_2BPP,      // 4 color image, palette-based
    DIB_4BPP,      // 16 color image, palette-based
```

```
DIB_4BPPRLE,      // 16 color image, palette-based, RLE compressed
DIB_8BPP,        // 256 color image, palette-based
DIB_8BPPRLE,      // 256 color image, palette-based, RLE compressed

DIB_16RGB555,    // 15 bit RGB color image, 5-5-5
DIB_16RGB565,    // 16 bit RGB color image, 5-6-5, 1 bit unused
DIB_24RGB888,    // 24 bit RGB color image, 8-8-8
DIB_32RGB888,    // 32 bit RGB color image, 8-8-8, 8 bit unused

DIB_32RGBA8888, // 32 bit RGBA color image, 8-8-8-8

DIB_16RGBbitfields, // 16 bit RGB color image, nonstandard bit masks,
// NT-only
DIB_32RGBbitfields, // 32 bit RGB color image, nonstandard bit masks,
// NT-only

DIB_JPEG,        // embedded JPEG image
DIB_PNG          // embedded PNG image
} DIBFormat;

typedef enum
{
    DIB_BMI_NEEDFREE = 1,
    DIB_BMI_READONLY = 2,
    DIB_BITS_NEEDFREE = 4,
    DIB_BITS_READONLY = 8
};

class KDIB
{
public:
    DIBFormat m_nImageFormat; // pixel array format
    int m_Flags; // DIB_BMI_NEEDFREE, ....
    BITMAPINFO * m_pBMI; // BITMAPINFOHEADER + mask + color table
    BYTE * m_pBits; // pixel array

    RGBTRIPLE * m_pRGBTRIPLE; // OS/2 DIB color table within m_pBMI
    RGBQUAD * m_pRGBQUAD; // V3,4,5 DIB color table within m_pBMI
    int m_nClrUsed; // No. of colors in color table
    int m_nClrlmpt; // Real color used
    DWORD * m_pBitFields; // 16, 32-bpp masks within m_pBMI

    int m_nWidth; // image pixel width
    int m_nHeight; // image pixel height, positive
    int m_nPlanes; // plane count
```

```
int      m_nBitCount;    // bit per plane
int      m_nColorDepth;  // bit count * plane count
int      m_nImageSize;   // pixel array size

                                // precalculated values
int      m_nBPS;          // byte per scan line, per plane
BYTE *   m_pOrigin;       // point to logical origin
int      m_nDelta;        // delta to next scan line

KDIB();                  // default constructor, empty image
virtual ~KDIB();         // virtual destructor

BOOL Create(int width, int height, int bitcount);

bool AttachDIB(BITMAPINFO * pDIB, BYTE * pBits, int flags);
bool LoadFile(const TCHAR * pFileName);
bool LoadBitmap(HMODULE hModlue, LPCTSTR pBitmapName);
void ReleaseDIB(void);    // release memory
int  GetWidth(void) const { return m_nWidth; }
int  GetHeight(void) const { return m_nHeight; }
int  GetDepth(void) const { return m_nColorDepth; }
BITMAPINFO * GetBMI(void) const { return m_pBMI; }
BYTE * GetBits(void) const { return m_pBits; }
int  GetBPS(void) const { return m_nBPS; }

bool IsCompressed(void) const
{
    return (m_nImageFormat == DIB_4BPPRLE) ||
           (m_nImageFormat == DIB_8BPPRLE) ||
           (m_nImageFormat == DIB_JPEG) ||
           (m_nImageFormat == DIB_PNG);
}

};
```

The first group of four data members in the KDIB class are the most important and basic information about a bitmap. All other data members can be derived from these members.

A DIB can have lots of different raster formats, which are determined by the bit count, compression flag, and even the bit masks. It will be much easier for image algorithms if there is a single value for the raster format. That's the reason behind defining the DIBFormat enumeration type. Now we see clearly that there are 15 different types of raster formats supported by the BMP format. The Windows NT/2000 DDK uses a similar method to name raster formats. For example, constants like BMF_4RLE, BMF_8BPP, or BMF_32BPP are passed to EngCreateBitmap to create a GDI-managed surface.

The KDIB class has many more data fields than the BITMAPINFOHEADER structure. Remember that we don't care about saving a dozen bytes here; performance is considered more important. Instances of the KDIB class will be stored in computer memory, so they really represent an in-memory DIB; therefore, the BITMAPFILEHEADER structure is not needed. The m_pBMI member is a pointer to its bitmap information block. The m_pBits field holds a pointer to the pixel array of the bitmap. With separate pointers to bitmap information block and pixel array, the KDIB class can support both packed DIB and nonpacked DIB.

The source of the bitmap could be from several sources. It could be loaded from a file, loaded from a resource, pasted from the clipboard, or even programmatically generated. The bitmap class needs to know whether the two pointers are read-only, or whether any of them needs to be deleted from memory when an instance of the KDIB class is being deleted. The m_Flags flag is designed to meet these needs. The m_Flags field is a combination of four flags.

- DIB_BMI_NEEDFREE: m_pBMI points to a heap allocation which needs to be freed in the destructor.
- DIB_BMI_READONLY: m_pBMI points to read-only data.
- DIB_BITS_NEEDFREE: m_pBits points to a separate heap allocation which needs to be freed in the destructor.
- DIB_BITS_READONLY: m_pBits points to read-only data.

The second group of five data members holds pointers to color tables either in RGB TRIPLE or RGBQUAD form, color count, and important color count. We are not supporting the palette-relative index color table here, which is not part of the DIB file format. Only one of m_pRGBTRIPLE and m_pRGBQUAD could be a non-NULL pointer. The m_pBitsFields member points to bit field masks if used. This first portion of the KDIB class gives direct pointers to the bitmap information header, color table, and bit mask.

The third group of data members spells out the details of the image format. We have image width, image height (always positive), plane count, bit count, color depth, and image size. Note in the bitmap information header that bitmap height could be negative to denote a top-down bitmap. A negative height is hard to deal with in image algorithms. So the image height is normalized to be positive in KDIB class, while the top-down orientation information is reflected in the next group of member variables.

The fourth group of data members frequently uses values that are precalculated based on the abovementioned members. Field m_nBPS is bytes per scan line, rounded to the closest DWORD. Field m_pOrigin points to pixel (0, 0) in the pixel array, which is the first byte in the pixel array for a top-down DIB. Field m_nDelta is the difference between a scan line and its next scan line. For top-down DIB, m_nDelta is positive, otherwise it's negative. These three values can be used to quickly calculate the address of a scan line, from which pixel address for an individual pixel can be

figured out. An independent m_nDelta field can also be used to store the pitch of a DirectDraw surface, which may not be the same as m_nBPS.

The KDIB class has a simple default constructor, a virtual destructor to allow for virtual functions, and derived classes. The ReleaseDIB method is responsible for releasing resources allocated for the current image being represented. Several constant functions are defined to return image geometric information.

The AttachDIB routine is the main item responsible for initializing a KDIB class instance, given a packed or a nonpacked DIB as input. The LoadBitmap routine loads a BMP resource from a Win32 module and initializes a read-only bitmap by calling AttachDIB. The LoadFile routine loads a BMP file from disk and initializes a KDIB instance by calling AttachDIB. The three routines are shown in [Listing 10-2](#).

Listing 10-2 Initializing KDIB Class from BMP File or Resource

```
bool KDIB::AttachDIB(BITMAPINFO * pDIB, BYTE * pBits, int flags)
{
    if ( IsBadReadPtr(pDIB, sizeof(BITMAPCOREHEADER)) )
        return false;

    ReleaseDIB();

    m_pBMI    = pDIB;
    m_Flags   = flags;

    DWORD size = * (DWORD *) pDIB; // always DWORD size

    int compression;
    // gather information from bitmap information header structures
    switch ( size )
    {
        case sizeof(BITMAPCOREHEADER):
        {
            BITMAPCOREHEADER * pHeader = (BITMAPCOREHEADER *) pDIB;

            m_nWidth   = pHeader->bcWidth;
            m_nHeight  = pHeader->bcHeight;
            m_nPlanes  = pHeader->bcPlanes;
            m_nBitCount = pHeader->bcBitCount;
            m_nImageSize= 0;
            compression = BI_RGB;

            if ( m_nBitCount <= 8 )
            {

```

```
m_nClrUsed = 1 << m_nBitCount;
m_nClrImpt = m_nClrUsed;
m_pRGBTRIPLE = (RGBTRIPLE *) ((BYTE *) m_pBMI + size);
m_pBits = (BYTE *) & m_pRGBTRIPLE[m_nClrUsed];
}

else
    m_pBits = (BYTE *) m_pBMI + size;
break;
}

case sizeof(BITMAPINFOHEADER):
case sizeof(BITMAPV4HEADER):
case sizeof(BITMAPV5HEADER):
{
    BITMAPINFOHEADER * pHeader = & m_pBMI->bmiHeader;

    m_nWidth = pHeader->biWidth;
    m_nHeight = pHeader->biHeight;
    m_nPlanes = pHeader->biPlanes;
    m_nBitCount = pHeader->biBitCount;
    m_nImageSize= pHeader->biSizeImage;
    compression = pHeader->biCompression;

    m_nClrUsed = pHeader->biClrUsed;
    m_nClrImpt = pHeader->biClrImportant;

    if ( m_nBitCount<=8 )
        if ( m_nClrUsed==0 ) // 0 means full color table
            m_nClrUsed = 1 << m_nBitCount;

    if ( m_nClrUsed ) // has a color table
    {
        if ( m_nClrImpt==0 ) // 0 means all important
            m_nClrImpt = m_nClrUsed;

        if ( compression==BI_BITFIELDS )
        {
            m_pBitFields = (DWORD *) ((BYTE *)pDIB+size);
            m_pRGBQUAD = (RGBQUAD *) ((BYTE *)pDIB+size +
                3*sizeof(DWORD));
        }
        else
            m_pRGBQUAD = (RGBQUAD *) ((BYTE *)pDIB+size);

        m_pBits = (BYTE *) & m_pRGBQUAD[m_nClrUsed];
    }
}
```

```
        }

    else
    {
        if ( compression==BI_BITFIELDS )
        {
            m_pBitFields = (DWORD *) ((BYTE *)pDIB+size);
            m_pBits     = (BYTE *) m_pBMI + size +
                           3 * sizeof(DWORD);
        }
        else
            m_pBits     = (BYTE *) m_pBMI + size;
    }
    break;
}

default:
    return false;
}

if ( pBits )
    m_pBits = pBits;

// precalculate information DIB parameters
m_nColorDepth = m_nPlanes * m_nBitCount;
m_nBPS       = (m_nWidth * m_nBitCount + 31) / 32 * 4;

if (m_nHeight < 0 ) // top-down bitmap
{
    m_nHeight = - m_nHeight; // change to positive
    m_nDelta  = m_nBPS;      // forward
    m_pOrigin = m_pBits;      // scan0 .. scanN-1
}
else
{
    m_nDelta  = - m_nBPS;    // backward
    m_pOrigin = m_pBits + (m_nHeight-1) * m_nBPS * m_nPlanes;
                           // scanN-1..scan0
}

if ( m_nImageSize==0 )
    m_nImageSize = m_nBPS * m_nPlanes * m_nHeight;

// convert compression mode to image format
switch ( m_nBitCount )
{
```

```
case 0:  
    if ( compression==BI_JPEG )  
        m_nImageFormat = DIB_JPEG;  
    else if ( compression==BI_PNG )  
        m_nImageFormat = DIB_PNG;  
    else  
        return false;  
  
case 1:  
    m_nImageFormat = DIB_1BPP;  
    break;  
  
case 2:  
    m_nImageFormat = DIB_2BPP;  
    break;  
  
case 4:  
    if ( compression==BI_RLE4 )  
        m_nImageFormat = DIB_4BPPRLE;  
    else  
        m_nImageFormat = DIB_4BPP;  
    break;  
  
case 8:  
    if ( compression==BI_RLE8 )  
        m_nImageFormat = DIB_8BPPRLE;  
    else  
        m_nImageFormat = DIB_8BPP;  
    break;  
  
case 16:  
    if ( compression==BI_BITFIELDS )  
        m_nImageFormat = DIB_16RGBbitfields;  
    else  
        m_nImageFormat = DIB_16RGB555; // see below  
    break;  
  
case 24:  
    m_nImageFormat = DIB_24RGB888;  
    break;  
  
case 32:  
    if ( compression == BI_BITFIELDS )  
        m_nImageFormat = DIB_32RGBbitfields;  
    else
```

```
m_nImageFormat = DIB_32RGB888; // see below
break;

default:
    return false;
}

// try to understand bit fields
if ( compression==BI_BITFIELDS )
{
    DWORD red  = m_pBitFields[0];
    DWORD green = m_pBitFields[1];
    DWORD blue  = m_pBitFields[2];

    if ( (blue==0x001F) && (green==0x03E0) && (red==0x7C00) )
        m_nImageFormat = DIB_16RGB555;
    else if ( (blue==0x001F) && (green==0x07E0) && (red==0xF800) )

        m_nImageFormat = DIB_16RGB565;
    else if ( (blue==0x0OFF) && (green==0xFF00) && (red==0xFF0000) )
        m_nImageFormat = DIB_32RGB888;
}

return true;
}

bool KDIB::LoadBitmap(HMODULE hModule, LPCTSTR pBitmapName)
{
    HRSRC hRes = FindResource(hModule, pBitmapName, RT_BITMAP);
    if ( hRes==NULL )
        return false;

    HGLOBAL hGlb = LoadResource(hModule, hRes);

    if ( hGlb==NULL )
        return false;

    BITMAPINFO * pDIB = (BITMAPINFO *) LockResource(hGlb);

    if ( pDIB==NULL )
        return false;

    return AttachDIB(pDIB, NULL, DIB_BMI_READONLY | DIB_BITS_READONLY);
}
bool KDIB::LoadFile(const TCHAR * pFileName)
```

```
{  
    if ( pFileName==NULL )  
        return false;  
  
    HANDLE handle = CreateFile(pFileName, GENERIC_READ, FILE_SHARE_READ,  
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);  
  
    if ( handle == INVALID_HANDLE_VALUE )  
        return false;  
  
    BITMAPFILEHEADER bmFH;  
  
    DWORD dwRead = 0;  
    ReadFile(handle, & bmFH, sizeof(bmFH), & dwRead, NULL);  
  
    if ( (bmFH.bfType == 0x4D42) &&  
        (bmFH.bfSize<=GetFileSize(handle, NULL)) )  
    {  
        BITMAPINFO * pDIB = (BITMAPINFO *) new BYTE[bmFH.bfSize];  
  
        if ( pDIB )  
        {  
            ReadFile(handle, pDIB, bmFH.bfSize, & dwRead, NULL);  
            CloseHandle(handle);  
  
            return AttachDIB(pDIB, NULL, DIB_BMI_NEEDFREE);  
        }  
    }  
    CloseHandle(handle);  
  
    return false;  
}
```

The LoadBitmap method accepts a module handle and a bitmap resource name. It uses FindResource to locate the resource, LoadResource to retrieve a handle to the resource, and LockResource to get a pointer to a packed DIB. Note that in the Win32 environment, the whole process of FindResource, LoadResource, and LockResource does not really allocate any resource for the resource, except that the corresponding pages on the disk are swapped into memory. The value returned by LockResource is a pointer to the memory-mapped image of the module where the resource is embedded. So the data pointed to is read-only and does not need to be freed after usage. LoadBitmap calls AttachDIB to initialize an instance of the KDIB class with a packed read-only bitmap. The flags passed to AttachDIB are DIB_BMI_READONLY | DIB_BITS_READONLY. No heap deallocation is needed when the KDIB class instance is deleted.

The LoadFile routine opens a file, reads a BITMAPFILEHEADER structure, and checks the BMP file

signature and file size. If everything looks OK, a memory block for a packed DIB is allocated and the rest of the file is read into the memory block. The pointer to the memory block is passed to AttachDIB for more checking and initialization of KDIB class data members. The flag passed to AttachDIB is DIB_BMI_NEEDFREE, so the allocated data block will be freed in the destructor.

The AttachDIB routine uses everything we described about the DIB format in the last section to initialize dozens of member variables. The first part of the code caters to four different versions of the bitmap information header. Image format decoding follows this. The last part of the routine tries to check if an image in the BI_BITFIELD compression mode fits the three groups of known bit masks supported on Windows 95/98. It handles both packed DIB and nonpacked DIB. For a nonpacked DIB, two separate pointers are passed to AttachDIB.

With this still simple DIB class, we can load any BMP files and start to have some fun with images.

[< BACK](#) [NEXT >](#)

10.3 DISPLAYING A DIB

The Win32 GDI provides two functions to display a device-independent bitmap in a device context:

```
int StretchDIBits(HDC hdc, int XDest, int YDest, int nDestWidth,  
    int nDestHeight, int XSrc, int YSrc, int nSrcWidth,  
    int nSrcHeight, CONST VOID * lpBits, CONST BITMAPINFO *  
    lpBitsInfo, UINT iUsage, DWORD dwRop);  
  
int SetDIBitsToDevice(HDC hDC, int XDest, int YDest,  
    DWORD dwWidth, DWORD dwHeight, int XSrc, int YSrc,  
    UINT uStartScan, UINT cScanLines, CONST VOID * lpvBits,  
    CONST BITMAPINFO * lp bmi, UINT fuColorUse);
```

StretchDIBits

The StretchDIBits function is an extremely important GDI function, so let's start with it. The first parameter is apparently a handle to a device context. The next four parameters, XDest, YDest, nDestWidth, and nDestHeight, define a rectangle on the destination surface to draw on, using logical coordinates. Following them is another quadruple, XSrc, YSrc, nSrcWidth, and nSrcHeight. The parameters which define a rectangle in the DIB bitmap. The lpBits parameter points to the pixel array of the DIB, while the lpBitsInfo parameter points to one of the BITMAPINFO structure variations. Together they define a DIB that may or may not be packed. The iUsage parameter is normally DIB_RGB_COLORS if an RGB-based color table is used, or DIB_PAL_COLORS if the color table contains logical palette indexes. The last parameter is a raster operation flag, which is a whole other story. For the moment, we just use the simplest raster operation, SRCCOPY.

What StretchDIBits does is take a subimage of the DIB image, clip to the current clip region, resize the subimage to be destination rectangle size, convert it to the same color format as the destination surface, and overwrite the data in destination surface with the result, assuming raster operation is SRCCOPY. Conceptually, the process consists of six steps: source selection, transformation, clipping, resizing, color format conversion, and raster operation.

Source Rectangle

The first selection step is quite straightforward. It's specified by the source rectangle parameters. If you want to display a full image, these four parameters should be 0, 0, image width, and image height. Note that for a top-down DIB, the biHeight field in the BITMAPINFOHEADER structure is negative. Its absolute value should be used. To GDI the source rectangle is actually defined by [XSrc, YSrc, XSrc+nSrcWidth, YSrc+nDstWidth], using the right, bottom exclusive rule. If an application uses image width, image height, -image width, -image height as parameters to Stretch DI Bits, the rectangle will be [image width, image height, 0, 0]. So is this the same rectangle as [0, 0, image width, image height]? GDI treats this as a coordinate system mapping; the whole image will be displayed but flipped both vertically and horizontally. The source rectangle parameter can be used to select a portion of the image. If part of the rectangle is off the image, that part is considered clipped. [Table 10-3](#) provides several possible combinations of these four parameters.

Note the source image coordinates are interpreted as logical coordinates of the source image, not physical coordinates. A vertical coordinate of 0 means the first logical scan line of the image, not the first physical scan line of the image. For a top-down bitmap, the first logical scan line is the first physical scan line, while for a bottom-up bitmap, the first logical scan line is the last physical scan line in the pixel array.

Table 10-3. StretchDibts XSrc, YSrc, nSrcWidth, nSrcHeight Parameters

Value	Source Rectangle
0, 0, width, height	Whole image, original orientation.
width, 0, -width, height	Whole image, flip horizontal.
0, height, width, -height	Whole image, flip vertical.
width, height, -width, -height	Whole image, flip both horizontal and vertical.
0, 0, width, 1	First scan line of the image.
0, 0, 1, height	First column of the image.

Destination Rectangle and Stretch Modes

By the same token, the destination rectangle is determined by mapping [XDst, YDst, XDst+nDestWidth, YDst+nDestHeight] from logical coordinate space to device coordinate space, under the right, bottom exclusive rule. If the left and right, bottom and right coordinates of the rectangle are not ordered, flipping occurs again. So the actual orientation of the image depends on both the source and destination rectangles.

The selected source image needs to be scaled to fit the dimensions of the destination rectangle. There could be three cases for the resizing: no scaling, up scaling, or down scaling. No scaling is simple; one pixel in the source image corresponds to one pixel in the destination surface, no more, no less. Up scaling needs to map single pixels into multiple destination pixels, which may not be an integer ratio. Down scaling needs to map multiple pixels into a single pixel; again, it may not have an integer ratio. There are lots of ways scaling can be handled, and there is no obvious winner. The way in which image scaling can be done is controlled by the stretch mode attribute in GDI. GDI lets the application control each device context's stretch mode through a pair of functions.

```
int SetStretchBltMode(HDC hDC, int iStretchMode);
int GetStretchBltMode(HDC hDC);
```

Function SetStretchBltMode sets the stretch mode flag in a device context, and GetStretchBltMode retrieves it. The allowed values are listed in [Table 10-4](#).

The stretch mode only deals with down scaling where a bigger image is displayed into a smaller destination rectangle. GDI implements up scaling using simple pixel duplication. If an application needs smooth edges for up-scaled images, implement your own scaling algorithm. Both STRETCH_ANDSCANS and STRETCH_ORSCANS are designed for black/white images. If you want to preserve black lines on a white background—that is, make sure thin black lines do not disappear or break during scaling—STRETCH_ANDSCANS should be used, because its logical-AND operation favors black (0) over white (1). If you want to preserve white lines on a black background, STRETCH_ORSCANS should be used, because it chooses white (0) over black (1). On color images, these two modes can generate strange images, so the next two modes should be used. STRETCH_DELETESCAN simply ignores extra data. It would be fast, but may not yield good results, because extra pixels are ignored when

downscaled, resulting in loss of information. STRETCH_HALFTONE takes an average of color pixels to get a better approximation of the human eye's perception of a smaller object. But it's definitely slower. Another problem is that STRETCH_HALF TONE is only implemented on NT-based systems.

Table 10-4. Image Stretch Modes

Stretch Mode (Old Name)	Meaning
STRETCH_ANDSCANS (BLACKONWHITE)	Combine multiple pixels using bitwise logical AND. Preserve black data in RGB mode.
STRETCH_ORSCANS (WHITEONBLACK)	Combine multiple pixels using bit-wise logical OR. Preserve black data in RGB mode.
STRETCH_DELETESCAN (COLORONCOLOR)	Preserve single pixels, delete the rest.
STRETCH_HALFTONE (HALFTONE)	Take average of multiple pixels. Only supported on NT-based systems. Call SetBrush OrgEx to realign brush origin after setting this mode.

Color Format Conversion

A color image can be displayed on a black/white printer, and a black/white image can be displayed on a color screen. GDI needs to convert pixels from the source image format to the destination device context format. Again, when source and destination have the same color format, the conversion is only for the storage format.

A DIB image is always a color image. Even a dual-color image needs to have color tables with two entries. So the color table translates pixel indexes into color value. For RGB mode device color, RGB color value can be used directly. To draw to a palette-based device, the logical palette and system palette are involved in mapping RGB values to palette indexes. We will discuss palette management in [Chapter 13](#). To draw a palette-based DIB on an RGB-based device, indexes stored in the bitmap need to be converted to RGB values using the color table associated with the bitmap.

Displaying a color image on a black/white device can be ambiguous. When the STRETCH_HALFTONE stretch mode is used, StretchDIBits halftones color images to black/white; when other stretch modes are used, StretchDIBits uses the closest color matchings. As a comparison, this behavior is different from displaying a device-dependent bitmap on a black/white device context.

Raster Operation

After color format conversion, we have a transformed pixel array, ready to be written to the destination surface. Like the binary raster operation that combines pen/brush color with destination color to define the new destination color, GDI allows raster operations to be performed to determine the final value of the destination pixel.

We will discuss the details of raster operations in the next chapter. For the moment, we stick with the simplest raster operation, SRCCOPY, which just copies the source pixel into the destination.

StretchDIBits Sample

Now let's write some code to illustrate the use of the StretchDIBits functions. First we need to add a function to the KDIB class to display a DIB.

```
int KDIB::DrawDIB(HDC hDC, int dx, int dy, int dw, int dh,
    int sx, int sy, int sw, int sh, DWORD rop)
{
    if ( m_pBMI )
        return ::StretchDIBits(hDC, dx, dy, dw, dh,
            sx, sy, sw, sh, m_pBits,
            m_pBMI, DIB_RGB_COLORS, rop);
    else
        return GDI_ERROR;
}
```

Now we can finally use the KDIB class to load a DIB and display it. The following code loads an image of a lion and displays it in four different orientations.

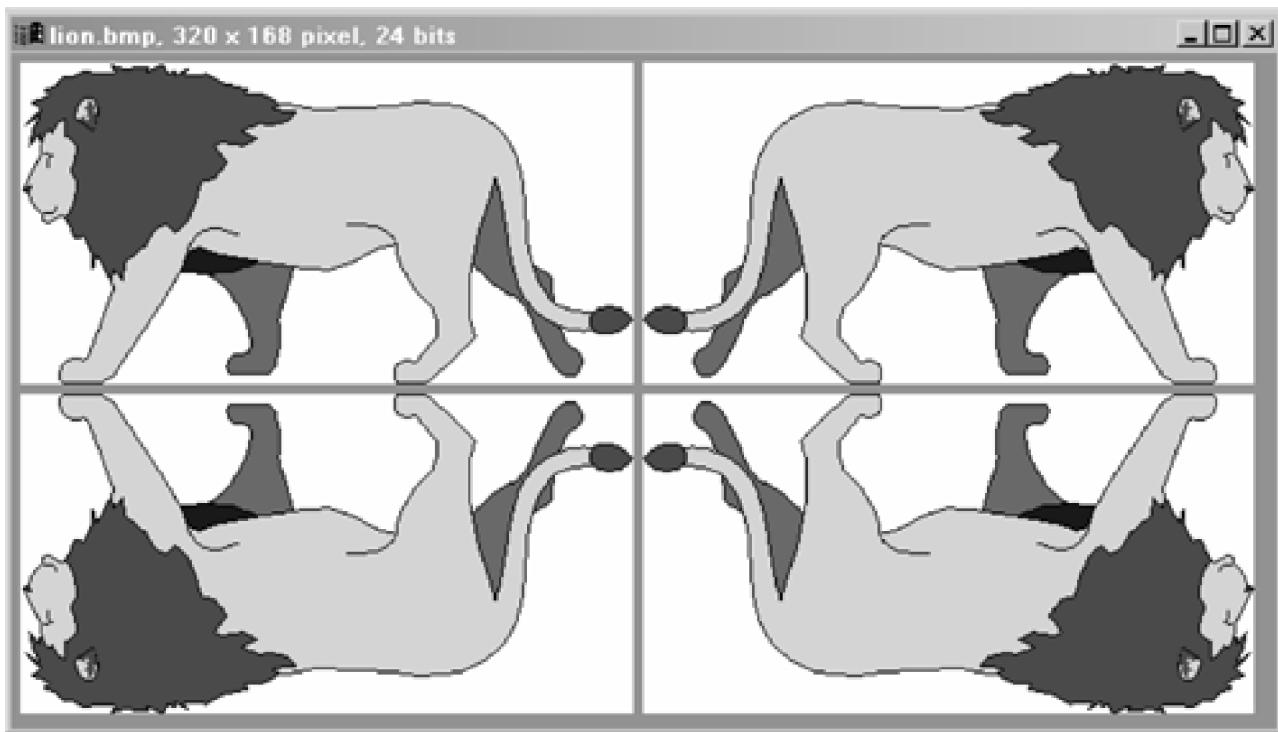
```
KDIB lion;
```

```
if ( lion.LoadFile(_T("lion.bmp")) )
{
    int w = DIB.GetWidth();
    int h = DIB.GetHeight();

    DIB.DrawDIB(hDC, 5, 5, w, h, 0, 0, w, h, SRCCOPY);
    DIB.DrawDIB(hDC, 10+w, 5, w, h, w, 0, -w, h, SRCCOPY);
    DIB.DrawDIB(hDC, 5, 10+h, w, h, 0, h, w, -h, SRCCOPY);
    DIB.DrawDIB(hDC, 10+w, 10+h, w, h, w, h, -w, -h, SRCCOPY);
}
```

The code above is for illustration only. Loading of images should be kept to a minimum, instead of every time a display is needed. The sample program, “[Bitmap](#),” that comes with this chapter allows opening DIB images through the file open dialog box and displaying each of them in the MDI child window. Each child window instance has an instance of the KDIB class, which loads an image once and displays it many times. [Figure 10-2](#) shows one child window.

Figure 10-2. Flipping images using StretchDIBits.



SetDIBitsToDevice

The SetDIBitsToDevice function is a quite a loner in the GDI API. Microsoft should have removed it from the Win32 API. If an application imports it, most likely the importing is compulsory through the MFC CDC class.

The function draws a DIB or part of a DIB in its original orientation and scale, regardless of the current world transformation and window-to-viewport mapping. The only parameters expressed in logical coordinate space are those of the destination position (*xDest*, *yDest*). The parameters *XSrc*, *YSrc*, *dwWidth*, and *dwHeight* define a sub image of a DIB. Do not try to play the same game with signs as we do with Stretch DIBits, because the width and height are passed as unsigned integers. There is no need to pass the size of the destination rectangle; it's always one-to-one in device coordinate space.

The most interesting thing about SetDIBitsToDevice is the way in which a DIB, or part of it, is passed. The *lpbmi* parameter is still the same pointer to the bitmap information header. The *lpvBits* parameter points to a buffer that holds several scan lines of the image, or the whole image. The function is designed for *lpvBits* to point to a partial DIB, instead of a full DIB. Two extra parameters are used to specify where the data in the buffer fits. The *uStartScan* parameter is the sequential scan-line number within the whole image of the first scan line pointed to by *lpvBits*, and *cScanLines* is the number of scan lines stored in the buffer. SetDIBitsToDevice always copies the source pixel into the destination, so there is no need for the raster operation. The last parameter, *fuColorUse*, tells how the DIB color table should be interpreted.

For a single call, SetDIBitsToDevice copies *cScanLines* from the buffer to the destination device surface, starting from (*xDest*, *yDest* + *uStartScan*), subject to source rectangle clipping and destination device context clipping. Note that destination coordinates need to be mapped to device coordinate space. The following call shows how to use SetDIBitsToDevice to display a whole image to location (*x*, *y*).

```
SetDIBitsToDevice(hDC, x, y, m_nWidth, abs(m_nHeight), 0, 0,  
0, abs(m_nHeight), m_pBits,
```

```
(const BITMAPINFO *) m_pDIB, DIB_RGB_COLORS);
```

The only advantage SetDIBitsToDevice offers over StretchDIBits is low memory consumption. If a Win16 application running on a 4-MB RAM laptop needs to display a 1-MB BMP image, it does not have enough RAM memory to load it fully into memory. With SetDIBitsToDevice, it can load the bitmap information header and color table into memory, set up all the parameters, and then repetitively read scan line by scan line into a buffer and call SetDIBitsToDevice each time. The memory buffer needed is a scan line worth of data, or if more memory is available, several scan lines can be read at once.

The StretchDIBits function can also implement the same drip-feeding design rationale of SetDIBitsToDevice. But you need to modify the height field in the bitmap information header to reflect the number of scan lines in the buffer, and adjust the destination and source rectangle for every call.

For Win32 API applications, memory is less of a concern than it is for Win16 applications. If huge image files need to be displayed, an application can use memory-mapped file techniques to map images into virtual memory, which then will be managed automatically by the operating-system memory manager.

For Windows 95/98 machines, displaying a big image using a single Stretch DI Bits call can still cause system performance problems. GDI on those systems is basically 16-bit code. Whenever GDI needs to draw something, it thunks from the 32-bit world into 16-bit GDI and blocks other threads from accessing GDI, because the 16-bit world is not multithread safe. For a single call with huge data, GDI can spend several seconds working on it, while your application can't move the mouse cursor. It's quite common for applications running on Windows 95/98 to divide a big image into small chunks.

SetDIBitsToDevice does not handle stretching itself. So when stretching is needed, an application has to implement its own stretching algorithm. This is a limitation of this function, instead of a feature. When image size is the same as destination rectangle size, StretchDIBits does not need to do stretching, so the application can prescale the data and call StretchDIBits.

If the application is running in MM_TEXT mode, SetDIBitsToDevice can be used to apply certain transformations to an image piece by piece in a small memory buffer. Here is an example of inverting an image's color during display.

```
if ( ! m_DIB.IsCompressed() )
{
    int bps = m_DIB.GetBPS();
    BYTE * buffer = new BYTE[bps];

    for (int i=0; i<m_DIB.GetHeight(); i++)
    {
        memcpy(buffer, m_DIB.GetBits() + bps*i, bps);

        for (int j=0; j<bps; j++)
            buffer[j] = ~ buffer[j];
        SetDIBitsToDevice(hDC, 10, 10, m_DIB.GetWidth(),
                           m_DIB.GetHeight(), 0, 0, i, 1, buffer,
                           m_DIB.GetBMI(), DIB_RGB_COLORS);
    }
    delete [] buffer;
}
```

The code will not run if the image is compressed, because it's quite hard to move from scan line to scan line in a compressed image. For every scan line, it's copied into a scan line size buffer, inverted, and displayed using SetDIBitsToDevice call. Each scan line is processed one by one. You can see that the only change in the parameter is the uStartScan parameter.

[< BACK](#) [NEXT >](#)

10.4 MEMORY DEVICE CONTEXTS

A device context as discussed up to now is always connected with a physical graphics device, a piece of hardware with a dedicated device driver. A device context provides an abstract graphics device to applications using GDI. Underneath the cover, GDI maintains a table of functions to entry points in a graphics device's driver for each device context, which will be called to perform DDI calls. This sounds like C++ virtual functions, or COM methods, and actually works in similar ways.

The power of this graphics device abstraction allows GDI to simulate a graphics device completely in memory, in the same sense as a RAM disk simulates a hard disk. Such an in-memory graphics device is managed by a memory device context.

For historical reasons, a memory device context is not completely independent from a physical graphics device. Instead, a memory device context is always associated with a physical graphics device through one of its device contexts. To be more specific, GDI only provides a single function to create a memory device context that is compatible with an existing device context. Here is the function.

HDC CreateCompatibleDC(HDC hDC);

The reference device context passed to the function must support raster operations; otherwise, the memory device context will not be very useful. A display device context is normally used to create a memory device context, because a video display card normally has complete and correct implementation of raster operations. By contrast, a printer device context may not be a good candidate to be the base to create memory device context. Printers have different degrees of support for color and raster operations. For example, a PostScript printer driver normally has limited support for raster operations. GDI even allows passing a NULL handle as the default value for creating a memory device context compatible with the current screen.

A memory device context is backed up by a bitmap. All the drawing commands are implemented as drawings on that bitmap, instead of on a physical device. A memory device context does not have a fixed bitmap; instead, the bitmap is an attribute of it, which can be selected and unselected just like a pen or a brush object handle. The bitmap attribute of a device context can be accessed using GetCurrentObject, with OBJ_BITMAP as the object type parameter. When a memory device context is first created, GDI sets its bitmap attribute to be a single-pixel monochrome bitmap. So at least you can set and get this pixel's color.

To really make use of a memory device context, a bitmap needs to be created and selected into it. A device-independent bitmap as described in the last section is not supported as such a bitmap. GDI only supports a device-dependent bitmap and a DIB section to serve as a drawing surface for a memory device context. They will be discussed in the next two sections.

Memory device context is indispensable in Windows graphics programming. But we have to leave this section now and continue discussing memory device context's usage after we get a handle on the device-dependent bitmap and DIB section.

[< BACK](#) [NEXT >](#)

10.5 DEVICE-DEPENDENT BITMAPS

Device-independent bitmaps (DIB) provide an easy way to accept images from the outside world, manipulate them programmatically, display them, or share image data with other applications or machines. The main problem DIB images have is that GDI does not support writing to a DIB directly. For that, GDI provides another class of bitmap, device-dependent bitmaps.

A device-dependent bitmap (DDB) is a GDI object managed by GDI and device drivers, having the same status as a logical pen object, a logical brush object, or a region object. When creating a DIB, GDI and the graphic device driver defines its internal data format and allocates memory for it from GDI's storage space. After that, all accesses to a DDB go through a GDI object handle. GDI uses HBITMAP as the type for device-dependent bitmap handles. DDBs are commonly known as bitmaps, or GDI bitmap objects.

GDI provides a very rich set of functions for device-dependent bitmap, because it's used everywhere by the operating system itself. DDBs can be used in geometric pens, pattern brushes, caret, menu displays, and common controls.

There are quite a few ways to create a GDI bitmap object:

```
HBITMAP CreateBitmap(int nWidth, int nHeight,UINT cPlanes,  
                     UINT cBitsPerPel, CONST VOID * lpvBits);  
HBITMAP CreateBitmapIndirect(CONST BITMAP * lpbm);  
HBITMAP CreateCompatibleBitmap(HDC hDC, int nWidth, int nHeight);  
HBITMAP CreateDiscardableBitmap(HDC hDC, int nWidth, int nHeight);  
HBITMAP CreateDIBitmap(HDC hdc, CONST BITMAPINFOHEADER * lpbmh,  
                      DWORD fdwInit, CONST VOID * lpblInit,  
                      CONST BITMAPINFO * lpbmi, UINT fuUsage);  
HBITMAP LoadBitmap(HINSTANCE hInstance, LPCTSTR lpBitmapName);
```

There are several noticeable differences between a DDB and a DIB. By its original design, a device-dependent bitmap is device dependent. This means that each graphics device can choose its own internal bitmap format to represent a device-dependent bitmap. If an application uses DDB, the actual runtime format may differ when running on different machines, even on the same machine with different display settings. A device-dependent bitmap also has an array of pixels, just like a device-independent bitmap. When passing data to a DDB, or retrieving data from a DDB, its scan lines always appear in top-down order, so you don't have to deal with negative height for bottom-up images. Unlike DIBs, which uniformly use a single-plane scan line, a DDB may use multiple planes to be compatible with a certain graphics device for best performance. The input pixel data to DDB creation routines needs to be 16-bit WORD aligned. A DDB does not have a color table associated with it, so the real color of each pixel in the image depends on the device the image is displaying on.

CreateBitmap

A DDB is defined by a width, a height, a plane count, a bits-per-pixel count, and an array of pixel colors or indexes. These five values form the parameter list to the Create Bitmap function. The CreateBitmap function creates an nWidth by nHeight image with data in cPlanes planes and cBitsPerPel bits per pixel per plane format. The lpvBits parameter points to the initial pixel array, which is expected to be $(nWidth * cBitsPerPel + 15)/16 * 2 * cPlanes * nWidth * nHeight$ in size. GDI will allocate that amount of memory and copy the initialization data to it. If lpvBits is NULL, the initial contents of the bitmap is not defined.

The bitmap data for a DIB and a DDB are managed totally differently. A DIB is defined by either a packed DIB with a single pointer or by a nonpacked DIB with two pointers. All these pointers point to user mode address space, which is backed up either by a memory-mapped file or a system paging file. The size of a DIB is only limited by your free hard disk space, until you overflow your two-gigabyte user mode address space. For Windows 95/98, DDBs are stored into GDI's 32-bit heap, although GDI's implementation is virtually all 16-bit code. A DDB is limited to 16 MB in size on these systems. Each scan line is limited to 64 KB. For NT-based systems since Windows NT 4.0, DDBs are allocated off the paged-pool, which lives in kernel address space. The kernel paged-pool stores lots of other GDI objects like regions, device contexts, paths, and other non-GDI objects. Each DDB is limited to 48 MB in size, whereas the whole paged-pool has a 192-MB limit. A DDB consumes another limited system resource, a GDI object handle. To summarize, a DDB consumes system-wide shared resources instead of per-process resources, so extreme care should be taken in creating large DDBs, creating lots of DDBs, or leaking DDBs.

The only standard DDB format is the 1-bit-per-pixel, single-plane monochrome-bitmap format. For other pixel formats, the nPlanes and cBitsPerPel parameter only serves as a minimum requirement for GDI. Modern graphics devices use the standard single-plane DIB format internally. GDI relies on a device driver to choose a next-available format that has at least nPlanes * cBitsPerPel as its bit count. For example, a 3-plane, 8-bits-per-pixel request is met by a single-plane, 24-bits-per-pixel DDB; a 3-plane, 10-bits-per-pixel request is met by a single-plane, 32-bits-per-pixel DDB. The nPlanes and cBitsPerPel also determines how the initial pixel array data should be interpreted. For the current GDI implementation, when nPlanes * cBitsPerPel is more than 32, the creation call will fail.

The following code fragment shows how to create a 1-bpp DDB initialized with a 4-by-4 chessboard pattern, and a 24-bpp uninitialized DIB.

```
const WORD Data88_1pp[] = { 0xCC, 0xCC, 0x33, 0x33,
                            0xCC, 0xCC, 0x33, 0x33 };
HBITMAP hBmp1bpp = CreateBitmap(8, 8, 1, 1, Data88_1pp);
HBITMAP hBmp24bpp = CreateBitmap(8, 8, 3, 8, NULL);
```

CreateBitmapIndirect

The CreateBitmapIndirect function provides another way of creating a DDB through a pointer to a BITMAP structure, which is defined as:

```
typedef struct tagBITMAP {  
    LONG  bmType; // bitmap type, must be 0  
    LONG  bmWidth;  
    LONG  bmHeight;  
    LONG  bmWidthBytes;  
    WORD   bmPlanes;  
    WORD   bmBitsPixel;  
    LPVOID bmBits;  
} BITMAP;
```

Compared with the parameter list to CreateBitmap, the BITMAP structure has three differences. Although the bmType field is new, it must be 0. The bmWidthBytes is the number of bytes per scan line of the pixel array. It must be an even number as required by CreateBitmap. The bmBits field points to the pixel array, but it's not defined as a constant pointer. Microsoft documentation mistakenly states that each scan is a multiple of 32 bits. This is not a requirement. In a real implementation, GDI always tries to merge calls into a few system service calls. CreateBitmapIndirect calls Create Bitmap. But if the bmWidthBytes is more than what's required for 16-bit WORD alignment, GDI actually allocates a temporary memory block off the system heap and copies the initial pixel array before calling CreateBitmap.

GetObject on DDB

When calling CreateBitmap or CreateBitmapIndirect, the parameters are just in requirement and initial pixel array format. GDI, or the graphics device driver, may decide to store the bitmap in an appropriate format supported by the hardware. After all, it's allowed, because it's device-dependent. Given a DDB object handle, GetObject gives a glimpse of the actual format used to represent the bitmap. The application does not have direct access to a device-independent bitmap, so the information is not complete. For example, the bmBits field returned by GetObject is always NULL, because GDI would not tell you where the bits are stored. The bmWidthBytes field is always rounded up to an even number, but internally the bitmap may be stored in a DIB (DWORD-aligned) format, the native format supported by the NT-based graphics engine.

Here is a sample usage of CreateBitmapIndirect and GetObject:

```
DWORD Chess44[] = { 0xCC,0xCC,0x33,0x33, 0xCC,0xCC,0x33,0x33 };  
BITMAP bmp = { 0, 8, 8, sizeof(Chess44[0]), 1, 1, Chess44 };
```

```
HBITMAP hBmp = CreateBitmapIndirect(&bmp);
GetObject(hBmp, sizeof(bmp), & bmp);
DeleteObject(hBmp);
```

The code fragment creates a DDB using `CreateBitmapIndirect` with a `DWORD` aligned pixel array, and then queries the DDB object using `GetObject`. The `BITMAP` structure filled by `GetObject` changes the `bmWidthBytes` field from 4 bytes to 2 bytes, and `bmBits` to `NULL`. If you change the `Chess44` array to be a `WORD` array, the result is the same.

CreateCompatibleBitmap and CreateDiscardableBitmap

Although you can create color bitmaps with `CreateBitmap` or `CreateBitmapIndirect`, the result may not be compatible with the device context you want to display on. A valid DDB incompatible with a device context may be rejected when actually used in that device context. So to create color bitmaps that are guaranteed to be device-compatible, use the `CreateCompatibleBitmap` function instead.

`CreateCompatibleBitmap` looks much simpler, accepting a reference device context handle and the required height and width of the bitmap. `CreateCompatibleBitmap` does not need the plane-count and bits-per-pixel parameters, as they can be calculated based on the reference device context. If the device context corresponds to a physical graphics device, GDI uses its plane count and bit count to create the bitmap. For example, if screen DC is used, `CreateCompatibleBitmap` creates an 8-bpp DDB if running in 256-color mode and a 32-bpp DDB if running in 32-bit true color mode. So if an application asks for a 1024×1024 DDB, the size of the memory required would be 1 MB in 256-color mode and 4 MB in 32-bit true color mode. If the device context is a memory device context, GDI creates a bitmap having the same pixel format as the current bitmap selected in the device context. We mentioned in the last section that a new memory device context has a single-pixel monochrome bitmap selected, so `CreateCompatibleBitmap` using such a device context generates a monochrome bitmap.

The following routine answers a few frequently asked questions about DDB—why `CreateCompatibleBitmap` failed to create a DDB, what's the maximum size supported by DDBs? The routine `LargestDDB` takes a device context handle and uses a binary search algorithm to find the largest DDB compatible with it.

```
HBITMAP LargestDDB(HDC hDC)
{
    int mins = 1;      // 1    pixel
    int maxs = 1024 * 128; // 16 Giga pixels

    while ( true ) // binary search for largest DDB
    {
        int mid = (mins + maxs)/2;
        HBITMAP hBmp = CreateCompatibleBitmap(hDC, mid, mid);
```

```
if ( hBmp )
{
    HBITMAP h = CreateCompatibleBitmap(hDC, mid+1, mid+1);
    if ( h==NULL )
        return hBmp;

    DeleteObject(h);
    DeleteObject(hBmp);
    mins = mid+1;
}
else
    maxs = mid;
}

return NULL;
}
```

When passed a default memory device context, a quite large monochrome DDB can be generated; when passed a display device context, the largest DDB is much smaller in pixel size because of higher color depth. [Table 10-5](#) summarizes the results of running the code on a device context with different color depths.

These seemingly simple statistics could change your program design. If an application uses DDBs, the size of a single DDB is effectively limited to 3.96 megapixels, considering the worst case of running on Windows 95/98 using 32-bpp display mode. Digital cameras are offering more than 2 megapixels nowadays. So an application can't even store one 2:1 scaled digital camera image using a DDB, which will be 16 MB.

In the old Win16 days, memory was a scarce resource. GDI provides the Create DiscardableBitmap function to create a discardable bitmap. The idea is to allow GDI to free the resource used in discardable bitmaps when GDI is running low on resources. Whenever an application wants to use a discardable bitmap, it needs to check if the bitmap is still valid, and recreate it if not. Although CreateDiscardableBitmap is still provided in the Win32 GDI, it does not create a discardable bitmap. Instead the function just calls CreateCompatibleBitmap. With 32-bit operating systems, the memory consumption of device-dependent bitmaps is still a big issue, especially with high-resolution true color display mode. But Win32 programs can use the DIB section to move memory usage on the GDI part to user applications.

Table 10-5. Largest Compatible DDB

DC Color Depth	Windows NT/2000	Windows 95/98
1	17,408 × 17,408, 36.125 MB	11,474 × 11,474, 15.71 MB
8	6144 × 6144, 36 MB	4079 × 4079, 15.87 MB
16	4352 × 4352, 36.125 MB	2880 × 2880, 15.82 MB
24	3584 × 3584, 36.76 MB	2352 × 2352, 15.82 MB
32	3072 × 3072, 36 MB	2039 × 2039, 15.86 MB

CreateDIBitmap

The functions we described so far still do not provide an easy way to create an initialized color DDB. Although CreateBitmap and CreateBitmapIndirect do allow a pointer to a pixel array in the parameter, the complexity of a color image is not adequately captured. A device-independent bitmap, on the other hand, provides a well-defined way to describe common color image formats. So a DIB is an indisputable way of describing a color image. GDI provides CreateDIBitmap to create an initialized DDB based on a DIB, or in some sense convert a DIB to a DDB.

The CreateDIBitmap function can be seen to work in two steps: first to create a DDB, second to convert a DIB to the DDB. The hdc parameter is a handle to a reference device context, which the created DDB should be compatible with. If NULL is passed, a monochrome DDB is created. The lpbmih parameter points to a bitmap information header structure. But only its width and height fields are actually used. If the height is negative, its absolute value is used. Other fields like bits count and compression modes are not used.

Initializing the newly created DDB is optional as controlled by the fdwInit parameter. If it's CBM_INIT, the next three parameters form a complete specification of an unpacked DIB. The lpblInit parameter points to a pixel array, the lpbmi parameter points to bitmap information header and color table, and fuUsage tells whether the color table contains a palette index or RGB color.

The following function can be added to the KDIB class to convert a DIB to a DDB.

```
HBITMAP KDIB::ConvertToDDB(HDC hDC)
```

```
{
    return CreateDIBitmap(hDC, m_pBMI->bmiHeader,
        CBM_INIT, m_pBits, m_pBMI,
        DIB_RGB_COLORS);
}
```

When a palette is involved in the DIB-to-DDB translation, the palette currently selected in the device context is used.

LoadBitmap

A common practice in Windows programming is to attach bitmaps as a resource to a module and load them as DDBs using the LoadBitmap function.

The LoadBitmap function accepts two parameters: hInstance is a handle to the module that contains the bitmap resource, and lpBitmapName is the name of the bitmap resource. If NULL is passed as the hInstance parameter, the second parameter can be predefined constants like OBM_BTNCORNERS, OBM_CHECK, etc., to access dozens of system predefined bitmaps. The bitmaps are actually either coming directly from USER32.DLL or being synthesized by it. If an integer identifier is used in the resource file to identify the bitmap, the MAKEINTRESOURCE macro can be used to cast an integer to a character string pointer.

Bitmap resources are stored in a packed DIB format in a Win32 module. What LoadBitmap does is find the bitmap resource, lock it to get a packed DIB handler, and create a DDB with it that is compatible with the current screen display. For monochrome bitmaps—that is, DIBs with only black and white in their color table—GDI is smart enough to use a monochrome DDB format instead of the more memory-consuming color format. When running in 256-color palletized display mode, loading high color or true color images could generate low-quality images, because the application does not have a good way to control the color conversion.

The following code fragment loads a toolbar bitmap from BROWSEUI.DLL.

```
HINSTANCE hMod = LoadLibrary("browseui.dll");
HBITMAP hBmp = LoadBitmap(hMod, MAKEINTRESOURCE(261));
FreeLibrary(hMod);
```

Microsoft documents that LoadBitmap has some trouble loading bitmaps larger than 64 KB on Windows 95, because of the underlying 16-bit code. When the size of a DIB resource is over 64 KB, its size is converted to a 16-bit value and a shift count in GDI's internal implementation. So the last few bits in the size may be cut off. A workaround is to pad the resource with extra zeros to round the size up.

A common pattern of LoadBitmap usage is loading a bitmap, displaying it once, and deleting the object after that. If performance is a concern, this pattern should be avoided. Converting a DIB to a DDB is a slow process, and a DDB consumes extra system resources. So using the DIB directly is a faster and low-resource alternative. But if the bitmap is loaded once, and used repetitively, using a DDB may save bitmap format conversion time needed during display compared with DIB.

Copying Bitmaps between DIBs and DDBs

Besides functions to create a new DDB, GDI provides two functions for copying pixels between a

DDB and a DIB.

```
int SetDIBits(HDC hdc, HBITMAP hbmp, UINT uStartScan,  
    UINT cScanLines, CONST VOID * IpvBits,  
    CONST BITMAPINFO * Ipbmi, UINT fuColorUse);  
int GetDIBits(HDC hdc, HBITMAP hbmp, UINT uStartScan,  
    UINT cScanLines, CONST VOID * IpvBits,  
    CONST BITMAPINFO * Ipbmi, UINT fuColorUse);
```

The two functions have the same parameter lists, although MSDN documentation uses parameter names with small differences. The first parameter specifies a reference-device context, whose palette will be used in the pixel array format conversion. The second parameter is a handle to an existing device-dependent bitmap object. The five remaining parameters specify a partial unpacked DIB and its color-table interpretation. We have already seen partial DIBs in the SetDIBitsToDevice function. A partial DIB can be either a complete DIB or a few continuous scan lines with a complete DIB. It's mainly designed to save memory, or to display partial bitmaps being downloaded from a slow connection. The IpvBits parameter points to the scan lines, uStartScan tells the scan-line number for the first scan line in the buffer, and cScanLines is the count of scan lines in the buffer.

SetDIBits converts pixels in the partial DIB to the DDB format, and copies the result to the DDB. GetDIBits converts pixels in the DDB format to the DIB format, and copies the result to the partial DIB buffer. SetDIBits is actually implemented using SetDIBitsToDevice using a memory device context as destination, which has the DDB selected. The palette from the device context referred to by hdc is selected into the memory device context during the conversion.

We have several ways to convert a DIB to a DDB. A DIB in resource can be loaded into a DDB using LoadBitmap. The trouble is that you don't have control over color conversion. CreateDIBitmap creates a brand-new DDB from a DIB, which is compatible to device context. So it provides more control over color conversion and at the same time is quite easy to use. SetDIBits is more primitive compared with LoadBitmap and CreateDIBitmap. Because it copies a partial DIB into a DDB, the caller can use a smaller buffer to do the conversion in multiple runs, it can put several small DIBs into a big DDB, and it can control the DDB format.

The GetDIBits function is unique in that it's the most common way of converting a DDB to a DIB. Converting a DDB to a DIB is quite complicated, because of the various DIB formats it needs to support. GetDIBits supports all valid DIB bit counts, RGB or pallettized, uncompressed and RLE compressed pixel array, and bit fields. The Ipbmi parameter points to a DIB information header that controls the DIB format. When RLE compressed is used, it's very hard for the application to know the exact size of the compressed image. So GetDIBits should be called twice, first with a NULL pointer to the pixel array to let GDI return the actual image size, and second to get the real image after allocating the required size.

The function in [Listing 10-3](#), BitmapToDIB, provides an easy-to-use wrap around the GetDIBits function. It accepts a GDI palette object handle to color table generation, a DDB object handle, a

required bit count, and a required DIB compression flag. The function calculates the size of the DIB, allocates a buffer, converts the DDB into the buffer, and returns a pointer to the buffer.

Listing 10-3 Function BitmapToDIB: Convert DDB to DIB

```
BITMAPINFO * BitmapToDIB(HPALETTE hPal, // palette for color conversion
                         HBITMAP hBmp,    // DDB for convert
                         int nBitCount, int nCompression) // format wanted
{
    typedef struct
    {
        BITMAPINFOHEADER bmiHeader;
        RGBQUAD      bmiColors[256+3];
    } DIBINFO;

    BITMAP ddbinfo;
    DIBINFO dibinfo;

    // retrieve DDB information
    if ( GetObject(hBmp, sizeof(BITMAP), & ddbinfo)==0 )
        return NULL;

    // fill out BITMAPINFOHEADER based on size and required format
    memset(&dibinfo, 0, sizeof(dibinfo));

    dibinfo.bmiHeader.biSize      = sizeof(BITMAPINFOHEADER);
    dibinfo.bmiHeader.biWidth     = ddbinfo.bmWidth;
    dibinfo.bmiHeader.biHeight    = ddbinfo.bmHeight;
    dibinfo.bmiHeader.biPlanes   = 1;
    dibinfo.bmiHeader.biBitCount = nBitCount;
    dibinfo.bmiHeader.biCompression = nCompression;

    HDC hDC = GetDC(NULL); // screen DC
    HGDIOBJ hpalOld;

    if ( hPal )
        hpalOld = SelectPalette(hDC, hPal, FALSE);
    else
        hpalOld = NULL;

    // query GDI for image size
    GetDIBits(hDC, hBmp, 0, ddbinfo.bmHeight, NULL,
              (BITMAPINFO *) &dibinfo, DIB_RGB_COLORS);

    int nInfoSize = sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) *
```

```
        GetDIBColorCount(dibinfo.bmiHeader);
        int nTotalSize = nInfoSize + GetDIBPixelSize(dibinfo.bmiHeader);

        BYTE * pDIB = new BYTE[nTotalSize];
        if ( pDIB )
        {
            memcpy(pDIB, & dibinfo, nInfoSize);

            if ( ddbinfo.bmHeight != GetDIBits(hDC, hBmp, 0, ddbinfo.bmHeight,
                pDIB + nInfoSize, (BITMAPINFO *) pDIB, DIB_RGB_COLORS) )
            {
                delete [] pDIB;
                pDIB = NULL;
            }
        }

        if ( hpalOld )
            SelectObject(hDC, hpalOld);

        ReleaseDC(NULL, hDC);
        return (BITMAPINFO *) pDIB;

    }
```

The BitmapToDIB function does not require the caller to supply a BITMAPINFO structure to simplify calling. It allocates a DIBINFO structure on the stack, which has 259 items in the color table, enough for a DIB using bit fields compression mode and 256 colors for a full palette. The width and height of the DIB is calculated based on the DDB size. After calling GetDIBits for the first time to query the real image size, it allocates a buffer, copies over the bitmap information header, and calls GetDIBits to get the whole DIB.

The GetDIBits function has another hidden feature. If you only set the biSize field of the bitmap information header structure, and leave everything else 0, GDI will fill it with the bit count and compression mode flag of the device context. This allows applications to query the exact pixel format a device context is using. This technique is especially useful if an application wants to deal with pixels directly in 16-bpp display modes. There are two common subtypes of 16-bpp display mode: either 5-5-5 or 5-6-5. Sometimes the application needs to know the exact pixel format. [Listing 10-4](#) shows a PixelFormat function to achieve this.

Listing 10-4 Function PixelFormat: Get Device Context Pixel Format

```
int PixelFormat(HDC hdc)
{
    typedef struct
    {
```

```
BITMAPINFOHEADER bmiHeader;
RGBQUAD      bmiColors[256+3];
} DIBINFO;

DIBINFO dibinfo;

HBITMAP hBmp = CreateCompatibleBitmap(hdc, 1, 1);

if ( hBmp==NULL )
    return -1;

memset(&dibinfo, 0, sizeof(dibinfo));

dibinfo.bmiHeader.biSize = sizeof(BITMAPINFOHEADER);

// 1st call to get hdc biBitCount.
GetDIBits(hdc, hBmp, 0, 1, NULL, (BITMAPINFO*) & dibinfo,
DIB_RGB_COLORS);

// 2nd call to get color table or bit fields
GetDIBits(hdc, hBmp, 0, 1, NULL, (BITMAPINFO*) & dibinfo,
DIB_RGB_COLORS);
DeleteObject(hBmp);

// try to understand bit fields
if ( dibinfo.bmiHeader.biBitCount==BI_BITFIELDS )
{
    DWORD * pBitFields = (DWORD *) dibinfo.bmiColors;
    DWORD red   = pBitFields[0];
    DWORD green = pBitFields[1];
    DWORD blue  = pBitFields[2];

    if ( (blue==0x001F) && (green==0x03E0) && (red==0x7C00) )
        return DIB_16RGB555;
    else if ( (blue==0x001F) && (green==0x007E) && (red==0xF800) )
        return DIB_16RGB565;
    else if ( (blue==0x00FF) && (green==0xFF00) && (red==0xFF0000) )
        return DIB_32RGB888;
    else
        return -1;
}

switch ( dibinfo.bmiHeader.biBitCount )
{
    case 1: return DIB_1BPP;
```

```
        case 2: return DIB_2BPP;
        case 4: return DIB_4BPP;
        case 8: return DIB_8BPP;
        case 24: return DIB_24RGB888;
        case 16: return DIB_16RGB555;
        case 32: return DIB_32RGB888;

    default: return -1;
}
```

SetDIBits and GetDIBits also support another GDI bitmap format, the DIB section, which will be discussed in [Section 10.7](#). It's documented (KB Q230499) that when GetDIBits is used to convert a 1-bpp or 4-bpp DIB section to an 8-bpp DIB, the color table of the DIB is not set up properly.

Access Raw DDB Pixel Array

The basic idea behind device-dependent bitmaps is that DDBs do not have a color table, and DDBs can have unique internal pixel format as defined by the hardware vendor. The only universal DDB format is the monochrome DDB format. So it does not make much sense to access the raw DDB pixel array directly, especially color DDBs.

But GDI provides a pair of functions to allow applications to access the raw DDB pixel array.

```
LONG GetBitmapBits(HBITMAP hBmp, LONG cbBuffer, LPVOID lpvBits);
LONG SetBitmapBits(HBITMAP hBmp, LONG cBytes, LPVOID lpBits);
```

The GetBitmapBits function copies the raw DDB pixel array to a buffer specified by the cbBuffer and lpvBits parameters. But how do you know how big a buffer you need to allocate? Microsoft fails to document that GetBitmapBits returns the size of the buffer when size is 0 and buffer pointer is NULL. The pixel array is copied without format and color translation. The SetBitmapBits function does the reverse operation: it copies a buffer into the bitmap's pixel array.

It's hard to think of valid uses for these two functions. One might be to implement some efficient algorithms on a DDB without involving the normal memory device context. For example, it is easy to reverse every color pixel, flip, and rotate the scan lines. Another use may be just to study the internal DDB format. The following routine shows some code to dump the pixel array of a DDB to a text window, assuming we have a HexDump routine. For Windows NT-based systems, almost all display cards use the single-plane DIB format frame buffer, so their DDB pixel array is quite close to the DIB pixel array. But when running in standard VGA display modes, or Windows 95-based systems, the DDB format may get complicated.

```
void DumpBitmap(HWND hWnd, HBITMAP hBmp)
{
    int size = GetBitmapBits(hBmp, 0, NULL);
    BYTE * pBuffer = new BYTE[size];

    if ( pBuffer )
    {
        GetBitmapBits(hBmp, size, pBuffer);
        HexDump(hWnd, pBuffer, size);
        delete [] pBuffer;
    }
}
```

[< BACK](#) [NEXT >](#)

10.6 USING DDBS

Device-dependent bitmaps are widely used in Windows programming. The last section focused on various ways of creating a DDB and converting between a DDB, a DIB and a raw pixel array. This section will discuss displaying a DDB and using a DDB in a menu, tool bar, etc.

Displaying DDBs

Although DDBs are very important, GDI does not provide a single function to display a DDB directly. Instead, to display a DDB, an application needs to create a memory device context, select the DDB into the memory device context, and then copy pixel data between the memory device context and the target device context. The nice thing about this device-context-based design is that now the DDB can be both the source and destination of a drawing operation. You can even copy one part of a DDB to another part of the same DDB. Compared with DIBs, GDI only provides routines to draw from a DIB—nothing to draw into a DIB.

GDI solves the problem of displaying a DDB by generalizing it to be a problem of transferring a rectangle array of pixels from one graphics device to another graphics device, each represented by a device context handle. Such transfers are traditionally called BitBlt, which stands for Bit boundary Block Transfer. Here are the two basic BitBlt functions.

```
BOOL BitBlt(HDC hdcDst, int nXDst, int nYDst, int nWidth, int nHeight,  
            HDC hdcSrc, int nXSrc, int nYSrc, DWORD dwRop);  
BOOL StretchBlt(HDC hDCDst, int nXDst, int nYDst, int nWDst, int nHDst,  
                 HDC hDCSrc, int nXSrc, int nYSrc, int nWSrc, int nHSrc,  
                 DWORD dwRop);
```

The BitBlt function transfers a rectangular block of pixels from the source device to a rectangle in the destination device. The source rectangle is defined by nXSrc, nYSrc, nWidth, and nHeight in the source device context's logical coordinate system. The destination rectangle is defined by nXDst, nYDst, nWidth, and nHeight in the destination device context's logical coordinate system. Both device contexts need to support RC_BITBLT raster capability. For example, if the source device context refers to a meta file device context, BitBlt will fail, because the metafile device, being a recording device, does not have a readable frame buffer. GDI also cannot handle the case when the source device context has a rotation or shearing transformation, which could turn the source rectangle into a parallelogram or a rectangle with sides not parallel to the axes. If the source rectangle and destination rectangle are not the same size in device coordinate space, the source image is stretched to fit into the destination rectangle. The destination device context is free to have any transformation, although the Windows 95-based system does not support world transformation. Just as with the Stretch DI Bits call, adjusting the source and destination rectangle allows you to flip the image horizontally and/or vertically. The last parameter dwRop is a ternary raster operation. It controls how source pixel, destination pixel, and brush are combined to determine the new destination pixel. For the moment, we're only using SRCCOPY, which overwrites the destination with source pixels.

The StretchBlt function is almost the same as the BitBlt function. The only difference is that the dimensions of the source and destination rectangles are defined independently. So StretchBlt is normally used when source and destination rectangles are of different size in logical coordinates. Whether stretching is really needed also depends

on the how logical coordinate spaces are set up.

Screen Scrambler

As an example on how to use BitBlt/StretchBlt, let's try to write a screen scrambler. The following program scrambles the screen display by stretching a randomly selected rectangle to another randomly selected rectangle. To make the display more colorful, the color uses a randomly generated raster operation and randomly generated solid color brush. The destination rectangle has a negative height and width, so the source rectangle will be rotated 180 degrees. The stretching makes the destination pixel twice as large as the source. After some iterations, you have a nice piece of unique artwork on your screen. We are not in business to make an actual screen-saver, so the code simply iterates 200 times and quits by asking the screen to repaint itself.

```
int WINAPI WinMain(HINSTANCE hInst, HINSTANCE, PSTR, int)
{
    HDC hDC = GetDC(NULL);

    int width = GetSystemMetrics(SM_CXSCREEN);
    int height = GetSystemMetrics(SM_CYSCREEN);

    for (int i=0; i<2000; i++)
    {
        HBRUSH hBrush = CreateSolidBrush(RGB(rand()%256,
                                              rand()%256, rand()%256));
        SelectObject(hDC, hBrush);

        StretchBlt(hDC, rand() % width, rand() % height, -64, -64,
                   hDC, rand() % width, rand() % height, 32, 32,
                   (rand() % 256) << 16);
        SelectObject(hDC, GetStockObject(WHITE_BRUSH));
        DeleteObject(hBrush);
        Sleep(1);
    }

    ReleaseDC(NULL, hDC);

    RedrawWindow(NULL, NULL, NULL, RDW_INVALIDATE | RDW_ALLCHILDREN);
    return 0;
}
```

Various Ways of Displaying DDBs

BitBlt and StretchBlt are normally used to draw device-dependent bitmaps. When used properly, the two functions can generate lots of graphics effects with bitmaps. [Listing 10-5](#) shows a simple DDB class. The KDDB::Draw routine can draw normal, centered, stretched, proportionally stretched, or tiled bitmaps.

Listing 10-5 Simple Device-Dependent Bitmap Class

```
class KDDB
{
protected:
    HBITMAP m_hBitmap;
    HBITMAP m_hOldBmp;
    void ReleaseDDB(void);

public:
    HDC    m_hMemDC;
    bool Prepare(int & width, int & height);

    typedef enum { draw_normal, draw_center, draw_tile,
                  draw_stretch, draw_stretchprop };

    HBITMAP GetBitmap(void) const
    {
        return m_hBitmap;
    }

    KDDB()
    {
        m_hBitmap = NULL;
        m_hMemDC = NULL;
        m_hOldBmp = NULL;
    }
    virtual ~KDDB()
    {
        ReleaseDDB();
    }

    BOOL Attach(HBITMAP hBmp);

    BOOL LoadBitmap(HINSTANCE hInst, int id)
    {
        return Attach ( ::LoadBitmap(hInst, MAKEINTRESOURCE(id)) );
    }

    BOOL Draw(HDC hDC, int x0, int y0, int w, int h, DWORD rop,
              int opt=draw_normal);
};

// query size, prepare memory DC, select bitmap into memory DC
bool KDDB::Prepare(int & width, int & height)
{
    BITMAP bmp;

    if ( ! GetObject(m_hBitmap, sizeof(bmp), & bmp) )

```

```
return false;

width = bmp.bmWidth;
height = bmp.bmHeight;

if ( m_hMemDC==NULL ) // ensure memdc is created
{
    HDC hDC = GetDC(NULL);
    m_hMemDC = CreateCompatibleDC(hDC);
    ReleaseDC(NULL, hDC);
    m_hOldBmp = (HBITMAP) SelectObject(m_hMemDC, m_hBitmap);
}

return true;
}

// Release resource
void KDDB::ReleaseDDB(void)
{
if ( m_hMemDC )
{
    SelectObject(m_hMemDC, m_hOldBmp);
    DeleteObject(m_hMemDC);
    m_hMemDC = NULL;
}

if ( m_hBitmap )
{
    DeleteObject(m_hBitmap);
    m_hBitmap = NULL;
}

m_hOldBmp = NULL;
}

BOOL KDDB::Attach(HBITMAP hBmp)
{
if ( hBmp==NULL )
    return FALSE;

if ( m_hOldBmp ) // deselect m_hBitmap
{
    SelectObject(m_hMemDC, m_hOldBmp);
    m_hOldBmp = NULL;
}

if ( m_hBitmap ) // delete current bitmap
    DeleteObject(m_hBitmap);
```

```
m_hBitmap = hBmp; // replace with new one

if ( m_hMemDC ) // select if has memory DC
{
    m_hOldBmp = (HBITMAP) SelectObject(m_hMemDC, m_hBitmap);
    return m_hOldBmp != NULL;
}
else
    return TRUE;
}

BOOL KDDB::Draw(HDC hDC, int x0, int y0, int w, int h, DWORD rop, int
opt)
{
    int bmpwidth, bmpheight;

    if ( ! Prepare(bmpwidth, bmpheight) )
        return FALSE;
    switch ( opt )
    {
        case draw_normal:
            return BitBlt(hDC, x0, y0, bmpwidth, bmpheight,
                m_hMemDC, 0, 0, rop);

        case draw_center:
            return BitBlt(hDC, x0 + (w-bmpwidth)/2, y0 + ( h-bmpheight)/2,
                bmpwidth, bmpheight, m_hMemDC, 0, 0, rop);
            break;

        case draw_tile:
        {
            for (int j=0; j<h; j+= bmpheight)
                for (int i=0; i<w; i+= bmpwidth)
                    if ( ! BitBlt(hDC, x0+i, y0+j, bmpwidth, bmpheight,
                        m_hMemDC, 0, 0, rop) )
                        return FALSE;

            return TRUE;
        }
        break;

        case draw_stretch:
            return StretchBlt(hDC, x0, y0, w, h, m_
                hMemDC, 0, 0, bmpwidth, bmpheight, rop);

        case draw_stretchprop:
        {
            int ww = w;
            int hh = h;
```

```
if ( w * bmpheight < h * bmpwidth ) // the minimum scale
    hh = bmpheight * w / bmpwidth;
else
    ww = bmpwidth * h / bmpheight;

// proportional scaling and centering
return StretchBlt(hDC, x0 + (w-ww)/2, y0 + (h-hh)/2, ww, hh,
    m_hMemDC, 0, 0, bmpwidth, bmpheight, rop);
}

default:
    return FALSE;
}
}
```

The KDD class has three member variables, one memory device context handle, and two HBITMAPs for the DDB and old DDB handle deselected from the memory DC. The main method to create a DDB is loading from resource. Instances of the KDD class should be put outside of the WM_PAINT message handler to minimize the costly process of converting the DIB formatted resource into a DDB and creating a memory device context. The KDD::Draw method implements various ways of drawing a bitmap as controlled by its last parameter. Currently, normal drawing, centerizing, tiling, stretching, and proportional stretching are implemented.

User interface design currently uses bitmaps more and more often in splash windows or in background displays. If a bitmap represents a small texture, tiling is often used; if a bitmap represents a single object, centering or proportional stretching is commonly used.

Window/Screen Capturing

Once a DDB is selected into a memory DC, GDI functions can be used to draw onto the bitmap. The simplest usage of that is implementing a window capturing routine, whole-screen capturing being its special case. Here is such a routine.

```
HBITMAP CaptureWindow(HWND hWnd)
{
    RECT wnd;

    if ( !GetWindowRect(hWnd, & wnd) )
        return NULL;

    HDC hDC = GetWindowDC(hWnd);

    HBITMAP hBmp = CreateCompatibleBitmap(hDC, wnd.right-wnd.left,
        wnd.bottom - wnd.top);

    if ( hBmp )
    {
```

```
HDC hMemDC = CreateCompatibleDC(hDC);
HGDIOBJ hOld = SelectObject(hMemDC, hBmp);

BitBlt(hMemDC, 0, 0, wnd.right - wnd.left, wnd.bottom - wnd.top,
hDC, 0, 0, SRCCOPY);

SelectObject(hMemDC, hOld);
DeleteObject(hMemDC);
}

ReleaseDC(hWnd, hDC);

return hBmp;
}
```

The CaptureWindow routine shown above returns a handle to a DDB, which can then be converted to a DIB and saved to a disk file or copied to the clipboard.

DDB Color Conversion

The source and destination device contexts may not have the same frame buffer format or palette setting. For example, the source bitmap may be a monochrome bitmap, and the destination surface may be 32-bpp color surface; or the source bitmap is a true color image and the destination surface is a monochrome surface. In this case, BitBlt/StretchBlt converts the source color pixels to match the destination color format.

When one of the device contexts is a memory device context with a DDB selected in it, the only mismatch case is when one of the device contexts is a monochrome device context. For example, if the screen is in 24-bpp mode, a memory device context is normally created to be compatible with it. LoadBitmap and CreateCompatible Bitmap only generate 24-bpp or monochrome bitmaps. GDI only allows selecting 24-bpp or monochrome DDB into the device context. If an application creates an 8-bpp bitmap, the bitmap creation succeeds, but selecting into the screen-compatible memory DC will fail.

When displaying a monochrome bitmap on a color device surface, GDI does not simply display black and white; instead, GDI allows coloring of the bitmap through the background color and text color attributes of a device context. The background color of a device context is white by default. The text color, which should be seen as foreground color, is black by default. They can be changed to any color reference using Set Bk Color or SetTextColor. In a monochrome bitmap, either 0 or 1 represents every pixel. Pixels having 0 are considered foreground pixels; those having 1 are considered background pixels. GDI draws a foreground pixel (0) using the text color of the destination device context and a background pixel (1) using the background color of the destination device context.

The following code fragment shows how to tile a monochrome bitmap using different background and foreground colors.

```
const COLORREF ColorTable[] = {
    RGB(0xFF, 0, 0),  RGB(0, 0xFF, 0),  RGB(0, 0, 0xFF),
    RGB(0xFF, 0xFF, 0), RGB(0, 0xFF, 0xFF), RGB(0xFF, 0, 0xFF)
};

for (int y=0; y<clientheight; y+= bmpheight )
for (int x=0; x<clientwidth; x+= bmpwidth )
```

```
{  
  
    SetTextColor(hDC, ColorTable[y/bmpheight]);  
    SetBkColor(hDC, ColorTable[x/bmpwidth] | RGB(0xC0, 0xC0, 0xC0));  
  
    BitBlt(hDC, x, y, bmpwidth, bmpheight, hMemDC, 0, 0, SRCCOPY);  
}  
  
Drawing a color bitmap into a monochrome device context requires mapping color pixels to either 0 or 1. We know  
that when a DIB is displayed on a monochrome device context, GDI tries to find the closest match. But when  
displaying a DDB, GDI does something totally different. Each pixel is compared with the background color of the  
source device context. Pixels matching the background color are converted to 1 (white), and the remaining pixels are  
converted to 0 (black). Note that only the background color is used in the conversion; the text color is not used.
```

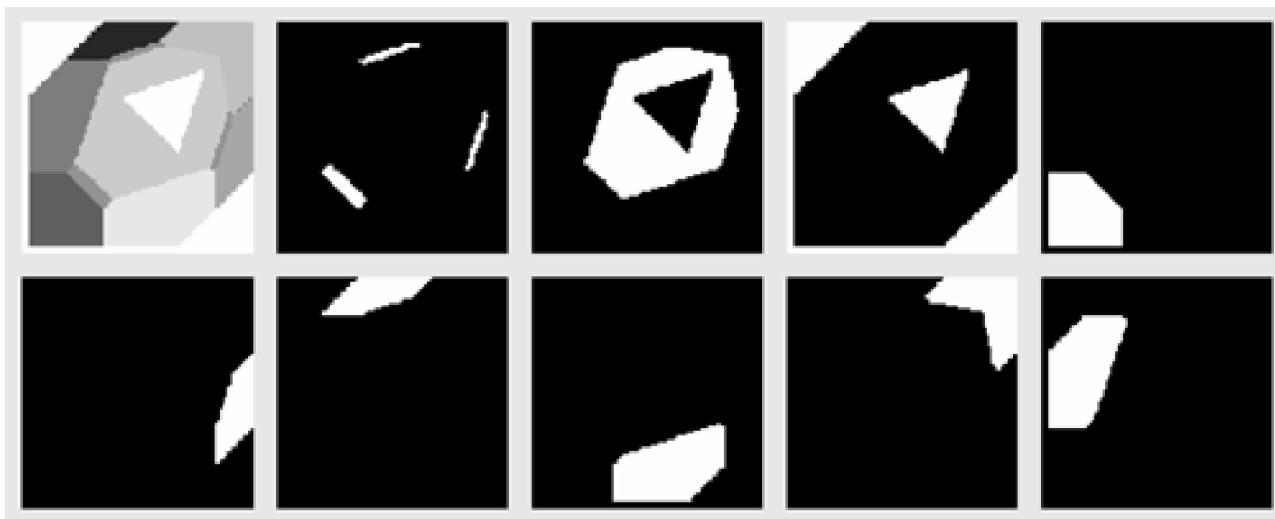
This seemingly naïve way of converting a color bitmap to a monochrome bitmap turns out to be very useful. It provides a simple way to divide pixels in a bitmap into the background pixels and nonbackground pixels by generating a bitmap which can be used as a masking bitmap. The masking bitmap can be used with ternary raster operations to display transparent bitmaps, or *sprites*. [Chapter 11](#) will cover all the details about transparent bitmap display.

Here is a new routine for the KDDB class; it generates a monochrome bitmap based on a background color. The routine uses CreateBitmap to create a monochrome bitmap, sets the color as the background color of the source memory device context, and uses BitBlt to convert a color bitmap to a monochrome bitmap.

```
HBITMAP KDDB::CreateMask(COLORREF crBackGround, HDC hMaskDC)  
{  
    int width, height;  
  
    if ( ! Prepare(width, height) )  
        return NULL;  
  
    HBITMAP hMask = CreateBitmap(width, height, 1, 1, NULL);  
    HBITMAP hOld = (HBITMAP) SelectObject(hMaskDC, hMask);  
  
    SetBkColor(m_hMemDC, crBackGround);  
    BitBlt(hMaskDC, 0, 0, width, height, m_hMemDC, 0, 0, SRCCOPY);  
  
    return hOld;  
}
```

[Figure 10-3](#) shows the result of KDDB::CreateMask to create 9 masks for every color used in a color image. The color bitmap is shown in the first position, followed by monochrome bitmaps generated using each of the colors as background. The mono chrome masks are displayed using the default background and text color, which maps 1 to white and 0 to black.

Figure 10-3. Map color DDB to monochrome masks.



Using Bitmaps in Menus

In Windows programming, an application is allowed to attach two small bitmaps with each menu item. One of them will be displayed next to the menu item when the menu item is unchecked, and the other when it's checked. By default, Windows does not use any bitmap for unchecked menu items and uses the standard check bitmap for checked items. But these bitmaps do not have to be associated with whether the item is checked or not. Adding a small printer bitmap before the "Print ..." menu item definitely makes the menu easier to understand.

The functions to change these check-mark bitmaps are `SetMenuItemBitmaps` and `SetMenuItemInfo`. Although these functions are quite easy to use in themselves, preparing and managing the bitmaps is not. To make the bitmap blend with the menu background, the background pixels in the image need to be changed to the menu's background color. The bitmaps also need to be scaled to the right size to fit into a menu bar. After the bitmap handles are passed to `SetMenuItemBitmaps` or `SetMenuItemInfo`, they can't be immediately deleted until the menu is no longer in use.

[Listing 10-6](#) shows a class for managing check-mark bitmaps.

Listing 10-6 Using Bitmaps as Menu Check Marks

```
class KCheckMark
{
protected:
    typedef enum { MAXSUBIMAGES = 50 };

    HBITMAP m_hBmp;
    int    m_nSubImageId[MAXSUBIMAGES];
    HBITMAP m_hSubImage [MAXSUBIMAGES];
    int    m_nUsed;

public:
    KCheckMark()
    {
        m_hBmp = NULL;
        m_nUsed = 0;
```

```
}

~KCheckMark();

void AddBitmap(int id, HBITMAP hBmp);
void LoadToolbar(HMODULE hModule, int resid,
                 bool transparent=false);
HBITMAP GetSubImage(int id);
BOOL SetCheckMarks(HMENU hMenu, UINT uPos, UINT uFlags,
                   int unchecked, int checked);

};

void KCheckMark::AddBitmap(int id, HBITMAP hBmp)
{
    if ( m_nUsed < MAXSUBIMAGES )
    {
        m_nSubImageId[m_nUsed] = id;
        m_hSubImage [m_nUsed++] = hBmp;
    }
}

void KCheckMark::LoadToolbar(HMODULE hModule, int resid, bool transparent)
{
    m_hBmp = (HBITMAP) ::LoadImage(hModule, MAKEINTRESOURCE(resid),
                                    IMAGE_BITMAP, 0, 0,
                                    transparent ? LR_LOADTRANSPARENT : 0);
    AddBitmap((int) hModule + resid, m_hBmp); // not reused as subimage
}

KCheckMark::~KCheckMark()
{
    for (int i=0; i<m_nUsed; i++)
        DeleteObject(m_hSubImage[i]);
}

HBITMAP KCheckMark::GetSubImage(int id)
{
    if ( id < 0 )
        return NULL;

    for (int i=0; i<m_nUsed; i++)
        if ( m_nSubImageId[i]==id )
            return m_hSubImage[i];

    BITMAP bmp;

    if ( !GetObject(m_hBmp, sizeof(bmp), &bmp) )
        return NULL;
```

```
if ( id *bmp.bmHeight >= bmp.bmWidth )
    return NULL;

HDC hMemDCS = CreateCompatibleDC(NULL);
HDC hMemDCD = CreateCompatibleDC(NULL);

SelectObject(hMemDCS, m_hBmp);

int w = GetSystemMetrics(SM_CXMENUCHECK);
int h = GetSystemMetrics(SM_CYMENUCHECK);

HBITMAP hRslt = CreateCompatibleBitmap(hMemDCS, w, h);

if ( hRslt )
{
    HGDIOBJ hOld = SelectObject(hMemDCD, hRslt);
    StretchBlt(hMemDCD, 0, 0, w, h, hMemDCS, id*bmp.bmHeight,
               0, bmp.bmHeight, bmp.bmHeight, SRCCOPY);
    SelectObject(hMemDCD, hOld);
    AddBitmap(id, hRslt);
}
DeleteObject(hMemDCS);
DeleteObject(hMemDCD);

return hRslt;
}

BOOL KCheckMark::SetCheckMarks(HMENU hMenu, UINT uPos, UINT uFlags,
                               int unchecked, int checked)
{
    return SetMenuItemBitmaps(hMenu, uPos, uFlags,
                            GetSubImage(unchecked), GetSubImage(checked));
}
```

The KCheckMark class supports loading a single bitmap that contains lots of small bitmaps tiled together, similar to bitmaps used in the toolbar. The bitmap is loaded with the LoadImage function, which supports replacing the background pixel with the default window color (COLOR_WINDOW), using the LR_LOADTRANSPARENT flag. The default window color is normally the same as the menu background color, so this solves the blending-with-background problem. The real workhorse of the class is the GetSubImage routine, which carves out a small subimage according to a sequential identifier. It assumes that subimages form a single row in the whole image, so the size of a subimage can be determined by the height of the image. The routine queries for the menu check-mark size and scales the subimage to it. The identifier and handle for the subimages are kept in a table so that they can be reused. The class uses the SetMenuItemImages routine to overwrite the default or existing check marks of a menu item. But the Windows operating system does not make copies of the bitmaps, so the handle table is also used in the class destructor to delete the bitmap objects. Instances of the KCheckMark class should be maintained in the window level or application level, so that their destructor is only called when the bitmaps are no longer in use.

[Figure 10-4](#) shows the result of using bitmaps to enhance common menu items. The original bitmap is loaded from

browserui.dll, resource id 275.

Figure 10-4. Bitmaps to enhance common menu items.

 Back

 Forward

 Favorites

 Add to Favorites

 View Tree

 Cut

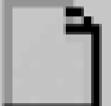
 Copy

 Paste

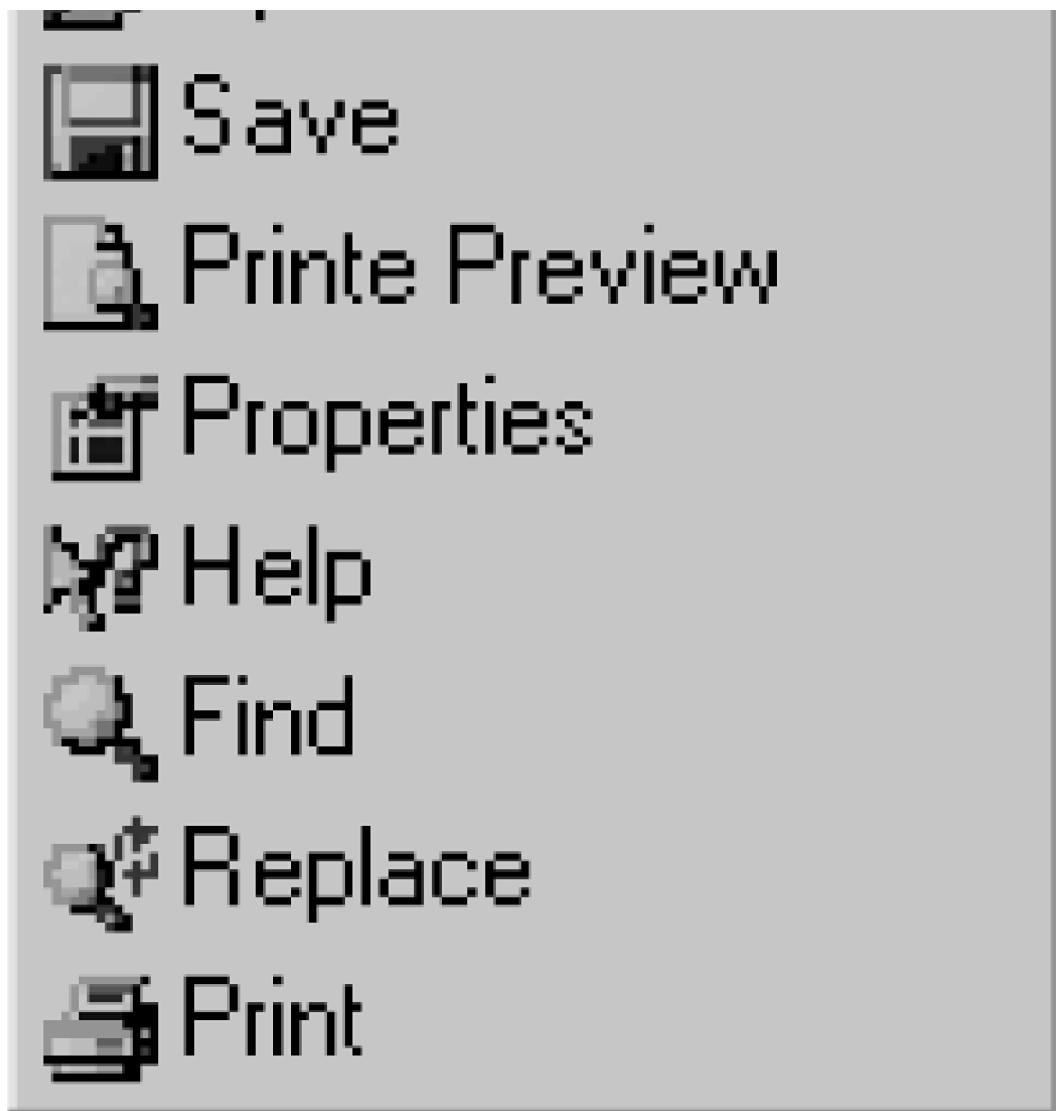
 Undo

 Redo

 Delete

 New

 Open



Bitmaps can also be used to replace the normal text display in a menu. This can be done using AppendMenu, InsertMenuItem, or SetMenuItemInfo. The following routine shows how to build a submenu using bitmaps.

```
void KBitmapMenu::AddToMenu(HMENU hMenu, int nCount, HMODULE hModule,
    int nID[], int nFirstCommand)
{
    m_hMenu      = hMenu;
    m_nBitmap    = nCount;

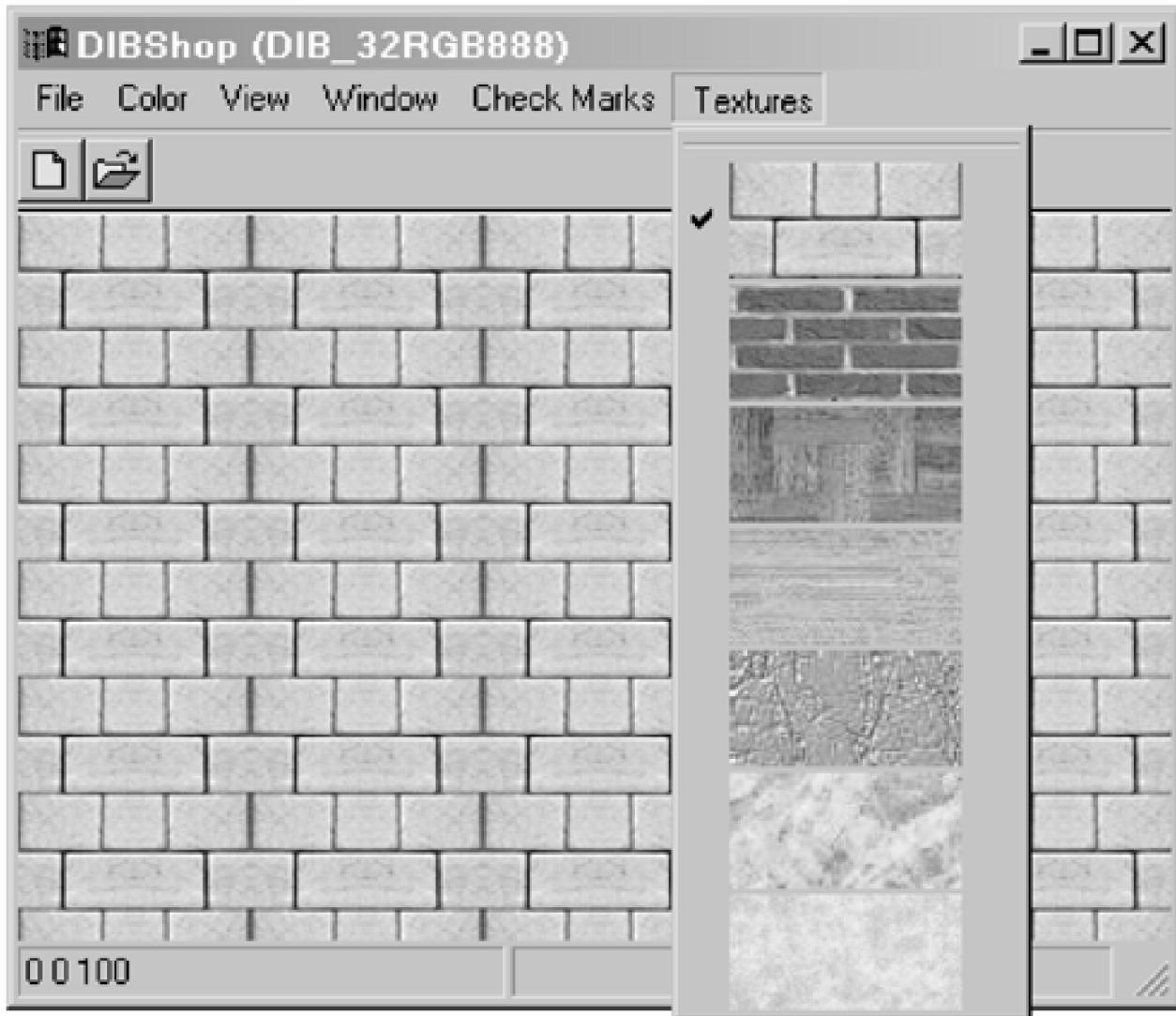
    m_nChecked   = 0;
    m_nFirstCommand = nFirstCommand;

    for (int i=0; i<nCount; i++)
    {
        m_hBitmap[i] = LoadBitmap(hModule, MAKEINTRESOURCE(nID[i]));
        if ( m_hBitmap[i] )
            AppendMenu(hMenu, MF_BITMAP, nFirstCommand + i,
```

```
(LPCTSTR) m_hBitmap[i]);  
}  
CheckMenuItem(m_hMenu, m_nChecked + nFirstCommand,  
    MF_BYCOMMAND | MF_CHECKED);  
}
```

[Figure 10-5](#) shows a texture bitmap menu and a window drawn using the KDDB ::Draw routine to tile texture bitmap.

Figure 10-5. Texture bitmap menu and bitmap tiling.



Using bitmaps in a menu through Win32 API still has its limitations. For example, bitmaps with a high number of colors would not be displayed nicely on the menu when running on 256-color display mode. The size of check marks is normally limited to 13 by 13 pixels, which is smaller than the 16-by-16 or 20-by-20 bitmap used on toolbars. You may also not like the way the system highlights menu items. To have full control over bitmap display in a menu, the owner draw menu is the solution. But it's outside the scope of this book.

Using a Bitmap as Window Background

A frequently asked question on using bitmaps is how to paint a bitmap in the background of a window, most likely an MDI client window, a dialog box, a property sheet, or one of the static controls. The Window background is normally handled by WM_ERASEBKGND message. A window message handler handing this message can draw whatever it wants to draw in the background. If this message is not handled, the default window message handler will draw the background using the background brush defined in the window class definition.

So the key problem here is handling WM_ERASEBKGND message. An application normally does not provide window message handlers for the MDI client window, dialog box, property sheet, or the static controls. They are provided by the operating system, either in USER32.DLL or COMMCTRL.DLL. So, to take over the background painting, subclassing is needed. Once you hook into the right place, drawing a bitmap is a simple problem.

[Listing 10-7](#) shows a generic class for taking over background drawing through subclassing.

Listing 10-7 Generic Background Painting Class

```
return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

LRESULT KBackground::WndProc(HWND hWnd, UINT uMsg, WPARAM wParam,
    LPARAM lParam)
{
    if ( uMsg == WM_ERASEBKGND )
        return EraseBackground(hWnd, uMsg, wParam, lParam);
    else
        return CallWindowProc(m_OldProc, hWnd, uMsg, wParam, lParam);
}

LRESULT KBackground::BackGroundWindowProc(HWND hWnd, UINT uMsg,
    WPARAM wParam, LPARAM lParam)
{
    KBackground * pThis = (KBackground *) GetProp(hWnd, Prop_KBackground);

    if ( pThis )
        return pThis->WndProc(hWnd, uMsg, wParam, lParam);
    else
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

BOOL KBackground::Attach(HWND hWnd)
{
    SetProp(hWnd, Prop_KBackground, this);
    m_OldProc = (WNDPROC) SetWindowLong(hWnd, GWL_WNDPROC,
        (LONG) BackGroundWindowProc);

    return m_OldProc!=NULL;
}

BOOL KBackground::Detach(HWND hWnd)
{
    RemoveProp(hWnd, Prop_KBackground);

    if ( m_OldProc )
        return SetWindowLong(hWnd, GWL_WNDPROC, (LONG) m_OldProc) ==
            (LONG) BackGroundWindowProc;
    else
        return FALSE;
}
```

The KBackground class defines two virtual method functions, besides the virtual destructor. The EraseBackground method handles window background drawing, which defaults to calling DefWindowProc. The WndProc method handles all window messages, although we are only interested in WM_ERASEBKGND message here. To subclass an existing window, the Attach method needs to be called. It attaches a property to the window whose value is a pointer to a KBackground class instance, and overrides the window's message procedure to a static function BackGroundWindowProc. When messages go to BackGroundWindowProc, it queries the attached property to find

the “this” pointer of the KBackground instance, and dispatches the call to its WndProc method. Note, we can't use the GWL_USERDATA field to store “this” pointer as we do with the KWindow class, because the window being subclassed may be created by another party which uses the GWL_USERDATA field. We can't use global variables either, because we want to use the class to support multiple windows at the same time, unless we want to build a global window map like MFC does.

The KBackground class is supposed to handle the generic problem of taking over background painting. Background can be drawn using lines forming a grid, area filling functions, DIB drawing functions, or DDB drawing functions. Each of them can be a derived class, which derives from the generic KBackground class. [Listing 10-8](#) shows a DDB-based implementation.

Listing 10-8 Class for Drawing a DDB Background

```
class KDDBackground : public KBackground
{
    KDD m_DDB;
    int m_nStyle;

    virtual LRESULT EraseBackground(HWND hWnd, UINT uMsg,
                                    WPARAM wParam, LPARAM lParam);

public:
    KDDBackground()
    {
        m_nStyle = KDD::draw_tile;
    }

    void SetStyle(int style)
    {
        m_nStyle = style;
    }
    void SetBitmap(HMODULE hModule, int nRes)
    {
        m_DDB.LoadBitmap(hModule, nRes);
    }
};

// Implementation
LRESULT KDDBackground::EraseBackground(HWND hWnd, UINT uMsg,
                                         WPARAM wParam, LPARAM lParam)
{
    if ( m_DDB.GetBitmap() )
    {
        RECT rect;
        HDC hDC = (HDC) wParam;

        GetClientRect(hWnd, & rect);
        HRGN hRgn = CreateRectRgnIndirect(&rect);
```

```
SaveDC(hDC);
SelectClipRgn(hDC, hRgn);
DeleteObject(hRgn);

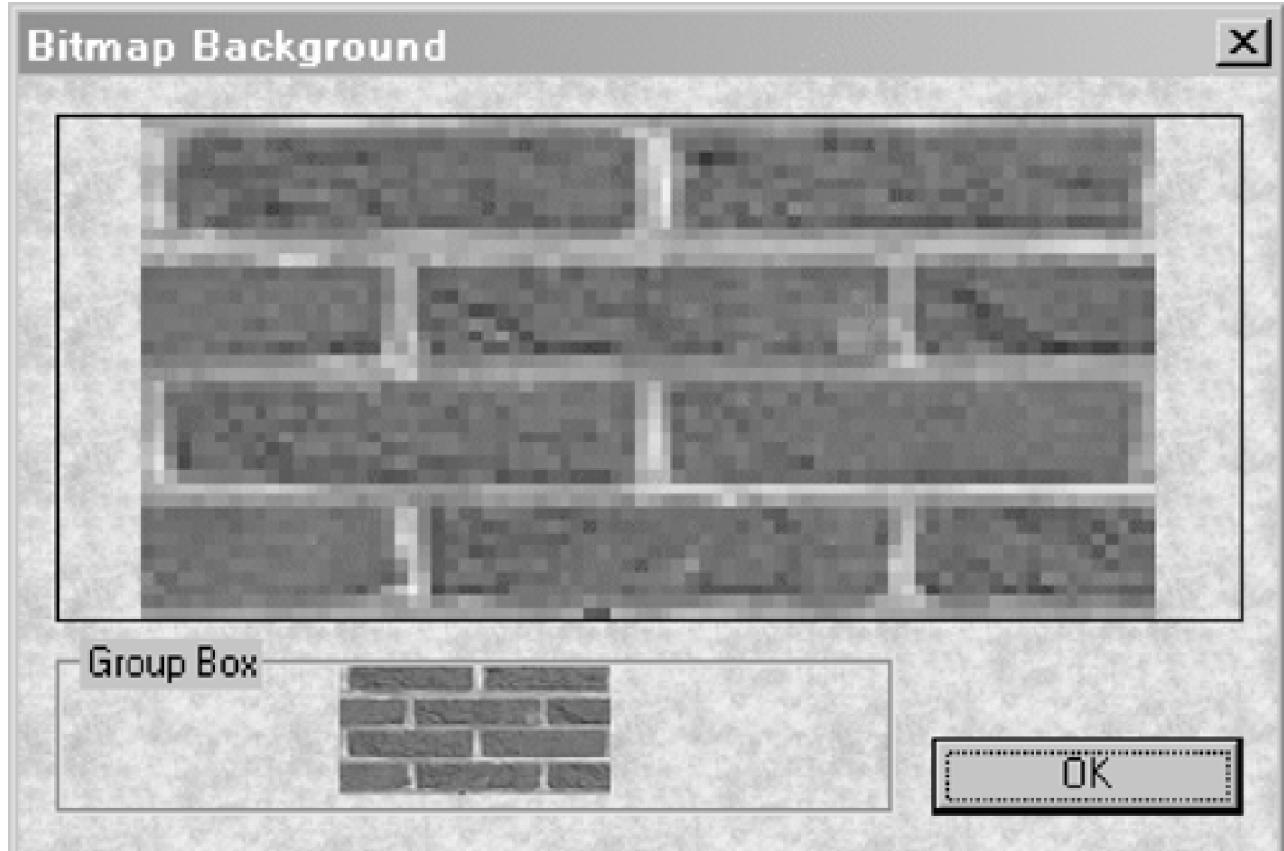
m_DDB.Draw(hDC, rect.left, rect.top, rect.right - rect.left,
    rect.bottom - rect.top, SRCCOPY, m_nStyle);
RestoreDC(hDC, -1);

return 1; // processed
}
else
return DefWindowProc(hWnd, uMsg, wParam, lParam);
}
```

The KDDBackground class uses the KDD class to handle DDB loading and drawing. It overrides the EraseBackground method to handle background drawing. Using the class is quite easy. You just need to create an instance of the KDDBackground class, set up a bitmap and drawing style, and then subclass a window with the Attach method.

[Figure 10-6](#) shows the result of subclassing a dialog box, a group box, and a static frame control inside it. A wooden texture is tiled in the dialog box, a brick is centered in the group box, and the brick is also stretched proportionally in the static frame control. Best of all, only three lines of code are needed for each window, all in WM_INITDIALOG handling.

Figure 10-6. Using KDDBackground class in a dialog.



```
// KDDBackground whole;  
// KDDBackground groupbox;  
// KDDBackground frame;  
  
whole.SetBitmap(m_hInstance, IDB_PAPER01);  
whole.setStyle(CDDB::draw_tile);  
whole.Attach(hWnd);  
  
groupbox.SetBitmap(m_hInstance, IDB_BRICK02);  
groupbox.setStyle(CDDB::draw_center);  
groupbox.Attach(GetDlgItem(hWnd, IDC_GROUPBOX));  
  
frame.SetBitmap(m_hInstance, IDB_BRICK02);  
frame.setStyle(CDDB::draw_stretchprop);  
frame.Attach(GetDlgItem(hWnd, IDC_FRAME));
```

[< BACK](#) [NEXT >](#)

10.7 DIB SECTION

So far we have discussed two bitmap formats supported by GDI: device-independent bitmaps (DIBs) and device-dependent bitmaps (DDBs). A DIB can have a variety of standard color-formats for different needs. A DDB, for practical reasons, can either be monochrome or have the same color format as the device. Using GDI, you can draw a DIB or a DDB, but GDI only supports drawing on a DDB when it's selected into a memory DC, not a DIB. The good thing about a DIB is that its storage is managed by the application, so the application can directly access its color table and pixel array, but GDI would not lend a hand to you in setting up a DIB. The good thing about a DDB is that you can use GDI to draw into it, but you don't have direct access to its storage because it's managed by GDI. Because DDBs are stored in system memory, there are limitations on both the maximum size of a single DDB, and the total size of all DDBs in the system. On the other hand, because DIB storage is managed by the application, it's only limited by the amount of free per-process virtual address space, and the amount of free disk space assigned to the system swap file. A natural question to ask is, can we have a bitmap type that has all the benefits of DIB and DDB? The answer is yes: a new bitmap type supported by the Win32 GDI API, DIB section.

The funny thing about DIB section is that it does not have a proper name. Probably an engineer who did not even bother to name it properly implemented it first, and then the technical writing community did not have a clue how to document it properly. Microsoft documentation defines a DIB section to be a DIB that applications can write to directly. But applications have always been able to write to a DIB without using a DIB section since Windows 3.1. Some documentation suggests a DIB section to be a section of a DIB, but actually a DIB section always contains a complete DIB.

To keep the record straight, a DIB section is a DIB that both the application and GDI can read and write directly. You may ask, what does this have to do with the word "section"? The pixel array of a DIB section can be from a memory-mapped file object, which is called a section object among the Windows operating system developers. Even when a DIB section is not created from a memory-mapped file, its pixel array storage is in virtual memory, which is backed up by the system paging file. A better name for DIB section would be Dual Access Device Independent Bitmap.

A DIB section is a GDI object, similar to a device-dependent bitmap. When you create a DIB section, GDI returns a handle to a GDI DIB section object, which happens to use the same type HBITMAP. But unlike a DDB, GDI also returns the address of the DIB section's pixel array so that applications can access its pixel array. After it has finished using a DIB section, an application should call DeleteObject to release the resource associated with it.

DIB sections share API functions with DDBs, so only one new structure and three new functions are needed to support DIB sections at the GDI API level.

```
HBITMAP CreateDIBSection(HDC hDC, CONST BITMAPINFP *pbmi, UINT iUsage,  
    PVOID * ppvBits, HANDLE hSection, DWORD dwOffset);  
UINT GetDIBColorTable(HDC hDC, UINT ustartIndex, UINT cEntries,  
    RGBQUAD * pColors);  
UINT SetDIBColorTable(HDC hDC, UINT ustartIndex, UINT cEntries,  
    CONST RGBQUAD * pColors);
```

```
typedef struct tabDIBSection {
```

```
BITMAP      dsBm;
BITMAPINFOHEADER dsBmih;
DWORD       dsBitfields[3];
HANDLE      dshSection;
DWORD       dsOffset;
} DIBSECTION;
```

CreateDIBSection

The CreateDIBSection is the fundamental function to create a DIB section object. The first three parameters are the essential parameters. The first parameter is the reference device context handle. The pbmi parameter points to a BITMAPINFO structure that contains a bitmap information header, bit masks, and color table. The iUsage parameter tells whether the color table contains the RGB value of the palette index. If it's DIB_PAL_COLORS, the logical palette currently selected in hdc is used. So the first three parameters completely define the dimensions, pixel format, and color table of the DIB section. The fourth parameter, ppvBits, is the address of a pointer variable, into which the address of the DIB section pixel array will be filled by GDI. The remaining two parameters are for providing the storage space and initial value of the pixel array using a block of memory from a memory-mapped file object. The hSection parameter is a file-mapping object handle returned by CreateFileMapping. Note that this parameter is named hSection, because a file-mapping object is also known as a section object. It may have contributed to the confusing name DIB section. The dwOffset parameter is the offset of the pixel array within the mapped file. The CreateDIBSection returns two values, a handle to a GDI DIB section object returned through the function return value, and a pointer to the pixel array returned through the ppvBits parameter.

Although the CreateDIBSection function looks complicated, an application normally only needs to worry about the second parameter, a pointer to a BITMAPINFO structure. That is, to create a DIB section, you need to specify its width, height, bit count, compression type, bit masks, and color table. GDI does not support all the valid DIB formats in a DIB section, because a DIB section is supposed to be both readable and writeable (drawable). So GDI only supports the uncompressed DIB format in a DIB section. You can't create a DIB section using BI_RLE4, BI_RLE8, BI_PNG or BI_JPEG as the compression type.

GDI allocates memory for managing a DIB section, which includes the bitmap information header, bit masks, and the color table. These data are maintained by GDI and not directly accessible to the application. GDI certainly reserves an entry in the GDI handle table, which links to GDI's internal data structure of the DIB section. There is a one-to-one correspondence between a GDI object handle and an entry in the GDI handle table. This is similar to a DDB, but different from a DIB, which is not managed by GDI.

If we're not creating the DIB section off a memory-mapped file object, GDI allocates the storage space for the pixel array from the application's virtual memory, and returns its pointer to the caller. Note the difference between pixel array memory allocation for a DDB and a DIB section. A DDB's pixel array is allocated from GDI's heap in Windows 95-based systems, and from kernel mode paged-pool in Windows NT-based systems. They use shared-system resources with limited capacity, and an application does not have direct access to the pixel array. In contrast, a DIB section's pixel array is allocated from the current application process's virtual memory space, which is only limited by an application's virtual memory space and free hard disk space, and it can be directly accessed by the application program. The value of the newly allocated pixel array is undefined, just like an uninitialized DDB.

We should also note that the allocation is from the virtual memory address space, not from the system heap. Although the system heap is allocated from the virtual memory address space, it provides a mechanism of suballocation that benefits large amounts of small objects. Allocation from virtual memory space needs to be done in multiples of memory-page size, which is 4 KB on the Intel CPU. For DIB sections, GDI is seen to allocate in 64-KB chunks of memory.

If a valid file-mapping object handle is provided, the dwOffset parameter must be a multiple of the size of a DWORD. With the BITMAPINFO structure, GDI can calculate the size of the pixel array. With the file-mapping object handle, the offset, and a length, GDI can call the MapViewOfFile function, or its counterpart, to map a block of data in the file to the application's virtual memory space.

If the data in the file is a valid pixel array, the DIB section is fully initialized without allocating memory for the pixel array and making a potentially costly data copy. Now you would think that you could create a DIB section off a memory-mapped BMP file. Too bad it's not supported, because CreateDIBSection requires a DWORD-aligned pointer to a memory-mapped-file region. The BITMAPFILEHEADER structure is 14 bytes long, while the BITMAPINFO structure is always a multiple of DWORD; the pixel array in a BMP file is always not DWORD aligned.

If the data in the memory-mapped file is not providing a valid pixel array, the last two parameters only provide an alternative way of managing memory. Why does Microsoft even want to provide such a feature? A disk file eventually backs up every byte of memory. If it's not an application-provided file, it's the system paging swap file. By providing a file-mapping object to CreateDIBSection, an application can control where the file is stored and how it could be shared among multiple processes. For example, your system swap file is on hard disk C, which only has 100 MB free space; an image editor editing a full-page 600-dpi 32-bpp image needs a 128-MB DIB section. If the editor is smart enough to figure out that hard disk D does have 500 MB free disk space, it can create a memory-mapped file on Disk D, and tell CreateDIBSection to use it. Now the smart editor can handle four such huge images.

A Class for the DIB Section

The DIB section deserves a class for wrapping it up. Luckily, we can share lots of code with the KDIB class and the KDDDB class. Actually, our DIB section class is derived from both of them. [Listing 10-9](#) shows the DIB section class.

Listing 10-9 DIB Section Class

```
class KDIBSection : public KDIB, public KDDDB
{
public:
    KDIBSection()
    {
    }

    virtual ~KDIBSection()
    {
    }

    BOOL CreateDIBSection(HDC hDC, CONST BITMAPINFO * pBMI,
        UINT iUsage, HANDLE hSection, DWORD dwOffset);
    UINT GetColorTable(void);
    UINT SetColorTable(void);

    void DecodeDIBSectionFormat(TCHAR desp[]);
};

void KDIBSection::DecodeDIBSectionFormat(TCHAR desp[])
```

```
{  
DIBSection dibsec;  
  
if ( GetObject(m_hBitmap, sizeof(DIBSECTION), & dibsec) )  
{  
    KDIB::DecodeDIBFormat(desp);  
    _tcscat(desp, " ");  
    DecodeDDB(GetBitmap(), desp + _tcslen(desp));  
}  
else  
    _tcscpy(desp, _T("Invalid DIB Section"));  
}  
  
BOOL KDIBSection::CreateDIBSection(HDC hDC, CONST BITMAPINFO * pBMI,  
        UINT iUsage, HANDLE hSection, DWORD dwOffset)  
{  
    PVOID pBits = NULL;  
  
    HBITMAP hBmp = ::CreateDIBSection(hDC, pBMI, iUsage, & pBits,  
        hSection, dwOffset);  
  
    if ( hBmp )  
    {  
        ReleaseDDB(); // free the previous object  
        ReleaseDIB();  
  
        m_hBitmap = hBmp;  
  
        int nColor = GetDIBColorCount(pBMI->bmiHeader);  
        int nSize = sizeof(BITMAPINFOHEADER) + sizeof(RGBQUAD) * nColor;  
  
        BITMAPINFO * pDIB = (BITMAPINFO *) new BYTE[nSize];  
  
        if ( pDIB==NULL )  
            return FALSE;  
  
        memcpy(pDIB, pBMI, nSize); // copy header and color table  
        AttachDIB(pDIB, (PBYTE) pBits, DIB_BMI_NEEDFREE);  
        GetColorTable(); // get color table from DIB section  
        return TRUE;  
    }  
    else  
        return FALSE;  
}
```

The KDIBSection class does not have any member variables, because it's using the KDDB and KDIB class member variables. A KDIBSection instance is both a KDDB class instance and a KDIB class instance. So when a DIB section instance is properly initialized, you can use KDDB class methods to draw into it using GDI drawing functions and use KDIB class methods to directly access its pixel array. The real code in this class is the

KDIBSection::CreateDIBSection functions. It calls the GDI function CreateDIBSection to create a DIB section. If the call is successful, a copy of the BITMAPINFO structure is made and filled with a new color table; the KDIB::AttachDIB function is then called to initialize the KDIB part of the member variables. Note that we only need to free the new BITMAPINFO structure; the destructor of KDDB class will call DeleteObject on the DIB section handle, which frees the pixel array allocated by the GDI.

GetObjectType/GetObject on DIB Sections

Both DDBs and DIB sections are GDI objects, but they are certainly different. If you only have a GDI object handle, it's not so easy to tell the difference between a DDB and a DIB section. For a DIB section, GetObjectType returns the same OBJ_BITMAP(7), GetObject(hBitmap, 0, NULL) always returns sizeof(BITMAP), and Get Object(hBit map, sizeof(BITMAP), & bmp) always succeeds and fills a BITMAP structure.

There are two ways to tell a DIB section from a DDB. First, the BITMAP structure returned by GetObject has a valid pointer to the pixel array. Recall that for a DDB, its pixel array address is not exposed to the application program, so the bmBits field is always NULL. For a DIB section, this value is the same value returned by CreateDIBSection through the ppvBits parameter. Second, GetObject can return a DIBSection structure if the cbBuffer parameter is sizeof(DIBSECTION) and lpvObjects points to a buffer big enough to hold a DIBSection structure.

The DIBSection structure is what GDI would like the application to know about a DIB section. Its first field is a BITMAP structure, which describes the DDB view of a DIB section. The second field is a BITMAPINFOHEADER structure, which describes DIB dimension and color format. Note that this structure could also be BITMAPV4HEADER or BITMAPV5HEADER. GDI should have defined DIBSECTIONV4 and DIBSECTIONV5 structures. The dsBitFields field is an array of three bit masks used for 16-bpp and 32-bpp modes. It will be filled in for both BI_RGB and BI_BITFIELDS compression modes. This provides another way of knowing the current pixel format if running in 16-bpp display mode. If you create a 16-bpp BI_RGB DIB section, querying for the DIBSection structure and checking the dsBitFields field could determine whether you're running in 5-5-5 mode or 5-6-5 mode. The last two fields are for creating a DIB section from a memory-mapped file object. They are the same parameters as passed to CreateDIBSection.

GetDIBColorTable/SetDIBColorTable

A DIB section is not a real DIB, because an application does not have direct access to the color table in the same way as it controls a DIB color table. Instead, a DIB section's color table is covered up by GDI, and the application can only access it through a well-controlled GetDIBColorTable/SetDIBColorTable function pair.

You may ask, why does the application even care to access a DIB section's color table, considering it actually provides the color table when creating the DIB section through CreateDIBSection call? There are at least two valid reasons. First, if the iUsage parameter to CreateDIBSection is DIB_PAL_COLORS, the application only has logical palette relative palette indexes, not a true RGB color table. If the application wants to save the DIB section to a BMP file, it needs the actual RGB color table. Second, quite a few image algorithms can be implemented by manipulating a color table for palette-based images. For example, adjust the hue, lightness, and saturation of a bitmap, or convert a bitmap to grayscale. To do that, an application needs to manipulate an RGB color table and report the changes.

The GetDIBColorTable function returns the RGB color table of a DIB section. Its first parameter is a memory device context, into which the DIB section should be selected. The uStartIndex and cEntries parameters tell the start table entry and number of entries to copy. The pColors parameter points to a buffer to receive the color table as an array of RGBQUAD structure. The GetDIBColorTable function copies the DIB section's color table to the buffer. The number of entries copied is returned as the function result; 0 is also for failure.

The SetDIBColorTable function sets a DIB section's color table from an application-provided color table. It accepts the same set of parameters and also returns the number of entries copied.

Here are the corresponding functions for the KDIBSection class. Note that we're using the same memory DC used in drawing a bitmap, and the KDIB::m_pRGB QUAD field is used to store the DIB section color table.

```
// Copy from DIB section color table to DIB color table
UINT KDIBSection::GetColorTable(void)
{
    int width, height;
    if ( (GetDepth()>8) || ! Prepare(width, height) ) // create memory DC
        return 0;

    return GetDIBColorTable(m_hMemDC, 0, m_nClrUsed, m_pRGBQUAD);
}

// Copy from DIB's color table to DIB section's color table
UINT KDIBSection::SetColorTable(void)
{
    int width, height;

    if ( (GetDepth()>8) || ! Prepare(width, height) ) // create memory DC
        return 0;

    return SetDIBColorTable(m_hMemDC, 0, m_nClrUsed, m_pRGBQUAD);
}
```

Using DIB Sections: Device-Independent Rendering

Compared with a DIB and a DDB, a DIB section offers several advantages in several areas.

- **Device-independent rendering using GDI.** GDI would not help you much in generating a DIB. For DDBs, the only supported color format is the one compatible with the current display mode. If a graphics application wants to implement 24-bpp rendering in 8-bpp display mode using GDI, a DIB section is the only solution.
- **Combining GDI rendering and direct pixel array manipulation.** Only DIB sections support GDI rendering and direct pixel manipulation by the application at the same time. Without a DIB section, you have to create a DIB and a DDB, and transfer pixels between them using GetDIBits and SetDIBits.
- **Flexible memory management.** Compared with DDBs, which allocate from system memory, DIB section memory is allocated from the application's virtual memory space, or memory-mapped file. The size of DDBs is only limited by virtual memory space and free disk space. For example, if you have enough free disk space, you will be able to create an 8192-by-8192, 32-bpp DIB section, which is 256 MB in size. Such a DDB is not possible, because the NT-based system limits DDB to be 48 MB, and the 95-based system limits DDB to be 16 MB. DIB sections also allow creating more big bitmaps at the same time. Memory management for DIBs is certainly more flexible. For example, DIBs can live in the read-only resource

section of an executable file.

As an example of implementing device-independent rendering, let's try screen capturing again. This time, we want to always capture the screen to a 24-bpp DIB. This is certainly doable through capturing the contents of a window to a DDB and then converting to a 24-bpp DIB. But we want to do something extra: add a 3D frame to the image, and a title on the image. If using a DDB, this would be very hard to do if you're running in 8-bpp display mode. A 3D frame involves gradually changing colors that are hard to represent in an 8-bpp DDB. If you want to add the frame in the 24-bpp DIB, GDI is not going to help you. With DIB section, only one 24-bpp DIB section is needed. GDI will do all the rendering, and after that you can save the image to a BMP file.

[Listing 10-10](#) shows the SaveWindow routine and a routine to draw a 3D frame.

Listing 10-10 Window Capturing Using a DIB Section, with Framing

```
BOOL SaveWindow(HWND hWnd, bool bClient, int nFrame, COLORREF crFrame)
{
    RECT wnd;

    if ( bClient )
    {
        if ( !GetClientRect(hWnd, & wnd) )
            return FALSE;
    }
    else
    {
        if ( !GetWindowRect(hWnd, & wnd) )
            return FALSE;
    }

    KBitmapInfo bmi;
    KDIBSection dibsec;

    bmi.SetFormat(wnd.right - wnd.left + nFrame * 2,
                  wnd.bottom - wnd.top + nFrame * 2, 24, BI_RGB);

    if ( dibsec.CreateDIBSection(NULL, bmi.GetBMI(), DIB_RGB_COLORS,
                                NULL, NULL) )
    {
        int width, height;
        dibsec.Prepare(width, height); // creates memDC, select dibsec

        if ( nFrame )
        {
            Frame3D(dibsec.m_hMemDC, nFrame, crFrame, 0, 0,
                     width, height);

            TCHAR Title[128];
            GetWindowText(hWnd, Title, sizeof(Title)/sizeof(Title[0]));
        }
    }
}
```

```
SetBkMode(dibsec.m_hMemDC, TRANSPARENT);
SetTextColor(dibsec.m_hMemDC, RGB(0xFF, 0xFF, 0xFF));
TextOut(dibsec.m_hMemDC, nFrame, (nFrame-20)/2,
        Title, _tcslen(Title));
}

HDC hDC;
if ( bClient )
    hDC = GetDC(hWnd);
else
    hDC = GetWindowDC(hWnd);

// Bitblt from screen to DIB section
BitBlt(dibsec.m_hMemDC, nFrame, nFrame, width - nFrame * 2,
        height - nFrame * 2, hDC, 0, 0, SRCCOPY);
ReleaseDC(hWnd, hDC);

return dibsec.SaveFile(NULL);
}
return FALSE;
}

void Frame3D(HDC hDC, int nFrame, COLORREF crFrame, int left, int top,
             int right, int bottom)
{
    int red  = GetRValue(crFrame);
    int green = GetGValue(crFrame);
    int blue  = GetBValue(crFrame);
    RECT rect = { left, top, right, bottom };

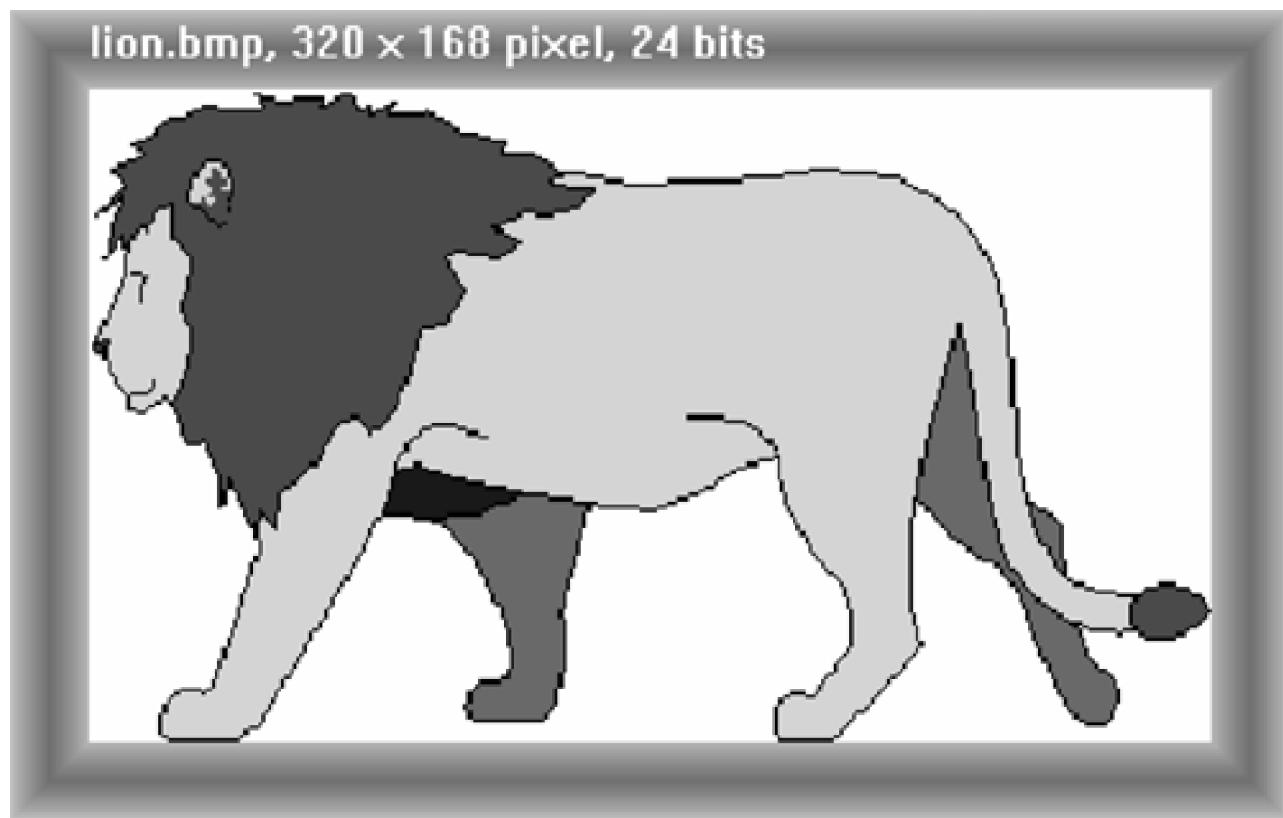
    for (int i=0; i<nFrame; i++)
    {
        HBRUSH hBrush = CreateSolidBrush(RGB(red, green, blue));
        FrameRect(hDC, & rect, hBrush); // one pixel
        DeleteObject(hBrush);

        if ( i<nFrame/2 )      // first half
        {
            red  = red  * 19/20; // darker
            green = green * 19/20;
            blue  = blue  * 19/20;
        }
        else                  // second half
            // lighter
        red  = red  * 19/18; if ( red>255 ) red  = 255;
        green = green * 19/18; if ( green>255 ) green= 255;
        blue  = blue  * 19/18; if ( blue>255 ) blue = 255;
    }
}
```

```
InflateRect(&rect, -1, -1); // smaller  
}  
}
```

The SaveWindow function accepts four parameters: hWnd is the window handle, bClient tells whether the whole window or its client area should be captured, nFrame is the width of frame to add, and crFrame determines the frame color. The function sets up a BITMAPINFO structure for a 24-bpp DIB section using the KBitmapInfo class, a simple wrapper class for initializing the BITMAPINFO structure. An instance of the KDIBSection class is created on the stack. After creating the DIB section, a memory DC is created and the DIB section is selected into it. Now we are ready to use GDI to draw the picture using the Frame3D routine, add a title to it using the window's title text, and finally copy the pixels from screen DC to the center of the DIB section. Finally, the code uses the KDIB::SaveFile method to save the DIB section to a BMP file. The 3D frame is drawn using multiple single-pixel rectangles with colors gradually changing from the original color to darker colors and then to lighter colors, creating an impression of a rounded frame. [Figure 10-7](#) shows a sample output.

Figure 10-7. Captured client area with a frame.



We've demonstrated how to use GDI to draw into a DIB section. The other part of the game, directly accessing the pixel array, is the same as for a DIB. We will cover the details in the next chapter.

Using DIB Sections: High-Resolution Rendering

Allocating DIB section storage from user mode address space allows for a much bigger image size than would be possible for DDBs. Supporting an application-specified memory-mapped file handle in creating DIB sections makes it easier to control disk/memory space allocation. These two features are particularly useful for image-processing software and the software-based raster image processor (RIP). A raster image processor takes a document written in page description language and converts it to a high-resolution raster image, which can then be sent to

high-resolution printers after halftoning. For example, Ghostscript can be seen as software RIP. It takes a PostScript document and renders it into raster images of different resolution.

We're going to develop a simple software raster image processor using a DIB section. Of course, we can't handle the complexity of PostScript documents, but with GDI's help, we can use something expressive, useful and easily available—Windows enhanced metafiles (EMF). Our plan is to create a memory-mapped file to be the basis for a high-resolution DIB section, and then render the EMF file into the DIB section. Once rendering is done, deleting the DIB section and closing the file gives us a high-resolution raster image of the EMF file.

The tricky thing here is that the CreateDIBSection function requires the pixel array offset parameter to be a multiple of the DWORD size. BMP files do not meet this requirement, because their pixel array never starts on the DWORD boundary. To solve this problem, we choose to use the 24-bpp Targa image file format, which has a very simple and adjustable-length header and supports an uncompressed 24-bpp RGB pixel array.

[Listing 10-11](#) shows the KTarga24 class, which implements a memory-mapped DIB section using the 24-bpp Targa image format.

Listing 10-11 Memory-Mapped DIB Section

```
class KTarga24 : public KDIBSection
{
    #pragma pack(push,1)

    typedef struct {
        BYTE IDLength; // 00: length of Identifier string
        BYTE CoMapType; // 01 0 = no map
        BYTE ImgType; // 02 2 = TGA_RGB
        WORD Index; // 03 index of first color map entry
        WORD Length; // 05 number of entries in color map
        BYTE CoSize; // 07 size of color map entry
        WORD X_Org; // 08 0
        WORD Y_Org; // 0A 0
        WORD Width; // 0C width
        WORD Height; // 0E height
        BYTE PixelSize; // 10 pixel size
        BYTE AttBits; // 11 0
        char ID[14]; // 12 make sure ImageHeader is DWORD aligned
    } ImageHeader;
    #pragma pack(pop)

    HANDLE m_hFile;
    HANDLE m_hFileMapping;

public:
    KTarga24()
    {
        m_hFile = INVALID_HANDLE_VALUE;
        m_hFileMapping = INVALID_HANDLE_VALUE;
    }
}
```

```
~KTarga24()
{
    ReleaseDDB();
    ReleaseDIB();

    if ( m_hFileMapping!=INVALID_HANDLE_VALUE )
        CloseHandle(m_hFileMapping);

    if ( m_hFile != INVALID_HANDLE_VALUE )
        CloseHandle(m_hFile);
}

BOOL Create(int width, int height, const TCHAR * filename);

};

BOOL KTarga24::Create(int width, int height, const TCHAR * pFileName)
{
    if ( width & 3 ) // avoid compatibility problem with TGA
        return FALSE;

    ImageHeader tgaheader;

    memset(&tgaheader, 0, sizeof(tgaheader));
    tgaheader.IDLength = sizeof(tgaheader.ID);
    tgaheader.ImgType = 2;
    tgaheader.Width = width;
    tgaheader.Height = height;
    tgaheader.PixelSize = 24;
    strcpy(tgaheader.ID, "BitmapShop");

    m_hFile = CreateFile(pFileName, GENERIC_WRITE | GENERIC_READ,
                         FILE_SHARE_READ | FILE_SHARE_WRITE,
                         NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);

    if ( m_hFile==INVALID_HANDLE_VALUE )
        return FALSE;

    int imagesize = (width*3+3)/4*4 * height;

    m_hFileMapping = CreateFileMapping(m_hFile, NULL, PAGE_READWRITE,
                                       0, sizeof(tgaheader) + imagesize, NULL);
    if ( m_hFileMapping==INVALID_HANDLE_VALUE )
        return FALSE;

    DWORD dwWritten = NULL;
    WriteFile(m_hFile, &tgaheader, sizeof(tgaheader), &dwWritten, NULL);
    SetFilePointer(m_hFile, sizeof(tgaheader) + imagesize, 0, FILE_BEGIN);
    SetEndOfFile(m_hFile);
```

```
KBitmapInfo bmi;  
  
bmi.SetFormat(width, height, 24, BI_RGB);  
  
return CreateDIBSection(NULL, bmi.GetBMI(), DIB_RGB_COLORS,  
    m_hFileMapping, sizeof(tgaheader));  
}
```

The KTarga24 class is derived from the KDIBSection class. It has two new member variables, one for a file handle and one for a file-mapping handle. The Create method is its main method. It accepts image width, height, and a file name. The routine creates a file and a file-mapping object, writes a 32-byte Targa image header, and creates a DIB section using the file-mapping object. Because the header is 32 bytes long, the offset to the pixel array is 32, a multiple of the DWORD size. Note here that the file needs to be created with shared read and write access, and the file mapping also needs to have read and write access; otherwise DIB section creation or rendering into the file will fail. The file and file-mapping objects are closed in the destructor after the DIB section GDI object is deleted (in ReleaseDDB).

Enhanced metafile (EMF) is the topic of [Chapter 16](#). For the moment, just remember that an EMF file is a recorded GDI command sequence, which can be easily manipulated and played back. Here is the routine that uses the KTarga24 class to render an EMF file.

```
BOOL RenderEMF(HENHMETAFILE hemf, int width, int height,  
    const TCHAR * tgaFileName)  
{  
KTarga24 targa;  
  
int w = (width+3)/4*4; // make sure multiple of 4  
if ( targa.Create(w, height, tgaFileName) )  
{  
    targa.Prepare(w, height);  
  
    BitBlt(targa.m_hMemDC, 0, 0, width, height, NULL, 0, 0,  
        WHITENESS); // clear DIB section  
  
    RECT rect = { 0, 0, width, height };  
  
    return PlayEnhMetaFile(targa.m_hMemDC, hemf, &rect);  
}  
return FALSE;  
}
```

The RenderEMF routine accepts an EMF handle, the width and height of the image to render, and the image file name. It creates an instance of the KTarga24 class on the stack, initializes it, clears the DIB section with a white background, and calls PlayEnhMetaFile to stretch the EMF object into the DIB section.

Some user interface code is also needed to select the input EMF file, output Targa file name, and rendering scale. EMF files are rendered proportionally using the same scale along x-and y-axes.

As a test run, we rendered a 600-KB EMF file with complicated line art, using 900% scaling. The original pixel size of the EMF is 625 by 777 pixels; scaling 900% makes it 5625 by 6993 pixels. The size of the 24-bpp DIB section is 112 MB. The size of this job is close to that of rendering a full letter-size document at 600 dpi.

10.8 SUMMARY

In this chapter we covered three types of bitmaps supported by GDI: device-independent bitmaps, device-dependent bitmaps, and DIB sections. We've focused on the creation, conversion, displaying, and simple uses of the bitmap types, as well as the differences between these bitmaps.

The bitmap is such an important topic that we have to divide it into three chapters. [Chapter 11](#) will continue the more advanced and exciting aspects of bitmap usage: raster operations, transparency, direct pixel manipulation, and alpha blending. [Chapter 12](#) will focus on image processing using direct pixel access. Also, [Chapter 17](#) covers how to decode and print JPEG images.

Further Reading

The topic of this chapter is maintaining the three bitmap formats supported by the Windows operating system and using GDI functions to do simple display and manipulation. So the topic itself is very Windows-specific. If you are interested in image formats, there are several good resources and even on-line resources.

Encyclopedia of Graphics File Formats, by James D. Murray and William Vanryper, provides a good overview of graphics file formats and gives details about 100 different file formats.

Computer-graphics-related FAQs (Frequently Asked Questions) can be found at [“www.faqs.org/faqs/graphics/”](http://www.faqs.org/faqs/graphics/). For example, [“www.faqs.org/faqs/graphics/fileformats-faqs/”](http://www.faqs.org/faqs/graphics/fileformats-faqs/) contains a four-part FAQ on the computer graphics image format.

Sample Programs

Two sample programs are developed for this book. But more importantly, quite a few very useful bitmap handling classes are created for these two programs, which will be extended and shared by other chapters on bitmaps (see [Table 10-6](#)).

Table 10-6. Sample Programs for Chapter 10

Directory	Description
Samples\Chapt_10\Bitmaps	Demonstrates loading, saving BMP files, screen capture, various ways to display bitmaps, device-dependent bitmaps, bitmap check marks, bitmap menus, bitmap backgrounds, DIB sections, device independent rendering using DIB sections, and decoding bitmaps into hex format.
Samples\Chapt_10\Scrambler	Demonstrates using StretchBlt to scramble the screen display.

Chapter 11. Advanced Bitmap Graphics

Bitmaps are such an important area of Windows graphics programming that we can't cover the topic in a single chapter. The last chapter basically covered the three bitmap types supported by GDI: device-independent bitmaps, device-dependent bitmaps, and a DIB section. We discussed fundamentals such as various ways of displaying bitmaps, using them in user interface design, and even implementing software high-resolution rendering.

There is much more about bitmaps we have not covered yet. This chapter will cover raster operations, transparency, and alpha blending. We will cover image processing using Window bitmaps in [Chapter 12](#) and palettes in [Chapter 13](#).

11.1 TERNARY RASTER OPERATIONS

When using GDI to draw a line or fill an area, GDI uses binary raster operations to combine the pen or brush pixel with the original destination pixel to form the new destination pixel. Sixteen binary raster operations are supported through the SetROP2 and Get ROP2 function pair.

For images, you would expect similar raster operations to create a variety of special effects. Now we have a new player in the game: pixels from a source image. Counting them together, we have three players: the pen or brush pixel, the destination pixel, and the source image pixel. Raster operations combining them together are naturally called ternary raster operations.

GDI supports only bitwise logical operations, not arithmetic operations or other fancy functions. In a bitwise logical operation, the logical operations are applied to each bit in a pixel independently of each other, and only Boolean logical operations are used. The Boolean logical operations used are AND (&), OR (|), NOT (~), and XOR (^). Given these restrictions, there are a total of 256 ternary raster operations, which is $2^8(2^3)$ or 2^8 .

Raster Operation Encoding

Only a single byte is really needed to encode 256 raster operations. Given that we can treat each bit independently, the encoding mechanism is quite simple. Let P be the pen or brush bit, S be the source image bit, and D be the destination bit. If the operation result is always the same as P, its code is 0xF0; if the operation result is always the same as S, its code is 0xCC; if the operation result is always the same as D, its code is 0xAA. Here are the C/C++ definitions for these raster operation codes.

```
const BYTE rop_P = 0xF0; // 1 1 1 1 0 0 0 0
const BYTE rop_S = 0xCC; // 1 1 0 0 1 1 0 0
const BYTE rop_D = 0xAA; // 1 0 1 0 1 0 1 0
```

All other raster operations can be defined based on these three constants using Boolean operations. For example, if you want to define a raster operation whose result is the logical AND of S and P, just calculate $\text{rop_S} \& \text{rop_P}$; you get 0xC0. If you want a raster operation which returns P if S is 1, and D otherwise, calculate $(\text{rop_S} \& \text{rop_P}) | (\neg \text{rop_S} \& \text{rop_D})$; you get 0xE2.

For each raster operation, there is at least one corresponding Boolean algebra formula using P, S, and D which exactly defines the raster operation. There may be more than one formula for each raster operation, but they are all logically equivalent. Traditionally, they are written in postfix notation, which puts the operator immediately to the right of its operands. The benefit of postfix notation is that parentheses are not needed, and it can easily be evaluated by a computer program. The original designer of these raster operations could have been a fan of Forth (a stack-based extensible language without type-checking), Postscript, or HP calculators, which all use postfix notation.

In the postfix notation for raster operations, P, S, and D are the operands; a, o, n, and x, which stand for AND, OR, NOT, and XOR, are the operators. For example, the formula for raster operator 0xE2 is “DSPDxax.” Translating it into infix notation gives $D^a(S&(P^oD))$. A more intuitive equivalent formula would be $(S&P)(\neg S&D)$, or “SPaSnD&o.” Why is the former favored over the latter? Two reasons. Most computer CPUs have an XOR instruction, which is as

fast as an AND operation. The first formula uses three operations and the second one uses four operations. The first formula needs only one extra register to evaluate, while the second formula needs two. Here is the pseudocode for their implementation, assuming a 32-bpp frame buffer:

```
// implementing 0xE2 DSPDxax using D^(S&(P^D))
mov eax, P      // P
xor eax, D      // p^D
and eax, S      // S&(P^D)
xor eax, D      // D^(S&(P^D))
mov D, eax      // write result back

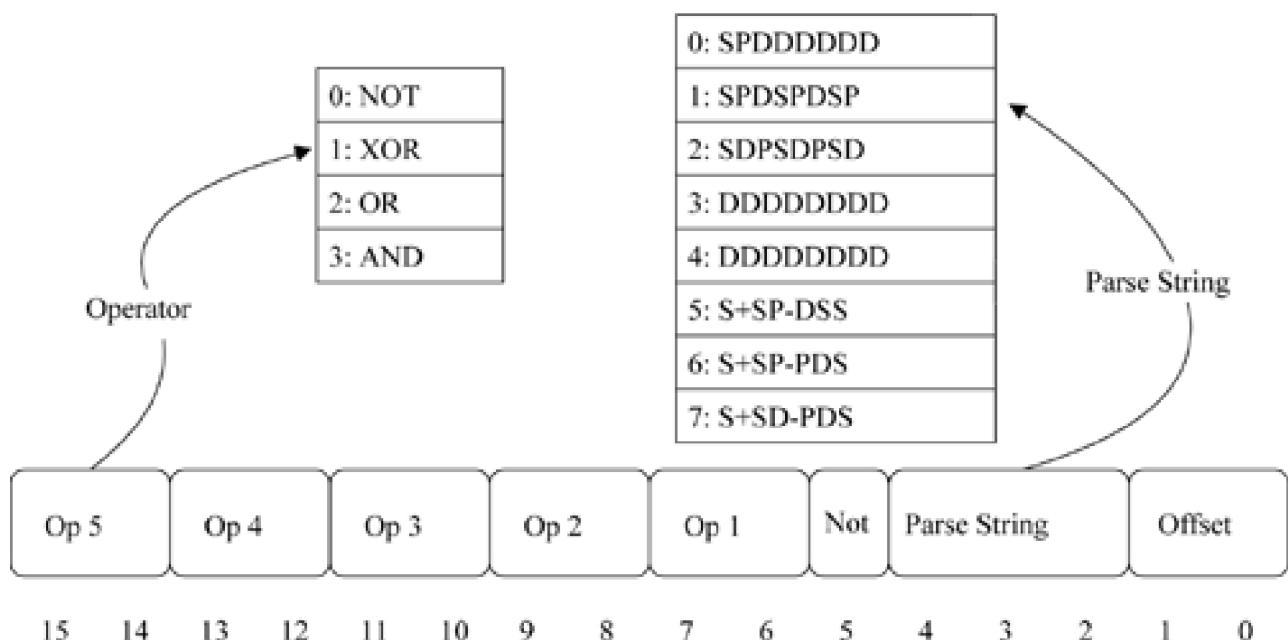
// implementing 0xE2 SPaSnD&o using (S&P)|(~S&D)
mov eax, S      // S
and eax, P      // S&P
mov ebx, S      // S
not ebx         // ~S
and ebx, D      // ~S&D
or  eax, ebx    // (S&P) | (~S&D)
mov D, eax      // write result back
```

Raster operations in GDI are actually encoded using 32-bit DWORDs, instead of just a simple byte from 0 to 255. The higher word is the one of the 256 single-byte raster operation codes we talked about above; the lower word is an encoding of the formula defining the raster operation. In the original design, only the lower word is used to specify a raster operation; the higher word is an add-on for hardware bitblter. The oldest implementations of these raster operations were hand-coded in optimized assembly code, so a generic algorithm which takes a formula is favored over having a specific case for each of the 256 raster operations and a big jump table. Later implementations use an automatic raster-operation routine generator, which would not complain so much about writing 256 routines as its human predecessor did.

The encoding mechanism for the lower word of ternary raster operations is a piece of art, reflecting the extra thought programmers had to put behind every line of machine code and every bit of computer memory in the early days of computer programming. A raster operation formula uses 7 different symbols, and it can be as long as 12 symbols long. A simple encoding mechanism will need $\log_2(7^{12})$, or 33.7 bits of information, but the designer had to put them into a 16-bit WORD. A raster operation formula is divided into two parts: the operators and the operands, just like the operator and operand stacks used in some stack machines. The operators are encoded using the higher 11 bits; the remaining 5 bits are for the operands. Of the 11 bits, 10 bits are used to encode five logical operations (two bits each, 0 for NOT, 1 for XOR, 2 for OR, and 3 for AND); the last bit indicates whether the last operation is a NOT operation. Encoding the operand string using 5 bits is really tough, but the programmers were smart enough to find repetitive patterns in those operand strings. They are called parse strings. Eight of those parse strings were identified, which need 3 bits to encode. The last two bits are used to shift parser strings to the right place.

[Figure 11-1](#) illustrates the encoding used in the lower word of GDI ternary operations.

Figure 11-1. Lower word of raster operation: encoding raster formula.



An example is definitely needed. Let's take the 0xE2 rop; its full raster operation code is 0x00E20746, so its lower word is 0x0746. So Op5 is NOT, Op4 is NOT, Op3 is XOR, Op2 is AND, Op1 is XOR, an extra NOT is not needed, the parse string index is 1, and the offset is 2. The actual parse string to use is SPDSPDSP shifted by two symbols, which is DSPDSPSP. We have five operators, but only three of them are binary operators; the last two are unary operators. So only four operands are needed. Now the parse string is truncated to DSPD; applying operators starting from the last symbol gives us “DSPDxaxnn”, which can be simplified to be “DSPDxax”, exactly the string used in the GDI definition of the raster operation.

The “+” and “-” signs in the parse string are called special operands. Of the total 256 raster operations, 16 of them can't be expressed using the single accumulator mechanism, where only one calculated value is kept. For these 16 raster operations, an intermediate result needs to be pushed on the stack, and popped back when needed. The special operands always appear in pairs. On the first occurrence, the current result is pushed, and the next operand is loaded; on the second occurrence, a binary logical operation is performed between the current result and the pushed value. Internally, the two special symbols are represented using the same bits (0x00), to ensure that a parse string can be stored in a 16-bit WORD. In [Figure 11-1](#), the “-” sign shows a push, and “+” indicates a pop. Note the parse string is read backwards.

Clearly, this nice antique design saves memory and lines of assembly code at the cost of performance and clarity. Newer GDI implementation does not actually use these slow mechanisms. So normally, it's safe to disregard the lower word of ternary raster operations. But it's hard to say whether a certain graphic device driver will check for the exact match of every bit of the 32-bit raster operation code. So to be safe, when using a new raster operation, check the full raster operation code and use it.

A ternary raster operation uses only 24 bits of a 32-bit ROP code. The most significant 8 bits of a ROP code are normally all zeros. Windows 98 and Windows 2000 introduce two new flags that control the operation of bitmap transfer: CAPTUREBLT and NOMIRRORBITMAP.

Flag CAPTUREBLT (0x40000000) is for a layered window, a window that has its own rendering surface that can be alpha blended with other windows. When CAPTUREBLT is used, pixels for all windows that are layered on top of the current window are included in the result image. By default, the image contains only the current window's image.

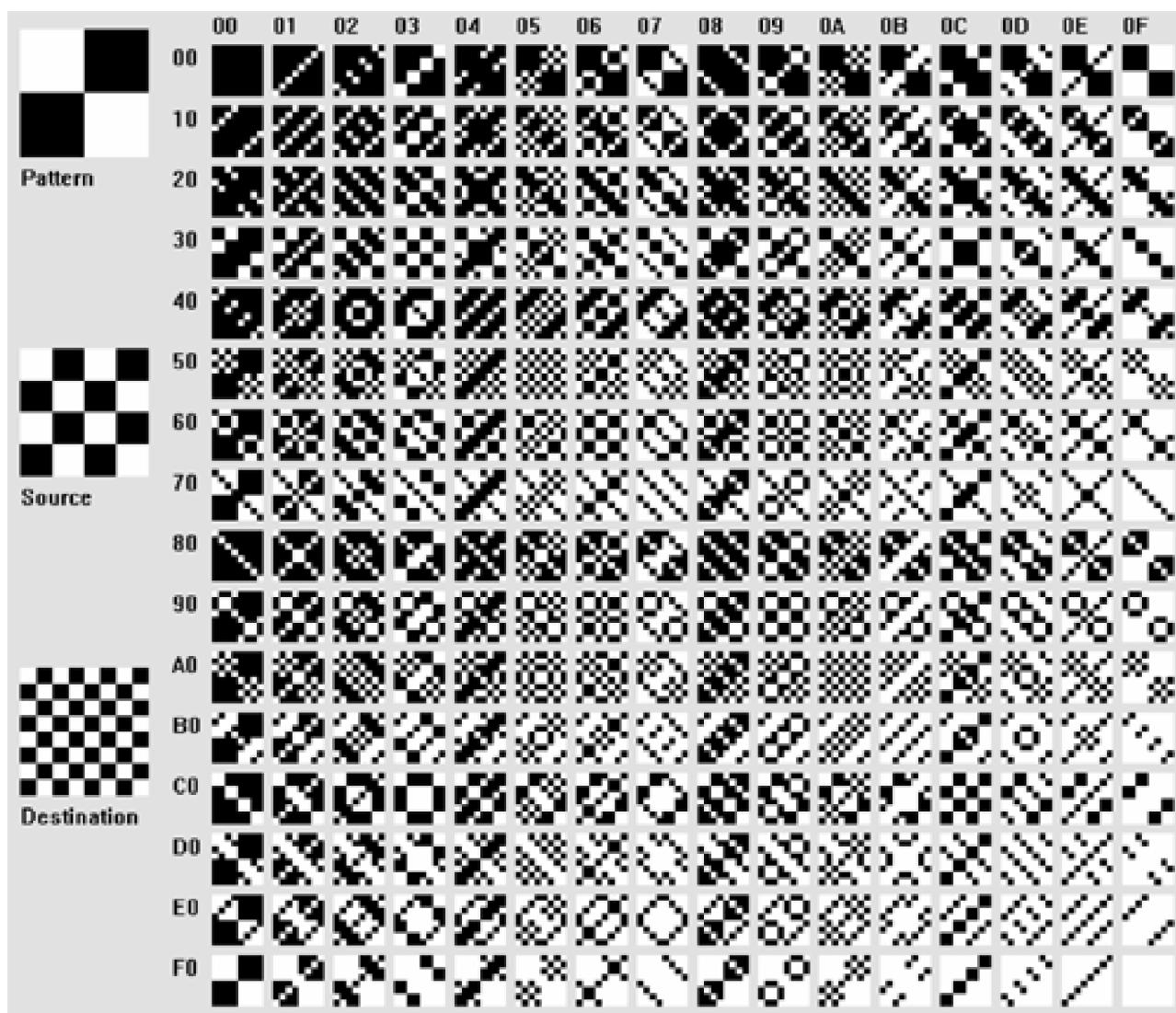
Flag NOMIRRORBITMAP (0x80000000) prevents bitmaps from being mirrored vertically and horizontally due to axes directions or differences between source and destination rectangles.

Ternary Raster Operation Chart

Although algebraic formulae for raster operations are accurate and precise, a visual representation of raster operations will certainly help us understand all their variations. If we can define bitmap patterns to represent the three variables used, we will be able to generate the resulting bitmap by applying the algebraic formula, and visualize all the operations.

[Figure 11-2](#) shows a simple ternary-raster operation chart. The chart is generated by applying raster operations to the three bitmaps on the left.

Figure 11-2. Ternary raster operation chart.



The 8-by-8 pixel bitmap for pattern is generated by applying 0xF0 in both X and Y directions, the source bitmap is generated using 0xCC, and the destination bitmap is generated using 0xAA. Note that white is logical 1, and black is logical 0 here.

The 256 small bitmaps represent all possible raster operations. They are generated by creating a pattern brush using the pattern bitmap, and then combining source and destination bitmaps using the raster operation when the

pattern brush is selected.

Here is the code that generates the raster operation chart. Note that 8-by-8 bitmaps are used to make sure the pattern brush also works on Windows 95-based systems. The resulting bitmap is generated in a memory DC and then is stretched to screen DC, because the pattern brush does not scale during stretching.

```
const WORD Bit_Pattern [] = { 0xF0, 0xF0, 0xF0, 0xF0,
                               0x0F, 0x0F, 0x0F, 0x0F };
const WORD Bit_Source [] = { 0xCC, 0xCC, 0x33, 0x33,
                            0xCC, 0xCC, 0x33, 0x33 };

const WORD Bit_Destination[] = { 0xAA, 0x55, 0xAA, 0x55,
                                 0xAA, 0x55, 0xAA, 0x55 };

void Rop3Chart(HDC hDC)
{
    HBITMAP Pbmp = CreateBitmap(8, 8, 1, 1, Bit_Pattern);
    HBITMAP Sbmp = CreateBitmap(8, 8, 1, 1, Bit_Source);
    HBITMAP Dbmp = CreateBitmap(8, 8, 1, 1, Bit_Destination);
    HBITMAP Rbmp = CreateBitmap(8, 8, 1, 1, NULL);

    HBRUSH Pat = CreatePatternBrush(Pbmp); // pattern brush
    HDC Src = CreateCompatibleDC(hDC); // memdc for source
    HDC Dst = CreateCompatibleDC(hDC); // memdc for destination
    HDC Rst = CreateCompatibleDC(hDC); // memdc for result
    SelectObject(Src, Sbmp);
    SelectObject(Dst, Dbmp);
    SelectObject(Rst, Pbmp);

    StretchBlt(hDC, 20, 20, 80, 80, Rst, 0, 0, 8, 8, SRCCOPY);
    StretchBlt(hDC, 20, 220, 80, 80, Src, 0, 0, 8, 8, SRCCOPY);
    StretchBlt(hDC, 20, 420, 80, 80, Dst, 0, 0, 8, 8, SRCCOPY);

    SetBkMode(hDC, TRANSPARENT);
    TextOut(hDC, 20, 105, "Pattern", 7);
    TextOut(hDC, 20, 305, "Source", 6);
    TextOut(hDC, 20, 505, "Destination", 11);

    SelectObject(Rst, Rbmp);
    SelectObject(Rst, Pat);

    for (int i=0; i<16; i++)
    {
        char mess[3];
        wsprintf(mess, "0%X", i); TextOut(hDC, 140 + i*38, 10, mess, 2);
        wsprintf(mess, "%X0", i); TextOut(hDC, 115, 30+i*38, mess, 2);
    }
}
```

```
}

for (int rop=0; rop<256; rop++)
{
    BitBlt(Rst, 0, 0, 8, 8, Dst, 0, 0, SRCCOPY);
    BitBlt(Rst, 0, 0, 8, 8, Src, 0, 0, RopDict[rop].opcode);

    StretchBlt(hdc, 140 + (rop%16)*38, 30 + (rop/16)*38, 32, 32,
               Rst, 0, 0, 8, 8, SRCCOPY);
}

DeleteObject(Src); DeleteObject(Dst);
DeleteObject(Rst); DeleteObject(Pat);
DeleteObject(Pbmp); DeleteObject(Sbmp);
DeleteObject(DBmp); DeleteObject(Rbmp);
}
```

Commonly Used Raster Operations

The 256 raster operations may sound impressive, but only dozens of them are actively being used to produce useful effects. Microsoft cares to name only 15 of them. Considering that there are 16 named binary raster operations, 15 ternary raster operations are clearly not enough, because every binary raster operation has a corresponding ternary raster operation which does not depend on S. [Table 11-1](#) lists 30 ternary raster operations that we think are useful in real GDI programming.

Compared with binary raster operations, the naming for ternary raster operations is really a mess. Binary raster operations all start with R2_, so naturally ternary raster operations should have started with R3_. But it's not so. Binary raster operation naming uses NOT for "NOT," XOR for "XOR," MERGE for "OR," and MASK for "AND." Ternary raster operations use INVERT sometimes for "NOT," sometimes for "XOR"; PAINT for "OR"; and ERASE for "AND NOT." Be extremely careful when using unfamiliar ternary raster operations. Check the exact formula to make sure it does what you want to achieve.

The raster operations in [Table 11-1](#) are arranged according to their dependency on the three variables P, S, and D. It is different from most raster operation tables that are arranged according to numerical value of operation code. Understanding the dependency helps you write better programs. For example, if a raster operation depends on D, it's not safe to use during printing. Raster operation is really designed for raster devices, on which every pixel is addressable, readable, and writeable. Some graphics devices like Postscript printers are not purely raster-based, in the sense that they do not support the full set of raster operations, especially those depending on destination pixels. If a raster operation does not depend on the source bitmap, the source bitmap does not need to be passed to BitBlt, StretchBlt, or StretchDIBits calls. Actually, stretching is not even needed when the source bitmap is not needed. GDI provides a special function for raster operations that do not use source.

Table 11-1. Ternary Raster Operations

Dependency	ROP3 Name	ROP Code	Formula	ROP2 Name
None	BLACKNESS	0x000042	0	R2_BLACK
	WHITENESS	0xFF0062	1	R2_WHITE
Pattern	PATCOPY	0xF00021	P	R2_COPYPEN
		0x0F0001	$\sim P$	R2_NOTCOPYPEN
Source	SRCCOPY	0xCC0020	S	
	NOTSRCCOPY	0x330008	$\sim S$	
Destination		0xAA0029	D	R2_NOP
	DSTINVERT	0x550009	$\sim D$	
Pattern & Source	MERGECOPY	0xC000CA 0xF0008A	P & S P S	
		0x0500A9 0x0A0329	$\sim(P \mid D) \sim(P \mid D)$ $\sim P \& D$	R2_NOTMERGESEN R2_MASKNOTPEN
		0x500325	P & $\sim D$	R2_MASKPENNOT
	PATINVERT	0x5A0049	P \wedge D	R2_XORPEN
		0x5F00E9	$\sim(P \& D)$	R2_NOTMASKPEN
		0xA000C9	P & D	R2_MASKPEN
		0xA50065	$\sim(P \wedge D)$	R2_NOTXORPEN
		0xAF0229	$\sim P \mid D$	R2_MERGENOTPEN
		0xF50225	P $\mid \sim D$	R2_MERGESENNOT
		0xFA0089	P $\mid D$	R2_MERGESEN
Destination & Source	NOTSRCERASE	0x1100A6	$\sim(S \mid D) S \& \sim D$	
	SRCEARASE	0x440328		
	SRCINVERT	0x660046	S \wedge D	
	SRCAND	0x8800C6	S & D	
	MERGEPAINT	0xBB0226	$\sim S \mid D$	
	SRCPAINT	0xEE0086	S $\mid D$	
Pattern, Source, & Destination	PATPAINT	0xFB0A09 0xB8074A	P $\mid \sim S \mid D$ P \wedge (S $\&$ (P \wedge D)))	
		0xE20746	D \wedge (S $\&$ (P \wedge D)))	

```
BOOL PatBlt(HDC hDC, int nXLeft, int nYLeft, int nWidth, int nHeight,
DWORD dwRop);
```

The PatBlt function combines the currently selected brush with the destination pixel in a rectangle region. Note that source bitmap parameters are not needed. The raster operations you can use are not restricted to the Microsoft named ROPs. You can use any ROP that does not use source.

To check if a raster operation uses a certain variable is quite simple. You just need to check if the ROP generates the same results when the variable is 1 and 0. Here are three in-line functions that provide the checking.

```
bool inline RopNeedsNoDestination(int Rop)
{
    return ((Rop & 0xAA) >> 1) == (Rop & 0x55);
}

bool inline RopNeedsNoSource(int Rop)
{
    return ((Rop & 0xCC) >> 2) == (Rop & 0x33);
}

bool inline RopNeedsNoPattern(int Rop)
{
    return ((Rop & 0xF0) >> 4) == (Rop & 0x0F);
}
```

BLACKNESS, WHITENESS

These two ROPs are commonly used to clear a surface to an initial state. BLACKNESS clears all the pixels to zero, WHITENESS sets every bit in all the pixels to 1. On a palletized device, the result is not necessarily black and white, but generally the palette should be set to make sure the first palette entry is black, and the last one is white.

These two raster operations do not depend on any of the three variables. You can imagine the operation is implemented using the most efficient method: constant memory set. BLACKNESS can be implemented using `memset(pBits, 0, nImageSize)`, and WHITENESS can be implemented using `memset(pBits, 0xFF, nImageSize)`.

When a DDB or a DIB section is first created, its pixel array is normally undefined, unless it's an initialized DDB or a memory-mapped DIB section. It's a good habit to set the surface to a known state.

Black and white are special colors because they are represented using 0 and 1. For color C, for example, we have the following properties:

Black AND C = Black
Black OR C = C
Black XOR C = C
White AND C = C
White OR C = White
White XOR C = NOT C

When mask bitmaps are used to make a bitmap partly black, or partly white, these properties play an important role in making interesting mixes of bitmaps.

Pattern-Only ROPs: PATCOPY, R3_NOTCOPYPEN

PATCOPY can be used to fill a rectangle area with the current brush, similar to the FillRect function. Its opposite

function (0x0F0001) does not have an official name, so in [Table 11-1](#) it's named after its corresponding binary ROP R2_NOTCOPYPEN as R3_NOTCOPYPEN. R3_NOTCOPYPEN fills a rectangle area with the opposite color of the current brush.

Source-Only ROPs: SRCCOPY, NOTSRCCOPY

We have seen lots of usage of SRCCOPY; it simply copies the source pixel to destination. It's normally used to display the full bitmap in its original code. NOTSRCCOPY copies the opposite copy of the source pixel to destination. In high color or true color modes, it can be used to generate the negative image of an image.

It's documented (KB Q174534) that Windows NT 4.0 may not always handle NOTSRCCOPY raster operation properly, which should have been fixed in service pack 4.

Destination-Only ROPs: R3_NOP, DSTINVERT

DSTINVERT replaces a destination pixel with its opposite color. R3_NOP replaces a destination pixel with itself, a total waste of CPU power. You should expect GDI to be smart enough to skip the R3_NOP operation before moving each pixel to nowhere.

Destination-Free ROPs: MERGECOPY

There are ten ternary raster operations that depend on both source and pattern, but not on destination. Only one of them is named: MERGECOPY.

MERGECOPY (0xC000CA) is a misleading name. MERGE means logical OR in binary raster operations, but here it's used to mean logical AND. MERGECOPY replaces a destination pixel with the logical AND of a source pixel and a brush pixel.

When the source bitmap is a monochrome bitmap, MERGECOPY colors white pixels to the brush color, and leaves black pixels unchanged.

When the source bitmap is not a monochrome bitmap, sometimes it's better to use the brush as a mask. For example, to display only the red channel of a color image, create a solid red (RGB(0xFF, 0, 0)) brush, and use MERGECOPY as the raster operation to display the image. It copies the red channel untouched, and sets the other two channels to 0. The result is an image with only different shades of red color.

Here is a routine that displays one channel of an image according to a mask parameter. If it's RGB(0xFF, 0, 0), the red channel will be displayed.

```
void DisplayChannel(HDC hDC, int x, int y, int width, int height,  
    HDC hDCSource, COLORREF mask)  
{  
    HBRUSH hRed = CreateSolidBrush(mask);  
    HBRUSH hOld = (HBRUSH) SelectObject(hDC, hRed);  
    BitBlt(hDC, x, y, width, height, hDCSource, 0, 0, MERGECOPY);  
    SelectObject(hDC, hOld);
```

```
    DeleteObject(hRed);  
}
```

Image channel splitting is a common feature provided by image editors. The theory is that graphic artists would like to examine an image per individual channel, touch it to remove possible defects, and then combine them to form a new image. For such channel splitting, the end result should be true grayscale images. If the destination surface in the routine shown above is an 8-bpp surface, and its color table is set up to match the single-channel color scale, that channel of the original image will be converted to a grayscale image. You need to change only the color table later to be a grayscale color table.

[Listing 11-1](#) shows a ChannelSplit routine that can split any color DIB to three gray-scale images, one for each RGB channel.

Listing 11-1 Split Bitmap into RGB Channels

```
if ( hRslt==NULL )
    return NULL;

SelectObject(hMemDC, hRslt);

HBRUSH hBrush = CreateSolidBrush(Mask); // solid red, green, or blue
HGDIOBJ hOld = SelectObject(hMemDC, hBrush);

StretchDIBits(hMemDC, 0, 0, width, height, 0, 0, width, height,
    pBits, pBMI, DIB_RGB_COLORS, MERGECOPY);

for (i=0; i<256; i++) // convert to real grayscale color table
{
    bmi8bpp.bmiColor[i].rgbRed = i;
    bmi8bpp.bmiColor[i].rgbGreen = i;
    bmi8bpp.bmiColor[i].rgbBlue = i;
}
SetDIBColorTable(hMemDC, 0, 256, bmi8bpp.bmiColor);

SelectObject(hMemDC, hOld);
DeleteObject(hBrush);

return hRslt;
}
```

The ChannelSplit routine creates an 8-bpp DIB section, with a color table containing shades of color in one of the RGB channels. For example, if the Mask parameter is RGB(255, 0, 0), the color table contains RGB(0, 0, 0), RGB(1, 0, 0), up to RGB(255, 0, 0). The DIB section is selected into a memory DC, together with a solid brush created from the same channel mask. StretchDIBits uses MERGECOPY ROP to split the channel, and matches each pixel with the DIB section color table. If a pixel is RGB(r, g, b) in the source DIB, MERGECOPY generates RGB(r, 0, 0), and matching with the DIB section color table should give index r, which will be stored in the DIB section pixel array. The color finally modifies the color table to be a true grayscale color table, so that when the DIB section is displayed, it's displayed as a grayscale image.

The good news is that ChannelSplit works on any DIB image—images with different color depths, bit masks, compressed or uncompressed—with much coding on your side. The bad news is that although the grayscale image generated is quite good in most cases, the result is not perfect. For an accurate match, GDI would have to search through the full color table to find either a perfect match, or the best approximation, which could be quite slow. Judging from the result, which is apparent only on images with smooth shades of colors, GDI is using an approximation scheme, apparently for performance reasons. Instead of searching through the whole color table for every pixel, it's possible to build an RGB grid of N by N by N points. Each point on the grid is accurately matched with the color table when it's needed. For every RGB pixel generated by the raster operation, the closest point on the grid approximates it, whose matched color index will be the color index for the original pixel.

To implement a perfect channel-splitting algorithm, we have to manipulate the pixel array directly. We will present more on this topic later in this chapter.

Source-Free ROPs

Of the 10 ternary raster operations that depend on both pattern and destination, but not on source, only one is named PATINVERT. These raster operations provide the same features as the binary raster operations set by SetROP2.

These source-free ROPs can be used with the PatBlt functions, without using the more complicated function calls. They can be used to modify the current picture in a device context, which could be a physical device, or a memory device backed up by a DDB or a DIB section. For example, if you have a black/white picture, you can use R3_MASKPEN (0xA000C9) to color it with a color brush, or split its RGB channels. If a chessboard pattern brush is used, R3_MASKPEN can create a masked image effect, where half the pixels remain the same and the other half turn black.

Pattern-Free ROPs

The next group of raster operations uses both source and destination, but not pattern. Of the 10 possible raster operations, 6 are named. You can see Microsoft gives them more weight.

A named ternary raster operation normally means the operating system uses it itself. SRCAND and SRCINVERT are used by Windows to display icons and cursors. An icon or a cursor resource usually contains a group of icons with different sizes and color depths. Each icon or cursor is usually made up of two bitmaps: a black/white mask bitmap and a color mask. A black/white icon or cursor may have only one bitmap, but it's doubled in height, with the second half being the bitmap to display. The mask bitmap is displayed using SRCAND to clear the area the color bitmap is going to cover. The color bitmap is then displayed using SRCINVERT.

Here is a piece of code to help us understand a cursor and how it's displayed using raster operations:

```
HICON hlcon = (HICON) LoadImage(hMod, MAKEINTRESOURCE(resid),
    IMAGE_ICON, 48, 48, LR_DEFAULTCOLOR);

if ( hlcon )
{
    DrawIcon(hDC, x, y, hlcon);
    ICONINFO iconinfo;
    GetIconInfo(hlcon, & iconinfo);
    DestroyIcon(hlcon);

    BITMAP bmp;
   GetObject(iconinfo.hbmMask, sizeof(bmp), & bmp);

    HGDIOBJ hOld = SelectObject(hMemDC, iconinfo.hbmMask);
    BitBlt(hDC, x+56,y, bmp.bmWidth, bmp.bmHeight, hMemDC,0,0,SRCCOPY);
    SelectObject(hMemDC, iconinfo.hbmColor);
    BitBlt(hDC, x+112,y, bmp.bmWidth, bmp.bmHeight, hMemDC,0,0,SRCCOPY);

    SelectObject(hMemDC, iconinfo.hbmMask);
    BitBlt(hDC, x+168,y, bmp.bmWidth, bmp.bmHeight, hMemDC,0,0,SRCAND);
    SelectObject(hMemDC, iconinfo.hbmColor);
    BitBlt(hDC, x+168,y, bmp.bmWidth, bmp.bmHeight, hMemDC,0,0,SRCINVERT);
```

```
SelectObject(hMemDC, hOld);
DeleteObject(iconinfo.hbmMask);
DeleteObject(iconinfo.hbmColor);
}
```

The code loads a 48-by-48 icon from a module using LoadImage, and displays it using the normal Win32 icon drawing function DrawIcon that stretches the icon to its normal size, which is usually 32-by-32. To satisfy our curiosity, GetIconInfo is called to return two DDB handles, one for the mask bitmap, the other for the color bitmap. The two bitmaps are then displayed separately so we can have a close look at them individually. The next part of the code simulates an icon display by displaying the two bitmaps at the same spot, using SRCAND for the mask bitmap, and SRCINVERT for the color bitmap. [Figure 11-3](#) shows the formation of several icons used by the Windows shell.

Figure 11-3. Raster operation usage in icon display.



For the sake of discussion, let's define two terms here. When displaying an icon, which is defined in a rectangular region, we want only some of the pixels in that region to be changed. The region we want to change is called the *opaque region*; the remaining region is called the *transparent region*. Normally, the opaque region is the shape of the icon, for example a recycle bin, or a file folder.

[Figure 11-3](#) shows that a mask bitmap uses black (0) for the opaque region and white (1) for the transparent region. So the first BitBlt call using the SRCAND ROP changes opaque region to black, while leaving the transparent region unchanged. The color bitmap uses color pixels for the opaque region and black (0) for the transparent region. The second BitBlt call using the SRCINVERT ROP displays the color pixels without changing the transparent region.

If the mask bitmap is created using white (1) for the opaque region and black (0) for the transparent region, instead of using SRCAND, ROP 0x220326 (DSna) should be used to clear the opaque region. Note that the NOT operator in DSna effectively reverses the mask bitmap on the fly.

It's possible that the mask bitmap and the color bitmap use different opaque regions. If the opaque region in the mask bitmap is smaller than the opaque region in the color bitmap, when the second ROP is applied some of the pixels in the opaque region are not cleared to black (1); SRCINVERT replaces the destination pixel with D^S instead of S. On the other hand, if the two opaque regions are the same, SRCPAINT (DSo) can be used to do the same job.

What if we have the same mask bitmap as shown in [Figure 11-3](#), but the transparent pixel in the color bitmap is

white (1), instead of black (0)? Applying SRCAND and SRCINVERT will invert the transparent pixel, instead of leaving it untouched. We need to change the first ROP that applies the mask to be SRCERASE (SDna). If the mask bitmap has reversed colors, the first ROP needs to be changed to NOTSRC ERASE (SDon). MERGEPAINT (~S | D, DSno) can be used to draw an inverted source bitmap.

Now we have found uses for all six pattern-free ternary raster operations to which Microsoft has given a name. If you feel a little bit lost, [Table 11-2](#) summarizes how to use them to display both a mask bitmap and a color bitmap under different conditions.

Table 11-2. Mask-bitmap-based Transparent Bitmap Display

Mask Bitmap	Mask ROP	Masking Result	Color Bitmap	Color ROP	Final Result
(White, Black)	SRCAND	(D, Black)	(Black, C)	SRCINVERT	(D, C)
(White, Black)	SRCAND	(D, Black)	(Black, C)	SRCPAINT	(D, C)
(White, Black)	SRCAND	(D, Black)	(White, C)	MERGEPAINT	(D, Cn)
(Black, White)	R3_DSna	(D, Black)	(Black, C)	SRCINVERT	(D, C)
(White, Black)	SRCCERASE	(Dn, Black)	(White, C)	SRCINVERT	(D, C)
(Black, White)	NOTSRCERASE	(Dn, Black)	(White, C)	SRCINVERT	(D, C)

[Table 11-2](#) uses an (X, Y) notation to denote the formation of pixels in a bitmap, where X is the color of transparent pixels, and Y is the color of opaque pixels. So the first row in the table can be read as: If the mask bitmap has white as the transparent color and black as the opaque color, after SRCAND is used to draw the mask, the transparent region is unchanged and the opaque region is changed to black color. If a color bitmap with black as the transparent color is drawn using SRCINVERT as ROP, the pixels in the opaque region are changed to pixels in the color bitmap while the pixels in the transparent region remain the same.

Other Raster Operations

We have discussed ternary raster operations that are independent of pattern, source, and brush, depend on only one of them, and depend on only two of them. The total number of raster operations that do not depend on all three variables is $2 + 2^2 \cdot 3 + 10 \cdot 3$, or 38. The remaining ternary raster operations depend on all three variables, and there are 218 of them.

GDI names only one of the 218 three-variable raster operations: PATPAINT (P | ~S | D). PATPAINT combines a brush pixel, the inverse of a source pixel and a destination pixel using logical OR. Without knowing the restrictions on the variables, it's hard to figure out why someone would need a raster operation like this. So let's ignore PATPAINT, and turn our attention to something we can really find a use for.

One way to draw transparent bitmaps is to use a mask bitmap and a source (color) bitmap, as demonstrated by the icon display example above. The trouble with this mask-bitmap-based method is that you have to create a mask bitmap that matches the color pixels precisely. What if we can define the mask using a brush, instead of a separate bitmap? For example, suppose we create a chessboardlike pattern, in which half of the pixel is black and the other half is white. Can we display alternate pixels in the color bitmap on a device surface without changing the other half? What kind of raster operation needs to be used?

We could deduct the raster operation using some simple Boolean algebra. The effect we want to achieve can be described as $(P \& S) | (\sim P \& D)$. That is, if the brush pixel is 1 (white), the result should be the source bitmap pixel;

otherwise, use the destination pixel. Replacing P, S, D with 0xF0, 0xCC, and 0xAA, we get 0xCA. Check a table with a complete listing of ternary raster operations; the full code for it is 0xCA07A9, and its official formula is $D \wedge (P \wedge (S \wedge D))$.

If we want to reverse the logic of P, the intuitive formula will be $(\sim P \wedge S) \mid (P \wedge D)$, whose code is 0xAC, or 0xAC0744, and its official formula is $S \wedge (P \wedge (\sim D))$.

[Listing 11-2](#) shows a FadeIn routine that uses a raster operation 0xCA07A9 to fade-in an image. A source image is displayed in four steps, each using a different pattern brush. The first pattern enables 1/64 pixels from the source image, the second pattern enables 4/64, the third enables 16/64, and the last one enables all the pixels.

Listing 11-2 Gradually Display a Bitmap Using Raster Operation

```
// Gradually display a DIB on a destination surface in 4 steps
void FadeIn(HDC hDC, int x, int y, int w, int h,
            const BITMAPINFO * pBMI, const void * pBits)
{
    const WORD Mask11[8] = { 0x80,0x00,0x00,0x00, 0x00,0x00,0x00,0x00 };
    const WORD Mask22[8] = { 0x88,0x00,0x00,0x00, 0x88,0x00,0x00,0x00 };
    const WORD Mask44[8] = { 0xAA,0x00,0xAA,0x00, 0xAA,0x00,0xAA,0x00 };
    const WORD Mask88[8] = { 0xFF,0xFF,0xFF,0xFF, 0xFF,0xFF,0xFF,0xFF };

    const WORD * Mask[4] = { Mask11, Mask22, Mask44, Mask88 };

    for (int i=0; i<4; i++)
    {
        HBITMAP hMask = CreateBitmap(8, 8, 1, 1, Mask[i]);
        HBRUSH hBrush= CreatePatternBrush(hMask);
        DeleteObject(hMask);
        HGDIOBJ hOld = SelectObject(hDC, hBrush);

        // D^(P&(S^D)), if P then S else D
        StretchDIBits(hDC, x, y, w, h, 0, 0, w, h,
                      pBits, pBMI, DIB_RGB_COLORS, 0xCA07A9);

        SelectObject(hDC, hOld);
        DeleteObject(hBrush);
    }
}
```

Recall that we discussed how icons are displayed using the SRCAND and SRCINVERT raster operations in two bitbltng calls. If we are running on Windows NT-based systems, we can create a pattern brush using the mask bitmap, and combine the two drawing calls into a single call. The raster operation we need is $(D \wedge P) \wedge S$, which is 0x6C01E8. Here is a piece of code that implements this idea:

```
void MaskBitmapNT(HDC hDC, int x, int y, int width, int height,
                  HBITMAP hMask, HDC hMemDC)
```

```
{  
    HBRUSH hBrush = CreatePatternBrush(hMask);  
    HGDIOBJ hOld = SelectObject(hDC, hBrush);  
    POINT org = { x, y };  
    LPtoDP(hDC, &org, 1);  
    SetBrushOrgEx(hDC, org.x, org.y, NULL);  
  
    BitBlt(hDC, x, y, width, height, hMemDC, 0, 0, 0x6C01E8); // S^(P&D)  
  
    SelectObject(hDC, hOld);  
    DeleteObject(hBrush);  
}
```

Note that Windows 95-based systems do not support pattern brushes larger than 8-by-8 pixels. Also, moving the mask bitmap to a pattern brush restricts the routine to work well only in MM_TEXT mapping mode, because a pattern brush does not scale with mapping modes or world transformation.

If the source bitmap is a monochrome bitmap, we can use a color brush to color the bitmap and display it transparently. If we want to display black pixels (0) in the source bitmap using a brush color, and use white pixels (1) as transparent pixels, the formula we need is ($\sim S \& P$) | ($S \& D$). Its ROP code is 0xB8, or 0xB8074A, and its official formula is $P^{\wedge}(S \& (P^{\wedge}D))$. If we reverse the logic of the source bitmap, the intuitive formula becomes $(S \& P) | (\sim S \& D)$, ROP code is 0xE20746, and its official formula is $D^{\wedge}(S \& (P^{\wedge}D))$.

[Listing 11-3](#) demonstrates how to use ROP 0xB8074A to color the opaque pixels of a monochrome bitmap with an arbitrary brush. Routine ColorBitmap does the drawing using the 0xB8074A ROP. Routine TestColoring demonstrates using five different brushes to color a monochrome bitmap.

Listing 11-3 Coloring Monochrome Bitmap Using Raster Operation

```
void ColorBitmap(HDC hDC, int x, int y, int w, int h, HDC hMemDC,  
                 HBRUSH hBrush)  
{  
    // P^{\wedge}(S \& (P^{\wedge}D)), if (S) D else P  
    HGDIOBJ hOldBrush = SelectObject(hDC, hBrush);  
    BitBlt(hDC, x, y, w, h, hMemDC, 0, 0, 0xB8074A);  
    SelectObject(hDC, hOldBrush);  
}  
  
void TestColoring(HDC hDC, HINSTANCE hInstance)  
{  
    HBITMAP hPttrn;  
    HBITMAP hBitmap = LoadBitmap(hInstance, MAKEINTRESOURCE(IDB_CONFUSE));  
    BITMAP bmp;  
    GetObject(hBitmap, sizeof(bmp), & bmp);  
  
    SetTextColor(hDC, RGB(0, 0, 0));  
    SetBkColor(hDC, RGB(0xFF, 0xFF, 0xFF));
```

```
HDC hMemDC = CreateCompatibleDC(NULL);
HGDIOBJ hOld = SelectObject(hMemDC, hBitmap);

for (int I=0; I<5; I++)
{
    HBRUSH hBrush;
    switch (I)
    {
        case 0: hBrush = CreateSolidBrush(RGB(0xFF, 0, 0)); break;
        case 1: hBrush = CreateSolidBrush(RGB(0, 0xFF, 0)); break;
        case 2: hPttrn = LoadBitmap(hInstance,
                                     MAKEINTRESOURCE(IDB_COLOR));
                  hBrush = CreatePatternBrush(hPttrn);
                  DeleteObject(hPttrn);
                  break;
        case 3: hBrush = CreateHatchBrush(HS_DIAGCROSS,
                                         RGB(0, 0, 0xFF));
                  break;
        case 4: hPttrn = LoadBitmap(hInstance,
                                     MAKEINTRESOURCE(IDB_WOOD01));
                  hBrush = CreatePatternBrush(hPttrn);
                  DeleteObject(hPttrn);
    }

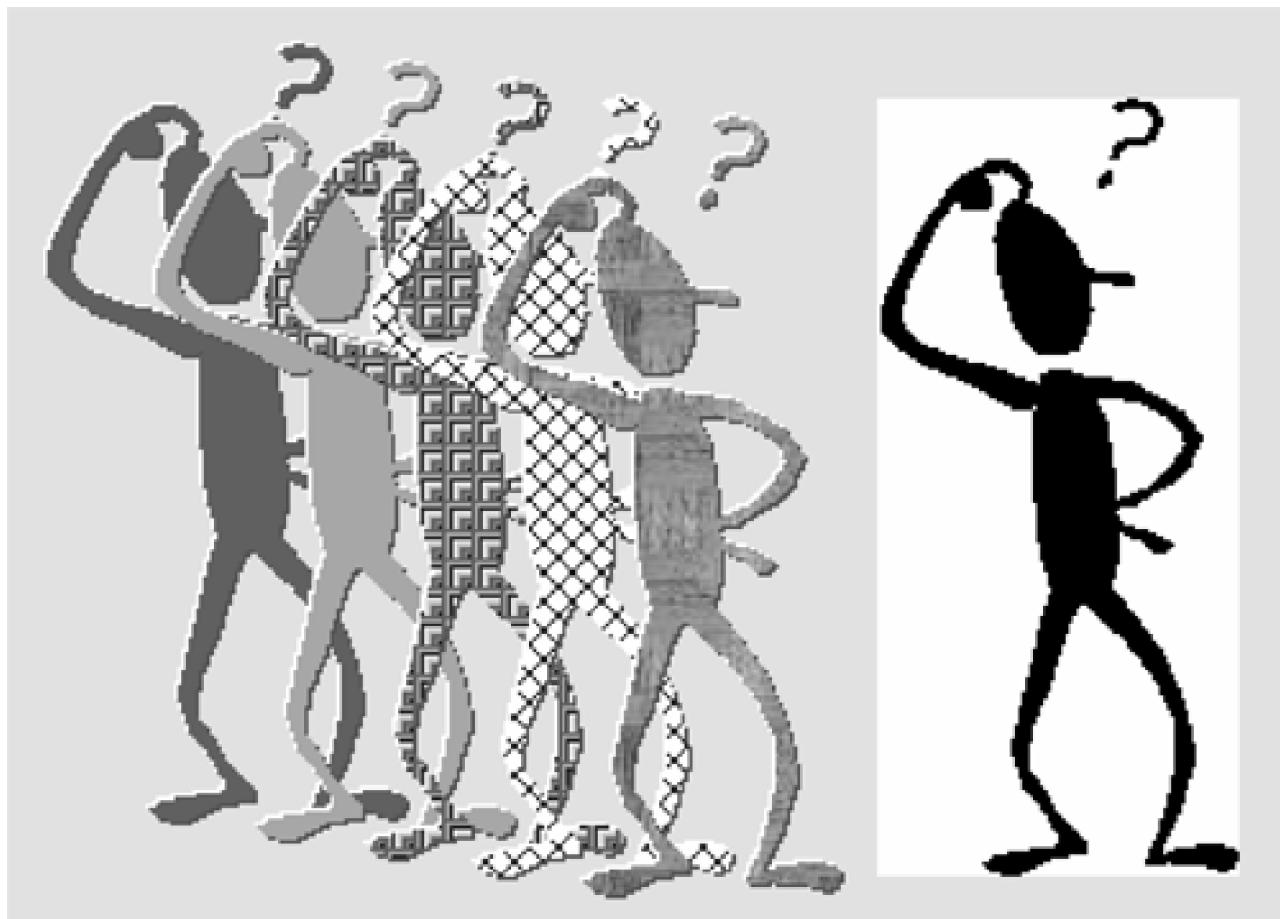
    ColorBitmap(hDC, I*30+10-1, I*5+10-1, bmp.bmWidth, bmp.bmHeight,
                hMemDC, (HBRUSH)GetStockObject(WHITE_BRUSH));
    ColorBitmap(hDC, I*30+10+1, I*5+10+1, bmp.bmWidth, bmp.bmHeight,
                hMemDC, (HBRUSH)GetStockObject(DKGRAY_BRUSH));
    ColorBitmap(hDC, I*30+10, I*5+10, bmp.bmWidth, bmp.bmHeight,
                hMemDC, hBrush);
    DeleteObject(hBrush);
}

BitBlt(hDC, 240, 25, bmp.bmWidth, bmp.bmHeight, hMemDC, 0, 0,
       SRCCOPY);

SelectObject(hMemDC, hOld);
DeleteObject(hBitmap);
DeleteObject(hMemDC);
}
```

The routine loads one bitmap that has a confused person, and displays five con fused people. For example, the white brush and the black brush are used to draw two slightly offset bitmaps to create a simple 3D effect. Because the 0xB8074A ROP draws bitmaps transparently, only opaque pixels are drawn. [Figure 11-4](#) shows the five colored bitmaps, together with the original colorless bitmap on the right.

Figure 11-4. Coloring monochrome bitmaps transparently.



[< BACK](#) [NEXT >](#)

11.2 TRANSPARENT BITMAPS

With so many different raster operations, Microsoft still thinks displaying transparent bitmaps is a hard thing to do, so three functions are added mainly to solve this problem.

```
BOOL PlgBlt(HDC hdcDest, CONST POINT * lpPoint, HDC hDCSrc, int nXSrc,
    int nYSrc, int nWidth, int nHeight, HBITMAP hbmMask,
    int xMask, int yMask);
BOOL MaskBlt(HDC hdcDest, int nXDest, int nYDest, int nWidth, int nHeight,
    HDC hdcSrc, int nXSrc, int nYSrc, HBITMAP hbmMask,
    int xMask, int yMask, DWORD dwRop);
BOOL TransparentBlt(HDC hdcDest, int nXOriginDest, int nYOriginDest,
    int nWidthDest, nHeightDest, HDC hdcSrc, int nXOriginSrc,
    int nYOriginSrc, int nWidthSrc, int nHeightSrc,
    UINT crTransparent);
```

These three functions are not supported on all Win32 platforms. PlgBlt and Mask-Blt are supported only on Windows NT-based systems. TransparentBlt is supported only on Windows 98, Windows 2000, or subsequent systems. We will discuss how they can be simulated using other basic bitmap displaying functions and direct bitmap accessing.

By looking at the function prototypes, you can quickly notice their similarities. They all accept two device context handles—one for the source device, one for the destination device. So these functions will work on a memory device context with a DDB or a DIB section selected, or on a physical graphics device supporting raster operations. But a DIB is not supported directly; you have to convert a DIB to either a DDB or a DIB section to use these functions.

Parallelogram Bit-Block Transfer: PlgBlt

The PlgBlt function performs two functions: transform a rectangle bitmap into a parallelogram, and control the transparency using a mask bitmap. So we have three major players in the game: destination, source, and mask.

The source rectangle is a subset of the source device surface defined by the parameters nXSrc, nYSrc, nWidth, and nHeight. All of them are in logical coordinate space in the source device context. As for other simpler bit-block transfer functions, the source device context cannot have rotation and shear transformation, but scaling, translation, and reflection are allowed. This restriction ensures that the source rectangle is always a rectangle in the source device's device coordinate space parallel to both axes.

A device context handle and an array of three points define the destination parallelogram. We mentioned before that an affine transformation is uniquely defined by mapping three points in one space to three points in another space. The three points pointed to by the lpPoint parameter uniquely define a parallelogram in the destination surface, whose fourth point is determined by $D = B + C - A$, where $A = \text{lpPoint}[0]$, $B = \text{lpPoint}[1]$, and $C = \text{lpPoint}[2]$. The upper-left corner of the source rectangle is mapped to A, the upper-right corner is mapped to B, the lower-left corner is mapped to C, and the lower-right corner is mapped to D.

The mapping from the source rectangle to the destination parallelogram is a generic affine transformation, which

allows translation, scaling, reflection, rotation, and shearing. In terms of geometric shape, PlgBlt adds rotation and shearing when compared with StretchBlt.

The mask bitmap is specified by a bitmap handle, and two integers. The bitmap must be a monochrome bitmap; otherwise, the function will fail. The two integers xMask and yMask specify the location of the pixel in the mask that corresponds to the upper-left corner of the source bitmap. When running out of pixels in the mask bitmap, it will be used repetitively, in the same way a pattern brush is used to fill an area.

If a mask bitmap is not specified, the whole source bitmap is displayed in the destination parallelogram. Otherwise, a value of 1 (white) indicates copying the source pixel to destination. A value of 0 (black) leaves the destination pixel unchanged. If we convert the mask bitmap to a pattern brush, the logic of masking can be expressed using raster operations 0xCA07A9, which is P&S | ~P&D.

So for a Windows NT-based system, StretchBlt can replace PlgBlt by adjusting world transformation, convert the mask bitmap to a pattern brush, and use 0xCA07A9 as raster operation. For non-NT-based systems, StretchBlt can replace PlgBlt only when no rotation and shearing are involved, and the mask bitmap is no more than 8-by-8 pixels.

[Listing 11-4](#) shows some code to display a 3D cube using PlgBlt. The DrawCube routine draws the three faces of a 3D cube using PlgBlt, with a source and mask bitmap. The MaskCube routine controls the geometric of the 3D and creation of a rounded rectangle mask of the same size as the source bitmap.

Listing 11-4 Using PlgBlt to Display a 3D Cube

```
void DrawCube(HDC hDC, int x, int y, int dh, int dx, int dy,
              HDC hMemDC, int w, int h, HBITMAP hMask)
{
    SetStretchBltMode(hDC, HALFTONE);

    //      6
    //  0      4
    //      1
    //  2      5
    //      3
    POINT P[3] = { { x-dx, y-dy }, { x, y }, { x-dx, y-dy+dh } }; // 012
    POINT Q[3] = { { x, y }, { x+dx, y-dy }, { x, y+dh } };      // 143
    POINT R[3] = { { x-dx, y-dy }, { x, y-dy-dy }, { x, y } }; // 061

    PlgBlt(hDC, P, hMemDC, 0, 0, w, h, hMask, 0, 0);
    PlgBlt(hDC, Q, hMemDC, 0, 0, w, h, hMask, 0, 0);
    PlgBlt(hDC, R, hMemDC, 0, 0, w, h, hMask, 0, 0);
}

void MaskCube(HDC hDC, int size, int x, int y, int w, int h,
              HBITMAP hBmp, HDC hMemDC, bool mask)
{
    HBITMAP hMask = NULL;

    if ( mask )
    {

```

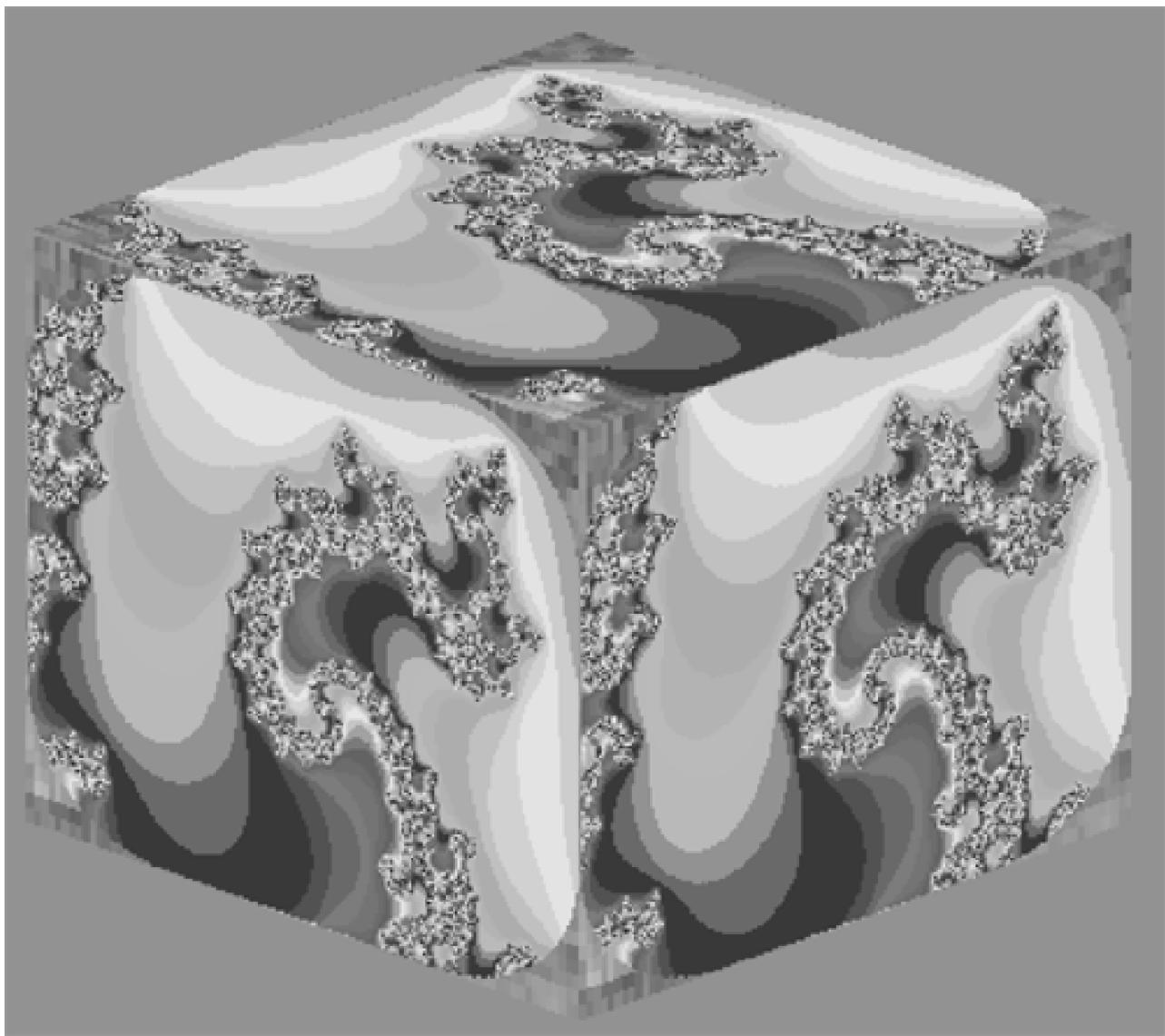
```
hMask = CreateBitmap(w, h, 1, 1, NULL);
SelectObject(hMemDC, hMask);
PatBlt(hMemDC, 0, 0, w, h, BLACKNESS);
RoundRect(hMemDC, 0, 0, w, h, w/2, h/2);
}

int dx = size * 94 / 100; // cos(20)
int dy = size * 34 / 100; // sin(20)
SelectObject(hMemDC, hBmp);
DrawCube(hDC, x+dx, y+size, size, dx, dy, hMemDC, w, h, hMask);

if ( hMask )
    DeleteObject(hMask);
}
```

Figure 11-5 shows the result of drawing two overlapping cubes. The first cube is drawn using a wooden texture without a mask bitmap, so a solid cube is formed. The second cube is drawn using a bitmap from the Mandelbrot set program with a rounded rectangle mask.

Figure 11-5. 3D cube displayed using PlgBlt.



Doesn't it look cool? But the bad news is that it's available only on Windows NT-based systems, not even on Windows 98. It will be a good review of GDI features we learned so far to provide an implementation of PlgBlt that will run on all Win32 platforms. So let's try it.

[Listing 11-5](#) shows the G_PlgBlt function, which simulates PlgBlt.

First let's solve the no-shear, no-rotation case. When no shear or rotation is needed, PlgBlt can be implemented using multiple StretchBlt calls with simple raster operations. The code in [Listing 11-5](#) creates an instance of a KReverseAffine class on the stack, which handles the transformation. When no shear or rotation is involved, the KReverseAffine::Simple method returns true. The routine checks if there is a valid mask bitmap handle. If not, it calls StretchBlt with SRCCOPY. When there is a mask bitmap, we have to figure out how to implement it using multiple calls.

Recall that if the mask pixel is white (1), the destination pixel should be changed to a source pixel; otherwise, leave the destination pixel unchanged. The Boolean algebra formula for this is $M \& S | \sim S \& D$, or $D^{\wedge}(M \&(S^{\wedge}D))$. Implementing the latter literally needs an intermediate bitmap, because D is used twice. But if we change the formula to be $S^{\wedge}(\sim M \&(S^{\wedge}D))$, S is used twice, and D is used only once. From this formula, we know how to implement the masking semantics. First change D to be $S^{\wedge}D$ using SRCINVERT, then combine the new D with $\sim M$ using AND, and finally use another SRC INVERT with S. For the second operation, the mask bitmap is treated as a source bitmap with tiling consideration. We have to use an unnamed ROP 0x220326, which does $\sim S \& D$.

Note that the mask bitmap is drawn using StretchTile, which tiles the mask bitmap over the destination surface.

Listing 11-5 Implementing PIgBlt

```
BOOL G_PlgBlt(HDC hdcDest, const POINT * pPoint,
    HDC hdcSrc, int nXSrc, int nYSrc, int nWidth, int nHeight,
    HBITMAP hbmMask, int xMask, int yMask)
{
    KReverseAffine map(pPoint);

    if ( map.Simple() ) // no shear and rotation
    {
        int x = pPoint[0].x;
        int y = pPoint[0].y;
        int w = pPoint[1].x-pPoint[0].x;
        int h = pPoint[2].y-pPoint[0].y;

        if ( hbmMask ) // has mask if (M) the S else D, S ^ (~M & (S^D))
        {
            StretchBlt(hdcDest, x, y, w, h, hdcSrc, nXSrc, nYSrc,
                nWidth, nHeight, SRCINVERT);
            StretchTile(hdcDest, x, y, w, h, hbmMask, xMask, yMask,
                nWidth, nHeight, 0x220326);
            return StretchBlt(hdcDest, x, y, w, h, hdcSrc, nXSrc,
                nYSrc, nWidth, nHeight, SRCINVERT);
        }
        else
            return StretchBlt(hdcDest, x, y, w, h, hdcSrc, nXSrc,
                nYSrc, nWidth, nHeight, SRCCOPY);
    }

    map.Setup(nXSrc, nYSrc, nWidth, nHeight);

    HDC hdcMask = NULL;
    int maskwidth = 0;
    int maskheight= 0;

    if ( hbmMask )
    {
        BITMAP bmp;
        GetObject(hbmMask, sizeof(bmp), & bmp);

        maskwidth = bmp.bmWidth;
        maskheight = bmp.bmHeight;

        hdcMask = CreateCompatibleDC(NULL);
        SelectObject(hdcMask, hbmMask);
```

```
}

for (int dy=map.miny; dy<=map.maxy; dy++)
for (int dx=map.minx; dx<=map.maxx; dx++)
{
    float sx, sy;
    map.Map(dx, dy, sx, sy);

    if ( (sx>=nXSrc) && (sx<=(nXSrc+nWidth)) )
    if ( (sy>=nYSrc) && (sy<=(nYSrc+nHeight)) )
        if ( hbmMask )
    {
        if ( GetPixel(hdcMask, ((int)sx+xMask) % maskwidth,
                      ((int)sy+yMask) % maskheight) )
            SetPixel(hdcDest, dx, dy, GetPixel(hdcSrc,
                                              (int)sx, (int)sy));
    }
    else
        SetPixel(hdcDest, dx, dy, GetPixel(hdcSrc,
                                          (int)sx, (int)sy));
}

if ( hdcMask )
    DeleteObject(hdcMask);

return TRUE;
}
```

When shearing or rotation is involved, we have to do some real work. The code calls the KReverseAffine::Setup method to set up a reverse affine transformation from the parallelogram in the destination surface to the rectangle in the source surface. This is a common practice in handling bitmap rotation and shearing. If we map the source pixel into the destination surface, there will be gaps when stretching is needed or duplicate calculation when shrinking is involved. By going backwards from destination to source, we ensure that every destination pixel is calculated and calculated only once. This technique handles stretching or shrinking automatically. The Setup routine also calculates the bounding box of the destination parallelogram.

After setting up a memory DC for the mask bitmap, the code starts to loop through every point in the bounding box of the parallelogram. Each point in it is mapped to source bitmap coordinate space and compared with the source rectangle to see if it's in the source rectangle. Note the bounding box is larger than the parallelogram. The mapped destination point is stored in floating-point number and compared with the source rectangle. If we convert the numbers too soon to integer values, there will be some edge pixels displayed incorrectly because of rounding problems.

If a point is in the source rectangle, that point is in the parallelogram, so its pixel value needs to be calculated. If there is a mask bitmap, the corresponding pixel in the mask bitmap is read. If the pixel is white (1), or nonzero, the source pixel is drawn into the destination surface. Otherwise, the destination is unchanged. This implements the mask bitmap semantics. If there is no mask bitmap, the source pixel is simply copied to destination. Note that xMask and yMask are added to the source bitmap coordinate to honor the mask bitmap shift. The coordinate into the mask bitmap is modulo to its dimension to achieve the tiling effect.

[Listing 11-6](#) shows the supporting code and class.

Listing 11-6 Supporting Code and Class for PlgBlt Simulation

```
// dx, dy, dw, dh defines a destination rectangle
// sw, sh is the dimension of source rectangle
// sx, sy is the starting point within the source bitmap,
// which will be tiled to sw x sh in size
BOOL StretchTile(HDC    hDC, int dx, int dy, int dw, int dh,
                  HBITMAP hSrc, int sx, int sy, int sw, int sh,
                  DWORD rop)
{
    BITMAP bmp;

    if ( ! GetObject(hSrc, sizeof(BITMAP), & bmp) )
        return FALSE;

    HDC  hMemDC = CreateCompatibleDC(NULL);
    HGDIOBJ hOld = SelectObject(hMemDC, hSrc);

    int sy0 = sy % bmp.bmHeight; // current tile y origin

    for (int y=0; y<sh; y+=(bmp.bmHeight - sy0))
    {
        int height = min(bmp.bmHeight - sy0, sh - y); // tile height
        int sx0   = sx % bmp.bmWidth;                   // tile x origin

        for (int x=0; x<sw; x+=(bmp.bmWidth - sx0))
        {
            int width = min(bmp.bmWidth - sx0, sw - x); // tile width

            StretchBlt(hDC, dx+x*dw/sw, dy+y*dh/sh, dw*width/sw,
                        dh*height/sh, hMemDC, sx0, sy0, width, height, rop);
            sx0 = 0; // after first tile, full tile width
        }
        sy0 = 0; // after the first row, full tile height
    }

    SelectObject(hMemDC, hOld);
    DeleteObject(hMemDC);

    return TRUE;
}
void minmax(int x0, int x1, int x2, int x3, int & minx, int & maxx)
{
    if ( x0<x1 )
    { minx = x0; maxx = x1; }
```

```
else
{ minx = x1; maxx = x0; }

if ( x2<minx) minx = x2; else if ( x2>maxx) maxx = x2;
if ( x3<minx) minx = x3; else if ( x3>maxx) maxx = x3;
}

class KReverseAffine : public KAffine
{
    int x0, y0, x1, y1, x2, y2;

public:
    int minx, maxx, miny, maxy;

KReverseAffine(const POINT * pPoint)
{
    x0 = pPoint[0].x; // P0      P1
    y0 = pPoint[0].y; //
    x1 = pPoint[1].x; //
    y1 = pPoint[1].y; // P2      P3
    x2 = pPoint[2].x;
    y2 = pPoint[2].y;
}

bool Simple(void) const
{
    return (y0==y1) && (x0==x2);
}

void Setup(int nXSrc, int nYSrc, int nWidth, int nHeight)
{
    MapTri(x0, y0, x1, y1, x2, y2, nXSrc, nYSrc,
           nXSrc+nWidth, nYSrc, nXSrc, nYSrc+nHeight);
    minmax(x0, x1, x2, x2 + x1 - x0, minx, maxx);
    minmax(y0, y1, y2, y2 + y1 - y0, miny, maxy);
}
};
```

We have just implemented our first image shearing and rotation algorithm, and a simulation for the mighty PlgBlt. If you try the code, almost the same 3D cube as shown in [Figure 11-5](#) will be drawn. This will take seven times as long because the code is using the slow GetPixel, SetPixel functions. Later in this chapter we will discuss direct pixel manipulation, which will improve performance greatly.

Quaternary Raster Operations: MaskBlt

Some higher life form felt that ternary raster operations are not complicated enough, and went forward to invent quaternary raster operations: raster operations taking four variables. It's quite strange considering that only a single ternary raster operation that uses pattern, source, and brush is given a name in GDI, PATPAINT (P | ~S | D). We

don't even know how to use PATPAINT, although in the last section we managed to show uses for a few true three-variable ternary raster operations.

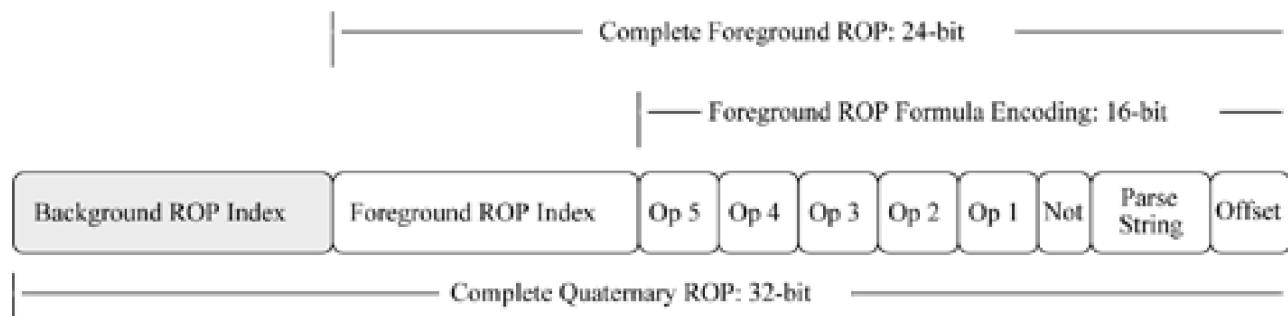
A ternary raster operation code can be seen as a combination of two binary raster operation codes, which control how S and D should be combined. The higher nibble of a ternary raster operation is used when P is 1; the lower nibble of it is used when P is 0. Take the identity ROP as an example; its code is 0xAA; the high nibble is 0xA, the lower nibble is 0xA. So the result of the ROP is independent of the brush. The code for D is 0xA, so the result is always D, the original destination. Now look at PAT INVERT; its code is 0x5A, the high nibble is 0x5, the lower nibble is 0xA. So when P is 1, ROP is ~D, otherwise it's D, which makes a P^D.

The fourth player in a quaternary raster operation is a monochrome mask bitmap. A quaternary raster operation code is made up of a foreground and a background ternary ROP code. The foreground ROP code is used when the mask pixel is 1, and the background ROP code is used when the mask pixel is 0. GDI defines a MAKEROP4 macro that combines two 24-bit ternary ROP codes to make one 32-quaternary ROP code.

```
#define MAKEROP4(fore, back) (DWORD (((back)<<8) & 0xFF000000) | (fore))
```

The macro takes the high 8-bit ROP index of the background ROP code, shifts it 8 bits to the left, and combines it with the 24-bit foreground ROP code to form a 32-bit quaternary ROP code. [Figure 11-6](#) illustrates the formation of a quaternary ROP code.

Figure 11-6. Quaternary raster operation code.



Note that the mask is not an equal player in a quaternary, because it's restricted to two values: 1 (white) or 0 (black). You can't have a color mask bitmap whose color pixels can be combined with brush, source, and destination color bitmaps. The main purpose of providing a quaternary ROP is to provide an easy and efficient way to achieve a transparent bitmap display. And MaskBlt is the only GDI function accepting a quaternary ROP.

MaskBlt is not hard to understand. The first five parameters specify a rectangle on the destination surface. The next three parameters determine a rectangle on the source surface, which has the same dimensions as the destination surface. These eight parameters are just like in BitBlt. But there is no StretchMaskBlt as we have StretchBlt for BitBlt. If an application wants stretching with MaskBlt, it has to set up the proper logical coordinate spaces. The next three parameters, hbmMask, xMask, yMask, specify a mask bitmap. It will be tiled in the same way as the mask bitmap in PlgBlt. The last parameter to MaskBlt is a quaternary ROP.

When the mask bitmap pixel is 1, the foreground ROP is used to combine brush, source surface, and destination surface to form a new destination surface; otherwise, the background ROP is used.

Due to the lack of good examples to demonstrate MaskBlt, let's use it to draw an icon one more time.

```
void MaskBltDrawIcon(HDC hDC, int x, int y, HICON hIcon)
{
    ICONINFO iconinfo;
    GetIconInfo(hIcon, &iconinfo);

    BITMAP bmp;
    GetObject(iconinfo.hbmMask, sizeof(bmp), &bmp);

    HDC hMemDC = CreateCompatibleDC(NULL);
    HGDIOBJ hOld = SelectObject(hMemDC, iconinfo.hbmColor);

    MaskBlt(hDC, x, y, bmp.bmWidth, bmp.bmHeight, hMemDC, 0, 0,
            iconinfo.hbmMask, 0, 0, MAKEROP4(SRCINVERT, SRCCOPY));

    SelectObject(hMemDC, hOld);
    DeleteObject(iconinfo.hbmMask);
    DeleteObject(iconinfo.hbmColor);
    DeleteObject(hMemDC);
}
```

Compared with previous code, now only a single call is needed. Compared with the MaskBitmapNT routine that uses a pattern brush and the strange 0x6C01E8 (S^P^D) ROP, now we pass the mask bitmap directly to MaskBlt without using a pattern brush. The quaternary ROP we use here is MAKEROP4(SRCINVERT, SRC COPY), which does XOR when the foreground pixel is 1, and copy when the foreground pixel is 0. This is according to the icon's definition. In actual use, when the foreground pixel is 1, the color bitmap pixel is always 0, so XOR does not change the destination pixel.

Simulating MaskBlt

MaskBlt provides a simple conceptual model of applying different raster operations to foreground and background pixels. But if you want to use MaskBlt, your application is restricted to Windows NT-based operating systems. We're going to provide a simulation of MaskBlt here.

Simulating MaskBlt turns out to be a very good exercise in gaining a deeper understanding of raster operations. The decision we made is that we don't want to provide a pixel-level raster operation implementation, because it will require us to implement all 256 ternary raster operations that MaskBlt depends on. Instead, we're trying to simulate MaskBlt using multiple ternary raster operations. You certainly lose performance in doing that, but still it's worth the effort for its educational value.

The routine G_MaskBlt shown in [Listing 11-7](#) provides a full implementation of MaskBlt. It breaks down the problem into several cases, from simple cases that involve no more than three raster operations to more general cases that involve a temporary bitmap and six raster operations. The routine first extracts the foreground and background ROP indexes from the quaternary ROP code. Recall that the goal of MaskBlt is to use the foreground ROP when the mask is 1 (white), and the background ROP when the mask is 0 (black). The foreground and background ROPs guide our implementation.

If the foreground and background ROPs are the same, the mask bitmap is useless; the original call can be

implemented using a single BitBlt call.

If the background ROP is independent of the destination bitmap, MaskBlt can be implemented using no more than three BitBlt calls. The first call uses source bitmap, and “fore XOR back” as raster operation. After that, the destination surface is changed to “fore XOR back.” The second call uses the mask bitmap, and SRCAND as the raster operation. After that the destination is changed to “if (M) (fore[^]back) else 0.” The third call uses the source bitmap, and “back XOR D” as the raster operation. So finally the destination surface is “if (M) fore else back”—the exact result we want.

Let's take an example. If the foreground ROP is D[^]S, background ROP is P. So the result we want to achieve is “if (M) D[^]S else P.” According to our code, the first ROP to use is D[^]S[^]P. After the first call, the destination surface is changed to “D[^]S[^]P.” Now the mask bitmap is applied using SRCAND, which changes the result to be “if (M) D[^]S[^]P else 0.” Finally, the source bitmap is used again with ROP D[^]P, and the result is “if (M) D[^]S else P.”

Why do we require the background ROP to be independent of the destination? The reason is that after applying the first ROP, the destination is already changed. Any subsequent ROP that references the original destination needs to be supported by a copy of it.

Similarly, if the foreground ROP is independent of the destination, we just need to change the second mask bitmap ROP to be NOTSRCAND (0x22), and the third ROP to be (fore[^]D).

Another simple case is when both the foreground and background ROPs are independent of the source bitmap. In this case we can treat the mask bitmap as the source bitmap, create a new ROP, and do BitBlt once when the mask bitmap selected into a memory DC. The ROP index we need can be calculated by (fore & 0xCC) | (back & 0x33), where 0xCC means source, and 0x33 means the “NOT source.” You see, we can take ROPs apart and also combine them. For example, suppose the foreground ROP is PATCOPY (0xF0), and the background ROP is PATINVERT (0x5A); and both are free of S. The new ROP will be (0xF0 & 0xCC) | (0x5A & 0x33), which is 0xC0 | 0x12, or 0xD2, or P[^](D&[~]S). Note that S here is the mask bitmap. The formula means that if S = 1 use P; otherwise, use P[^]D. That's it.

Listing 11-7 MaskBlt Simulation

```
BOOL TriBitBlt(HDC hdcDest, int nXDest,int nYDest,int nWidth,int nHeight,  
    HDC hdcSrc, int nXSrc, int nYSrc,  
    HBITMAP hbmMask, int xMask, int yMask,  
    DWORD rop1, DWORD rop2, DWORD rop3)  
{  
    HDC hMemDC = CreateCompatibleDC(hdcDest);  
    SelectObject(hMemDC, hbmMask);  
  
    if ( (rop1>>16)!=0xAA ) // not D  
        BitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,  
            hdcSrc, nXSrc, nYSrc, rop1);  
    BitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,  
        hMemDC, xMask, yMask, rop2);  
  
    DeleteObject(hMemDC);  
  
    if ( (rop3>>16)!=0xAA ) // not D
```

```
return BitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,
    hdcSrc, nXSrc, nYSrc, rop3);
else
    return TRUE;
}

inline bool D_independent(DWORD rop)
{
    return ((0xAA & rop)>>1)== (0x55 & rop);
}

inline bool S_independent(DWORD rop)
{
    return ((0xCC & rop)>>2)== (0x33 & rop);
}

BOOL G_MaskBlt(HDC hdcDest, int nXDest,int nYDest,int nWidth,int nHeight,
    HDC hdcSrc, int nXSrc, int nYSrc,
    HBITMAP hbmMask, int xMask, int yMask,
    DWORD dwRop
)
{
    DWORD back = (dwRop >> 24) & 0xFF;
    DWORD fore = (dwRop >> 16) & 0xFF;

    if ( back==fore ) // foreground=background, hbmMask not needed
        return BitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,
            hdcSrc, nXSrc, nYSrc, dwRop & 0xFFFFFFF);

    // if (M) D=fore(P,S,D) else D=back(P,S,D)

    if ( D_independent(back) ) // back independent of D
        return TriBitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,
            hdcSrc, nXSrc, nYSrc, hbmMask, xMask, yMask,
            fore^back << 16, // ( fore^back, fore^back )
            SRCAND, // ( fore^back, 0 )
            (back^0xAA) << 16); // { fore, back }

    if ( D_independent(fore) ) // fore independent of D
        return TriBitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,
            hdcSrc, nXSrc, nYSrc, hbmMask, xMask, yMask,
            (fore^back) << 16, // ( fore^back, fore^back )
            0x22 << 16, // ( 0, fore^back )
            (fore^0xAA) << 16); // { fore, back }

    // both foreground and background depend on D
    if ( S_independent(back) && S_independent(fore) )
        return TriBitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,
            NULL, 0, 0, hbmMask, xMask, yMask,
            0xAA << 16, // ( D, D )
```

```
( (fore & 0xCC) || (back & 0x33) ) << 16,  
0xAA << 16);  
  
// both foreground and background depend on D  
// either foreground or background depends on S  
HBITMAP hTemp = CreateCompatibleBitmap(hdcDest, nWidth, nHeight);  
HDC hMemDC = CreateCompatibleDC(hdcDest);  
SelectObject(hMemDC, hTemp);  
BitBlt(hMemDC, 0,0,nWidth,nHeight, hdcDest, nXDest, nYDest, SRCCOPY);  
SelectObject(hMemDC, GetCurrentObject(hdcDest, OBJ_BRUSH));  
BitBlt(hMemDC, 0,0,nWidth,nHeight, hdcSrc, nXSrc, nYSrc, back << 16);  
// hMemDC contains final background image  
  
BitBlt(hdcDest, 0, 0, nWidth, nHeight, hdcSrc, nXSrc, nYSrc,  
fore << 16); // foreground image  
  
TriBitBlt(hdcDest, nXDest, nYDest, nWidth, nHeight,  
hMemDC, 0, 0, hbmMask, xMask, yMask,  
SRCINVERT, // ( fore^back, fore^back )  
SRCAND, // ( fore^back, 0 )  
SRCINVERT); // { fore, back }  
  
DeleteObject(hMemDC);  
DeleteObject(hTemp);  
  
return TRUE;  
}
```

If the foreground and background ROP can't be handled by those simple cases, we're in trouble. We know both the foreground and background ROPs depend on the destination, but we don't have a way to implement this by using a single BitBlt call. Once one BitBlt call is made, the destination is already changed, so any subsequent reference to the destination is not using the original destination. The only solution we have is to create a temporary bitmap, make a copy of the destination surface, and build the background image in it. We then build the foreground image in the actual destination surface. The mask bitmap is then used to combine the foreground image and background image into one image. Case solved—using one to six BitBlt calls.

Color Keying: TransparentBlt

Both PlgBlt and MaskBlt use a monochrome mask bitmap to control the drawing of a source bitmap. The main trouble with this mask-bitmap-based approach is that we need two perfectly matching bitmaps: a source bitmap and a mask bitmap. Note that the mask bitmap needs to match the source bitmap in size and in every little detail in shape. It's lots of tedious work.

To solve this problem, we can turn to Hollywood special-effects artists for help. The movie industry has long been using the so-called blue-screen imaging techniques. A subject is first photographed in the front of an evenly lit, bright, pure blue background first. During composition, the blue background can be replaced with another image, known as the background plate. In this way, you can picture someone hanging on a few invisible wires in a studio; replace the background with a beautiful sky, and now that person is flying in the sky. Note that blue is not the only color that can be used, although it's the preferred choice. Another related chroma key process replaces everything in

the image over or under a brightness level with another image. These techniques are called color keying.

Windows 98 and Windows 2000 add color-keying support to GDI through the new `TransparentBlt` function. `TransparentBlt` accepts eleven parameters. The first five parameters specify a rectangle on the destination device surface, the next five parameters specify a rectangle on the source device surface, and the last parameter is the color key represented using an RGB value. `TransparentBlt` copies source surface pixels that do not match the color key to the destination surface; stretching or shrinking is applied when necessary. But `TransparentBlt` does not support mirroring, as does `StretchBlt`.

If a bitmap is prepared properly with a color key, `TransparentBlt` is very easy to use. Here is our guinea pig again, icon display, this time using `TransparentBlt`. Recall that an icon is made up of a monochrome mask bitmap and a color bitmap. For the transparent region, the color bitmap is normally black. Assuming the color is not used elsewhere, black can be taken as a transparent color key. A common practice is having the first pixel in a bitmap as the color key. The routine shown below uses the first pixel of the icon color bitmap as its color key to call `TransparentBlt`. The icon is displayed with a single bit-block transfer call.

```
void TransparentBltDrawIcon(HDC hDC, int x, int y, HICON hIcon)
{
    ICONINFO iconinfo;
    GetIconInfo(hIcon, &iconinfo);

    BITMAP bmp;
    GetObject(iconinfo.hbmMask, sizeof(bmp), &bmp);

    HDC hMemDC = CreateCompatibleDC(NULL);
    HGDIOBJ hOld = SelectObject(hMemDC, iconinfo.hbmColor);

    COLORREF crTrans = GetPixel(hMemDC, 0, 0);

    TransparentBlt(hDC,    x, y, bmp.bmWidth, bmp.bmHeight,
                  hMemDC, 0, 0, bmp.bmWidth, bmp.bmHeight,
                  crTrans);

    SelectObject(hMemDC, hOld);
    DeleteObject(iconinfo.hbmMask);
    DeleteObject(iconinfo.hbmColor);
    DeleteObject(hMemDC);
}
```

Again, `TransparentBlt` is a very cool function, especially considering its roots in Hollywood. The trouble is that `TransparentBlt` is supported only on Windows 98, Windows 2000, and follow-on products. It's not supported on Windows NT 4.0. It's exported from `MSIMG32.DLL`, not from `GDI32.DLL`, so an extra library `MSIMG32.LIB` needs to be linked to your program. It has been reported that `TransparentBlt` implementation on Windows 98 contains a resource leak, so it's not safe for long-time exposure. Anyway, we have enough reason to provide a platform-independent implementation here.

Simulating `TransparentBlt`

The critical step in implementing TransparentBlt is creating the mask bitmap from the color key. Monochrome DDB provides a very good support for creating the mask bitmap. Basically, when a color image is converted to a monochrome bit map, any pixel having a color matching the destination device background color is converted to 1 (white); all other pixels are converted to 0 (black). It could be a counterintuitive thing to remember. Just keep in mind that the default background color in a device context is white, which is made of all 1s.

The mask bitmap also needs an associated memory device context. Creating the mask bitmap and memory DC for each bitmap drawing could be costly for certain types of applications. [Listing 11-8](#) shows an implementation of TransparentBlt that has a helper class KDDBMask to manage the mask bitmap and memory device context. The G_TransparentBlt routine creates an instance of a KDDBMask class on the stack, creates the mask, and then uses it to draw the original bitmap transparently. In your own application, you may want to move the mask bitmap instance somewhere up to reduce creating it multiple times.

Listing 11-8 TransparentBlt Simulation

```
BOOL G_TransparentBlt(HDC hdcDest, int nDx0, int nDy0, int nDw, int nDh,
                      HDC hdcSrc, int nSx0, int nSy0, int nSw, int nSh,
                      UINT crTransparent)
{
    KDDBMask mask;

    mask.Create(hdcSrc, nSx0, nSy0, nSw, nSh, crTransparent);

    return mask.TransBlt(hdcDest, nDx0, nDy0, nDw, nDh,
                         hdcSrc, nSx0, nSy0, nSw, nSh);
}

class KDDBMask
{
    HDC    m_hMemDC;
    HBITMAP m_hMask;
    HBITMAP m_hOld;
    int    m_nMaskWidth;
    int    m_nMaskHeight;

    void Release(void)
    {
        if ( m_hMemDC )
        {
            SelectObject(m_hMemDC, m_hOld);
            DeleteObject(m_hMemDC);
            m_hMemDC = NULL;
            m_hOld = NULL;
        }

        if ( m_hMask )
        {
```

```
        DeleteObject(m_hMask); m_hMask = NULL;
    }
}

public:
KDDBMask()
{
    m_hMemDC = NULL;
    m_hMask = NULL;
    m_hOld = NULL;
}

~GDDBMask()
{
    Release();
}

BOOL Create(HDC hDC, int nX, int nY, int nWidth, int nHeight,
            UINT crTransparent);

BOOL ApplyMask(HDC HDC, int nX, int nY, int nWidth, int nHeight,
               DWORD Rop);

BOOL TransBlt(HDC hdcDest, int nDx0, int nDy0, int nDw, int nDh,
              HDC hdcSrc, int nSx0, int nSy0, int nSw, int nSh);

};

// Create a monochrome mask bitmap from a source DC
BOOL KDDBMask::Create(HDC hDC, int nX, int nY, int nWidth, int nHeight,
                      UINT crTransparent)
{
    Release();

    RECT rect = { nX, nY, nX + nWidth, nY + nHeight };
    LPtoDP(hDC, (POINT *) & rect, 2);
    m_nMaskWidth = abs(rect.right - rect.left);
    m_nMaskHeight = abs(rect.bottom - rect.top); // get real size

    // create monochrome mask bitmap, memory DC
    m_hMemDC = CreateCompatibleDC(hDC);
    m_hMask = CreateBitmap(m_nMaskWidth, m_nMaskHeight, 1, 1, NULL);
    m_hOld = (HBITMAP) SelectObject(m_hMemDC, m_hMask);

    COLORREF oldBk = SetBkColor(hDC, crTransparent); // map to 1, white
    BOOL rsIt = StretchBlt(m_hMemDC, 0, 0, m_nMaskWidth, m_nMaskHeight,
                           hDC, nX, nY, nWidth, nHeight, SRCCOPY);
    SetBkColor(hDC, oldBk);
```

```
return rslt;
}

BOOL KDDBMask::ApplyMask(HDC hDC, int nX, int nY,
    int nWidth, int nHeight, DWORD rop)
{
    COLORREF oldFore = SetTextColor(hDC, RGB(0, 0, 0)); // Black
    COLORREF oldBack = SetBkColor(hDC, RGB(255, 255, 255)); // White

    BOOL rslt = StretchBlt(hDC, nX, nY, nWidth, nHeight, m_hMemDC, 0, 0,
        m_nMaskWidth, m_nMaskHeight, rop);

    SetTextColor(hDC, oldFore);
    SetBkColor(hDC, oldBack);

    return rslt;
}

// D=D^S, D=D & Mask, D=D^S--> if (Mask==1) D else S
BOOL KDDBMask::TransBlt(HDC hdcDest, int nDx0, int nDy0, int nDw, int nDh,
    HDC hdcSrc, int nSx0, int nSy0, int nSw, int nSh)
{
    StretchBlt(hdcDest, nDx0, nDy0, nDw, nDh,
        hdcSrc, nSx0, nSy0, nSw, nSh, SRCINVERT); // D^S

    ApplyMask(hdcDest, nDx0, nDy0, nDw, nDh, SRCAND); // if M D^S else 0

    return StretchBlt(hdcDest, nDx0, nDy0, nDw, nDh,
        hdcSrc, nSx0, nSy0, nSw, nSh, SRCINVERT);
    // if trans D else S
}
```

The KDDBMask::Create method creates a monochrome bitmap from the source device context, based on a transparent color key. The method calculates the size of the bitmap by mapping the source rectangle to its device coordinate space, just in case the source device context is not using the default MM_TEXT mapping mode. The actual color-to-monochrome conversion is controlled by the background color in the source device context. The KDDBMask::TransBlt method uses our familiar three raster operations SRCINVERT, SRCAND, and SRCINVERT to achieve transparency.

11.3 TRANSPARENCY WITHOUT A MASK BITMAP

Almost all the ways to do a transparent bitmap display use a monochrome bitmap that serves as an on-off mask. Cursor and icon resources have a built-in mask bitmap. PlgBlt and MaskBlt use mask bitmaps. The only exception is TransparentBlt, which uses color keying. But again, our simulation falls back to using a mask bitmap.

The question now is, how can we achieve transparency without a mask? What if the mask is really hard to create? What if the mask consumes too many resources? What are the alternatives? We are going to look at several alternatives in this section.

Masking Using Geometric Shapes

The basic algorithm we are using to display a picture transparently under the control of a mask has three steps:

- XOR the picture on the destination surface.
- AND the mask on the destination surface.
- XOR the picture on the destination surface again.

After the three steps, the area corresponding to black (0) on the mask is replaced with the source picture, and the rest is unchanged. Note that the mask is used only once to clear the opaque area to black (0).

If the area formed by black pixels in the mask bitmap can be described using geometric shapes, we can use GDI area-fill commands to draw the mask, instead of having to create a mask bitmap. Here is an example of drawing an oval-shaped DIB without a mask bitmap:

```
BOOL OvalStretchDIBits(HDC hDC, int XDest, int YDest,
    int nDestWidth, int nDestHeight,
    int XSrc, int YSrc, int nSrcWidth, int nSrcHeight,
    const void *pBits, const BITMAPINFO *pBMI, UINT iUsage)
{
    StretchDIBits(hDC, XDest, YDest, nDestWidth, nDestHeight, XSsrc, YSrc,
        nSrcWidth, nSrcHeight, pBits, pBMI, iUsage, SRCINVERT);

    SaveDC(hDC);
    SelectObject(hDC, GetStockObject(BLACK_BRUSH));
    SelectObject(hDC, GetStockObject(BLACK_PEN));
    Ellipse(hDC, XDest, YDest, nDestWidth, nDestHeight);
    RestoreDC(hDC, -1);
```

```
    return StretchDIBits(hDC, XDest, YDest, nDestWidth, nDestHeight,
        XSrc, YSrc, nSrcWidth, nSrcHeight, pBits, pBMI,
        iUsage, SRCINVERT);
}
```

The routine uses StretchDIBits with SRCINVERT ROP to draw the source bitmap first, then uses the Ellipse function to draw a black ellipse, and the second Stretch DIBits with SRCINVERT ROP makes sure the image appears only in the area covered by the ellipse.

Using OvalStretchDIBits to display a nontrivial bitmap can cause flickering, because the pixels are actually inverted for a noticeable amount of time.

One way to reduce the flickering is to follow the icon color bitmap design, whose transparent pixels are normally black. In this case, once the destination opaque region is cleared with black, you can use SCRPAINT ROP to merge the source with the destination. This assumes that we can modify the source image to clear its transparent pixels to black, which is doable for a DDB or a DIB section selected into a memory DC. Routine OvalStretchBlt illustrates this idea.

```
BOOL OvalStretchBlt(HDC hDC, int XDest, int YDest,
    int nDestWidth, int nDestHeight, HDC hDCSrc, int XSrc, int YSrc,
    int nSrcWidth, int nSrcHeight)
{
    // Make the region outside the ellipse be BLACK in the source surface
    SaveDC(hDCSrc);
    BeginPath(hDCSrc);
    Rectangle(hDCSrc, XSrc, YSrc, XSrc+nSrcWidth+1, YSrc+nSrcHeight+1);
    Ellipse(hDCSrc, XSrc, YSrc, XSrc + nSrcWidth, YSrc + nSrcHeight);
    EndPath(hDCSrc);
    SelectObject(hDCSrc, GetStockObject(BLACK_BRUSH));
    SelectObject(hDCSrc, GetStockObject(BLACK_PEN));
    FillPath(hDCSrc);
    RestoreDC(hDCSrc, -1);

    // Draw a BLACK ellipse in the destination surface
    SaveDC(hDC);
    SelectObject(hDC, GetStockObject(BLACK_BRUSH));
    SelectObject(hDC, GetStockObject(BLACK_PEN));
    Ellipse(hDC, XDest, YDest, XDest + nDestWidth, YDest + nDestHeight);
    RestoreDC(hDC, -1);

    // Merge source surface to destination
    return StretchBlt(hDC, XDest, YDest, nDestWidth, nDestHeight,
        hDCSrc, XSrc, YSrc, nSrcWidth, nSrcHeight, SRCPAINT);
}
```

The first part of the routine uses path functions to draw the area outside the ellipse with a black brush. The pixels within the ellipse are kept untouched. Note that the result of this masking can be reused if the bitmap needs to be displayed multiple times. The second part, clearing destination surface, is the same. The third part changes ROP from SRCINVERT to SRCPAINT. We should expect to see less flickering, because only the pixels within the ellipse are changed from the original color to black and then to the source pixel. To completely remove flickering, render all drawings to an off-screen bitmap, and finally copy them to the screen.

Masking Using Clipping

If the masking shape is a simple geometric shape, a simple way to achieve a flicker-free transparent bitmap display is by using a clip region. The clip region is an underused GDI feature, mainly because the 16-bit GDI does not have good support for region.

Here is the routine to display a bitmap clipped to an oval shape using a single bit-block transfer call, with the help of a clipping region:

```
BOOL ClipOvalStretchDIBits(HDC hDC, int XDest, int YDest,
    int nDestWidth, int nDestHeight,
    int XSrc, int YSrc, int nSrcWidth, int nSrcHeight,
    const void *pBits, const BITMAPINFO *pBMI, UINT iUsage)
{
    RECT rect = { XDest, YDest, XDest + nDestWidth, YDest + nDestHeight };
    LPtoDP(hDC, (POINT *) & rect, 2);

    HRGN hRgn = CreateEllipticRgnIndirect(& rect);

    SaveDC(hDC);

    SelectClipRgn(hDC, hRgn);
    DeleteObject(hRgn);

    BOOL rslt = StretchDIBits(hDC, XDest, YDest, nDestWidth, nDestHeight,
        XSrc, YSrc, nSrcWidth, nSrcHeight, pBits, pBMI, iUsage, SRCCOPY);

    RestoreDC(hDC, -1);

    return rslt;
}
```

We may need to refresh our memory a little bit about region and clipping. The clipping region is defined in device coordinate space, instead of logical coordinate space. So the ClipOvalStretchDIBits has to first map the destination rectangle to the destination device context's device coordinate space, before calling CreateEllipticRgnIndirect to create a region. When a clip region is selected into a device context, GDI makes a copy of its contents, so we're free to delete the

region object immediately.

Prefabricating Image

In certain situations, a transparent bitmap is displayed only on surfaces with a uniform background color. For example, menu check marks are normally displayed on a menu background, which has a uniform color according to the current system settings. An efficient way to manage those images with special usage is prefabricating the final image before they are displayed.

Win32 API provides a very powerful function to help you do that.

```
HANDLE LoadImage(HINSTANCE hinst, LPCTSTR lpszName, UINT uType,  
    int cxDesired, int cyDesired, UINT fuLoad);
```

LoadImage can load cursors, icons, and bitmaps from resources or external files. For cursors and icons, you can specify the preferred size. For bitmaps, LoadImage can create either a DDB or a DIB section. Certain colors in an image can be mapped to a different color to form a prefabricated image. The first parameter specifies the module instance handle if loading from a resource, for which lpszName will be the resource name. The latter can also be an external file name, if the LR_LOADFROMFILE flag is in fuLoad. The uType parameter could be IMAGE_BITMAP, IMAGE_CURSOR, or IMAGE_ICON, for bitmap, cursor, or icon. The cxDesired and cyDesired pair is only for preferred cursor or icon size. The last parameter, fuLoad, controls how images are going to be transformed. For example, LR_CREATEDIBSection asks to create a DIB section, instead of a DDB for bitmaps. LR_LOADMAP3DCOLORS maps pixels with RGB(128, 128, 128) to COLOR_3DSADOWS, RGB(192, 192, 192) to COLOR_3DFACE, and RGB(223, 233, 223) to COLOR_3DLIGHT. LR_MONOCHROME loads the image in black/white. LR_TRANSPARENT maps the pixels matching the first pixel in the image to the default window color COLOR_WINDOW. LR_VGACOLOR uses true VGA colors in the bitmaps. For details, refer to MSDN documentation.

The most interesting thing here is the LR_TRANSPARENT flag. When this flag is on, LoadImage replaces pixels having the same color as the first pixel with the default window background color COLOR_WINDOW. So if the bitmap needs to be displayed on a background with COLOR_WINDOW as the color, displaying the whole bitmap with the simplest SRCCOPY ROP has the same effect as transparent displaying using a mask bitmap. But, you can take advantage of this feature only when the bitmap is palette-based and the background you want to display on has COLOR_WINDOW as the background. Why doesn't GDI provide a function to replace the transparent color with an arbitrary color?

The following code illustrates how to use LoadImage to load a mapped image and play a simple animation sequence. The object we are handling is a mosquito image from the DirectX SDK. Its animation sequence consists of three images with different wing and leg positions. When played in sequence at different locations, it generates quite a good flying mosquito effect.

```
void TestLoadImage(HDC hDC, HINSTANCE hInstance)
```

```
{  
HBITMAP hBitmap[3];  
  
for (int i=0; i<3; i++)  
    hBitmap[i] = (HBITMAP) LoadImage(hInstance,  
        MAKEINTRESOURCE(IDB_BITMAP1+i), IMAGE_BITMAP, 0, 0,  
        LR_LOADTRANSPARENT | LR_CREATEDIBSection );  
  
BITMAP bmp;  
GetObject(hBitmap[0], sizeof(bmp), & bmp);  
  
HDC hMemDC = CreateCompatibleDC(hDC);  
SelectObject(hDC, GetSysColorBrush(COLOR_WINDOW));  
  
int lastx = -1;  
int lasty = -1;  
  
HRGN hRgn = CreateRectRgn(0, 0, 0, 0);  
for (i=0; i<600; i++)  
{  
    SelectObject(hMemDC, hBitmap[i%3]);  
  
    int newx = i;  
    int newy = abs(200-i%400);  
  
    if ( lastx!=-1 )  
    {  
        SetRectRgn(hRgn, newx, newy, newx+bmp.bmWidth,  
            newy + bmp.bmHeight);  
        ExtSelectClipRgn(hDC, hRgn, RGN_DIFF);  
        PatBlt(hDC, lastx, lasty, bmp.bmWidth, bmp.bmHeight, PATCOPY);  
        SelectClipRgn(hDC, NULL);  
    }  
    BitBlt(hDC, newx, newy, bmp.bmWidth, bmp.bmHeight,  
        hMemDC, 0, 0, SRCCOPY);  
  
    lastx = newx; lasty = newy;  
}  
  
DeleteObject(hRgn);  
DeleteObject(hMemDC);  
DeleteObject(hBitmap[0]);  
DeleteObject(hBitmap[1]);  
DeleteObject(hBitmap[2]);  
}
```

The window is created with COLOR_WINDOW as the background color, and LoadImage replaces the transparent color (black) with it. So to display the image, we need to use only one BitBlt with SRCCOPY ROP. To make the animation work, we have to display one image, erase it, move to a new position and repeat displaying and erasing. The PatBlt call is used to erase the previous image. Note that, because we're on a clear background, we can just erase with the color COLOR_WINDOW. For a more complicated background, we need to save the background image and restore it. To reduce the flickering, we use clip region to move to the region that is going to be covered by the new image from the erasing area. For each scene, no screen pixel is changed twice. This generates a smooth simple animation.

[< BACK](#) [NEXT >](#)

11.4 ALPHA BLENDING

In the last few sections, we have discussed raster operations extensively. If you think deeply, most of the time the bitwise raster operations do not make any sense. What do you expect to get when you bitwisely logical-AND, OR, or XOR two pixels together? We manage to make sense of a few simple raster operation sequences to use them to display bitmaps transparently, to color monochrome bitmaps, to split the RGB channels of a color bitmap, or to gradually display a bitmap. Note that we normally use AND with a monochrome mask to mask off unwanted pixels; we use XOR to selectively combine the destination and source bitmap together. We never blindly combine two images together without knowing exactly what's going to happen. The trouble here is that bitwise raster operations on color images do not have much meaningful interpretation in the real world, except in the few cases we listed above.

The basic formula we use in a transparent bitmap display is " $(M \& D) | (\sim M \& S)$," which reads as, "if mask pixel is 1 (white), the operation result is the destination pixel, otherwise use the source pixel." This formula is defined using bitwise Boolean algebra semantics. If we change the formula to use arithmetic operations instead, it becomes " $M * D + (1 - M) * S$." This formula is the essence of alpha blending.

Alpha blending is an imaging technique that combines (blends) two images (source image and destination image) together using the weighted sum of the source and destination pixels. The weight on the source pixel is normally called the alpha value; the weight on the destination pixel is one minus alpha, where one is the largest color value. Alpha blending is defined on each color channel, instead of bitwisely.

An alpha of 0 means the source pixel is completely transparent, and 1 means completely opaque. Assuming we are using a 24-bpp or 32-bpp drawing surface, the conceptual formulae for alpha blending are:

```
Dst.red = Src.red * alpha + (1-alpha) * Dst.red ;
Dst.green = Src.green * alpha + (1-alpha) * Dst.green;
Dst.blue = Src.blue * alpha + (1-alpha) * Dst.blue ;
Dst.alpha = Src.alpha * alpha + (1-alpha) * Dst.alpha;
```

Alpha blending is a new GDI feature added in Windows 98 and Windows 2000. Only one structure and a single function are added to support it.

```
typedef struct _BLENDFUNCTION {
    BYTE BlendOp;
    BYTE BlendFlags;
    BYTE SourceConstantAlpha;
    BYTE AlphaFormat;
} BLENDFUNCTION;
```

```
BOOL AlphaBlend(HDC hdcDest, int nXOriginDest, int nYOriginDest,
                int nWidthDest, int nHeightDest,
                HDC hdcSrc, int nXOriginSrc, int nYOriginSrc,
                int nWidthSrc, int nHeightSrc,
```

```
BLENDFUNCTION blendFunction);
```

The prototype for AlphaBlend looks very much like StretchBlt. The first five parameters specify the destination device context and a rectangle on the destination surface in logical coordinates. Similarly, the next five parameters specify the source device context and a rectangle on the source surface in logical coordinates. The normal restriction for a source device context applies here—that is, the source rectangle must be in the source device context; and the source device context can't have shearing and rotation to confuse GDI. Note that using the source device context rules out directly using DIB with AlphaBlend.

The last parameter, blendFunction, holds a BLENDFUNCTION structure passed by value, replacing the raster operation code in StretchBlt. The BLENDFUNCTION structure controls how the source and destination bitmaps should be blended. Its BlendOp field specifies the source blend operation, but the only choice supported is AC_SRC_OVER, which places the source image over the destination image based on source alpha. OpenGL's alpha blending support provides other choices like a constant color source. The next field, BlendFlags, must be zero: another future opportunity. The last field, AlphaFormat, can have two choices: 0 for constant alpha value, or AC_SRC_ALPHA for per-pixel alpha channel.

If the AlphaFormat field is 0, all pixels in the source bitmap use the same constant alpha value specified in the SourceConstantAlpha field. The value is actually between 0 and 255, instead of 0 and 1. Here 0 means total transparency and 255 means total opaqueness. The destination pixels use 255-SourceConstantAlpha as their alpha value. The actual formulae used for alpha blending are changed to:

```
Dst.red = Round((Src.red * SourceConstantAlpha +
                 (255-SourceConstantAlpha) * Dst.red ))/255);
Dst.green = Round((Src.green * SourceConstantAlpha +
                   (255-SourceConstantAlpha) * Dst.green))/255);
Dst.blue = Round((Src.blue * SourceConstantAlpha +
                  (255-SourceConstantAlpha) * Dst.blue ))/255);
Dst.alpha = Round((Src.alpha * SourceConstantAlpha +
                   (255-SourceConstantAlpha) * Dst.alpha))/255);
```

If the AlphaFormat field is AC_SRC_ALPHA, the source device surface must have a per-pixel alpha channel. That is, it must be a physical device context in 32-bpp mode, or a memory device context with a 32-bpp DDB or DIB section selected. In all these cases, each source pixel has four 8-bit channels: red, green, blue, and alpha. The alpha channel of each pixel will be used together with the SourceConstantAlpha field to blend the source and destination together. The following formulae are actually used in doing the calculations.

```
Tmp.red = Round((Src.red * SourceConstantAlpha)/255);
Tmp.green = Round((Src.green * SourceConstantAlpha)/255);
Tmp.blue = Round((Src.blue * SourceConstantAlpha)/255);
Tmp.alpha = Round((Src.alpha * SourceConstantAlpha)/255);

beta = 255 - Tmp.alpha;
```

```
Dst.red = Tmp.red + Round((beta * Dst.red )/255);
Dst.green = Tmp.green + Round((beta * Dst.green)/255);
Dst.blue = Tmp.blue + Round((beta * Dst.blue )/255);
Dst.alpha = Tmp.alpha + Round((beta * Dst.alpna)/255);
```

Looking at these formulae carefully, the per-pixel source alpha value Src.alpha is applied only to the destination pixels, not to the source pixels. GDI assumes that the source bitmap has already been premultiplied with the source alpha. This is apparently designed to cater to game programming, where each scene is normally pregenerated by an external tool or the result of other components. An image with a premultiplied alpha is just like a transparent bitmap with a black background (color bitmap in an icon); it saves display time at the cost of flexibility. If SourceConstantAlpha is 255, the temporary variable Tmp does not need to be calculated.

If performance is on your mind, you may worry about the per-pixel division by 255. On an Intel CPU, true division can still take dozens of clock cycles, so it's considered extremely slow. You should trust the GDI implementation and a modern compiler to be smart enough to convert division by constant to a multiplication and a shift.

Simple Constant Alpha Blending

Alpha blending using a constant alpha is the easiest. It does not require having a 32-bpp source surface, because the alpha value can be expressed in the BLENDFUNCTION structure. Here is a simple example of blending a few solid color blocks together:

```
void SimpleConstantAlphaBlend(HDC hDC)
{
    const int size = 100;

    for (int i=0; i<3; i++)
    {
        RECT rect = { i*(size+10) + 20, 20+size/3,
                      i*(size+10) + 20 + size, 20+size/3 + size };

        COLORREF Color[]={ RGB(0xFF,0,0), RGB(0,0xFF,0), RGB(0,0,0xFF) };

        HBRUSH hBrush = CreateSolidBrush(Color[i]);
        FillRect(hDC, &rect, hBrush); // three original solid rectangles
        DeleteObject(hBrush);

        BLENDFUNCTION blend = { AC_SRC_OVER, 0, 255/2, 0 }; // alpha 0.5

        AlphaBlend(hDC, 360+((3-i)%3)*size/3, 20+i*size/3, size, size,
                   hDC, i*(size+10)+20, 20+size/3, size, size, blend);
    }
}
```

This example uses a single device context for both source and destination. Three solid red, green, and blue rectangles are drawn first; then AlphaBlend is called to blend each of them with the window background. A constant alpha of 0.5 is used for each call.

The window background starts as a solid white region, with color RGB(0xFF, 0xFF, 0xFF). When the first red rectangle is blended with it, every pixel is now RGB(0xFF, 0x80, 0x80). When the second green rectangle is blended, the pixels in the intersection of red and green rectangles are RGB(0x80, 0xBF, 0x40). When the third blue rectangle is blended, the pixels in the intersection of all three rectangles have color RGB(0x40, 0x60, 0x90). This

value may not be what you expected, but it's calculated according to the alpha blending definition. RGB(0x40, 0x60, 0x90) is $\text{RGB}(0xFF, 0xFF, 0xFF) * 0.125 + \text{RGB}(0xFF, 0, 0) * 0.125 + \text{RGB}(0, 0xFF, 0) * 0.25 + \text{RGB}(0, 0, 0xFF) * 0.5$. [Figure 11-7](#) shows the display result.

Figure 11-7. Constant alpha blending of color rectangles.



Fade In, Fade Out of Bitmaps

In [Section 11.2](#) we showed a routine to gradually fade in a bitmap using raster operations. Alpha blending provides a new way of displaying bitmaps in multiple steps. Here is a new routine to fade in a bitmap using constant alpha blending:

```
BOOL AlphaFade(HDC hDCDst, int XDst, int YDst, int nDstW, int nDstH,
                HDC hDCSrc, int XSrc, int YSrc, int nSrcW, int nSrcH)
{
    for (int i=5; i>=1; i - )
    {
        // Alpha      1/5, 1/4, 1/3, 1/2, 1/1
        // Accumulates to 1/5, 2/5, 3/5, 4/5, 5/5
        BLENDFUNCTION blend = { AC_SRC_OVER, 0, 255 / i , 0 };

        if ( !AlphaBlend(hDCDst, XDst, YDst, nDstW, nDstH,
                          hDCSrc, XSrc, YSrc, nSrcW, nSrcH,
                          blend) )
            return FALSE;
    }

    return TRUE;
}
```

The AlphaFade routine blends a source image in five steps into the destination surface, with alpha being 1/5, 1/4, 1/3, 1/2, and 1/1. After the first drawing call, part of the source image is already in the destination, so the accumulated ratios of the source bitmap are actually 1/5, 2/5, 3/5, 4/5 and finally 5/5.

Fading out a bitmap can use the same routine. If you want to fade to a bitmap, the source bitmap should be the final bitmap. If you want to fade to a solid color surface, create a solid color surface and pass to AlphaFade. Now, you should complain, why doesn't GDI allow a constant color as a source image?

Layered Window

Windows 98/2000 introduces a new extended window style, the layered window style. When the layered window flag, WS_EX_LAYERED is on, rendering to the window is cached in a bitmap of the size of the window, instead of written directly to the screen. It can then be blended together with the whole screen display using alpha blending. An application can even set a key color for a layered window. Any pixel having the same color as the color key will be transparent; that is, the pixel on the window under it will show through. When a layered window is uncovered, no repainting is needed on the application side. GDI just needs to redisplay the cached window bitmap to the screen. If implemented properly, a layered window provides new visual effects and improves system performance, at the cost of extra cache bitmaps.

The WS_EX_LAYERED style bit can either be included in CreateWindowEx call, or set later using SetWindowLong. Once a layered window is created or set up, SetLayeredWindowAttributes can be called to set a constant alpha value for the window and an optional transparent color key.

```
BOOL SetLayeredWindowAttributes(HWND hWnd, COLORREF crKey, BYTE bAlpha,  
    DWORD dwFlags);
```

The hWnd is the window handle of a window with a WS_EX_LAYERED style flag. The dwFlags parameter can have one or both of LWA_COLORKEY and LWA_ALPHA. If a LWA_COLORKEY flag is used, the crKey parameter is the transparent color key. If a LWA_ALPHA flag is used, the bAlpha parameter is the constant source alpha value. Only top-level windows can be layered windows.

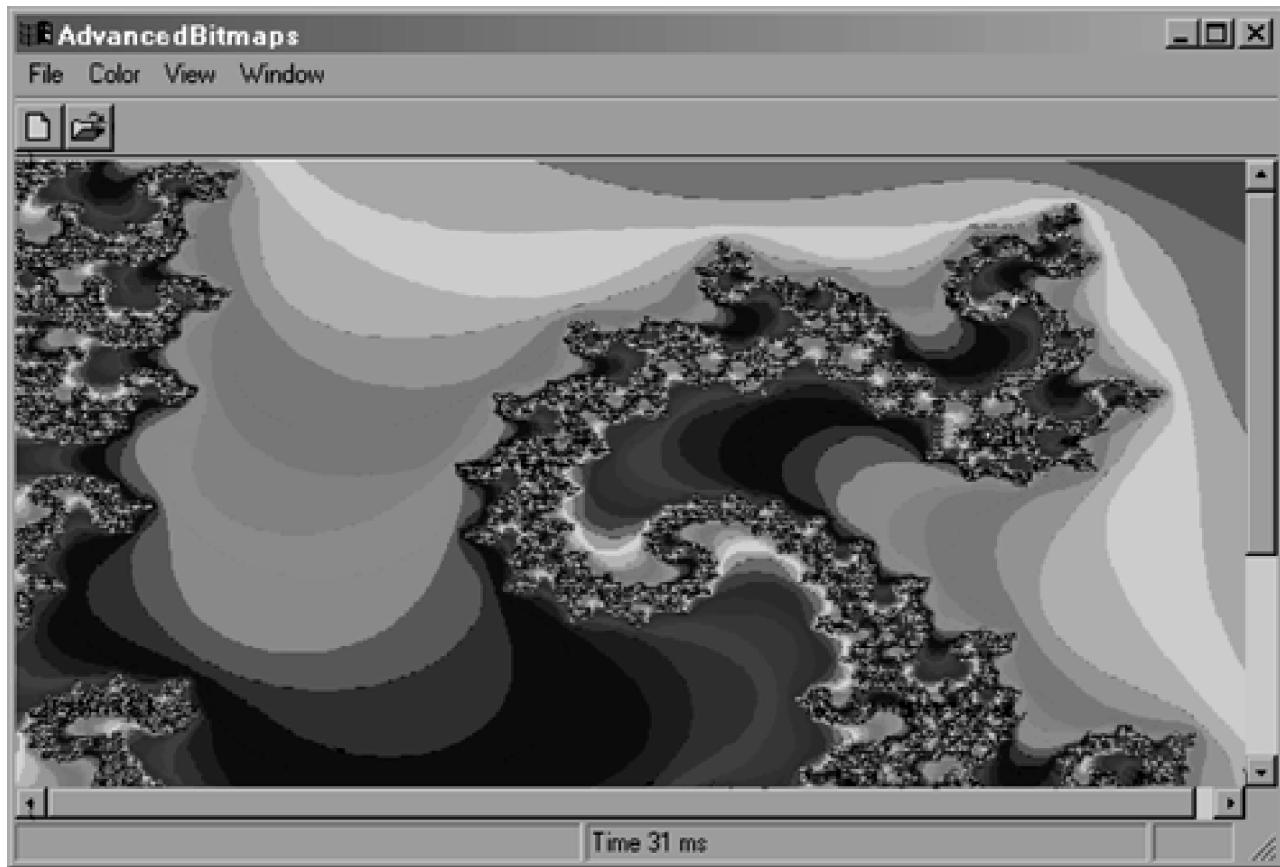
Here is an example of setting up a layered window in a window procedure:

```
switch ( uMsg )  
{  
    case WM_CREATE:  
        m_hWnd = hWnd;  
        SetWindowLong(m_hWnd, GWL_EXSTYLE,  
            GetWindowLong(m_hWnd, GWL_EXSTYLE) | WS_EX_LAYERED);  
        SetLayeredWindowAttributes(m_hWnd, RGB(0, 0, 1), 0xC0,  
            LWA_ALPHA | LWA_COLORKEY );  
        return 0;  
    ...  
}
```

In the code shown above, GetWindowLong is used to retrieve the current extended style flag, which is combined together with WS_EX_LAYERED and set back. The call to SetLayeredWindowAttributes sets the alpha to be 0.75 (0xC0/255) and color key to be RGB(0, 0, 1), an unusual color very close to true black. This code changes the window display fundamentally. Almost all display within the window, including the child window, is now semitransparent, although noticeably slower. But when a menu or a popup dialog box gets displayed, it's still totally

opaque. [Figure 11-8](#) shows a sample screen with a DIB bitmap blended with an MSVC IDE source code display. Note [Figure 11-8](#) is part of a whole-screen capture. If you capture only the window, the cached bitmap will be returned, instead of the actual display on screen.

Figure 11-8. Layered window: alpha blending between windows.



As you can imagine with new technology, a layered window is generated much more slowly than a normal window. You would not appreciate menus being displayed as opaque images, or windows not getting repainted properly.

Alpha Channel: AirBrush

Each of the examples shown above uses a constant alpha value that is applied to every pixel of the source bitmap. A constant alpha has limited usage. For example, it can't even achieve the transparent bitmap display that is achievable using a mask bitmap. A mask bitmap can be seen as a single-bit-per-pixel alpha channel, which is separated from the main bitmap. Let's now take a look at several uses of an alpha channel.

Although AlphaBlend supports either a DDB or DIB section in a memory DC, only a DIB section can be relied on if you want to use an alpha channel. The reason is that, in a non-32-bpp display mode, a 32-bpp DDB is not compatible with screen DC. In a 32-bit DIB section, each pixel is stored in four bytes. The first three bytes are normally the blue, green, and red channels, and the last byte is the alpha channel. Alpha Blend is the only function that reads and writes an alpha channel, so GDI is not helping a lot in setting up the alpha channel. When a per-pixel alpha is used, Alpha Blend assumes that the alpha is premultiplied to the source bitmap. So to use an alpha channel, we have to access the DIB section pixels directly.

Modern image-editing software normally provides different styles of brushes (not GDI brushes) that can be used to

draw big dots and thick lines. A brush in imaging software is defined by its shape, color, orientation, hardness, and other fancy attributes. A commonly used brush may be a round color brush, with 50% hardness, which determines the speed with which pixels on the brush change from a solid color in the center to totally transparent colors around the perimeter. When a dot or a line is drawn using such a brush, its edge blends smoothly with the background without showing any jagged edges, as is normal with GDI line drawing. These brushes are normally called airbrushes.

With alpha channel, we can easily implement airbrushes. [Listing 11-9](#) shows a KAirBrush class implemented using a DIB section.

Listing 11-9 AirBrush Class

```
class KAirBrush
{
    HBITMAP m_hBrush;
    HDC    m_hMemDC;
    HBITMAP m_hOld;
    int    m_nWidth;
    int    m_nHeight;

    void Release(void)
    {
        SelectObject(m_hMemDC, m_hOld);
        DeleteObject(m_hMemDC);
        DeleteObject(m_hBrush);

        m_hOld = NULL; m_hMemDC = NULL; m_hBrush = NULL;
    }

public:
    KAirBrush()
    {
        m_hBrush = NULL;
        m_hMemDC = NULL;
        m_hOld  = NULL;
    }

    ~KAirBrush()
    {
        Release();
    }

    void Create(int width, int height, COLORREF color);
    void Apply(HDC hDC, int x, int y);
};

void KAirBrush::Apply(HDC hDC, int x, int y)
{
```

```
BLENDFUNCTION blend = { AC_SRC_OVER, 0, 255, AC_SRC_ALPHA };

AlphaBlend(hDC, x-m_nWidth/2, y-m_nHeight/2, m_nWidth, m_nHeight,
           m_hMemDC, 0, 0, m_nWidth, m_nHeight, blend);
}

void KAirBrush::Create(int width, int height, COLORREF color)
{
    Release();

    BYTE * pBits;
    BITMAPINFO Bmi = { { sizeof(BITMAPINFOHEADER), width, height,
                         1, 32, BI_RGB } };
    m_hBrush = CreateDIBSection(NULL, &Bmi, DIB_RGB_COLORS,
                               (void **) &pBits, NULL, NULL);
    m_hMemDC = CreateCompatibleDC(NULL);
    m_hOld = (HBITMAP) SelectObject(m_hMemDC, m_hBrush);
    m_nWidth = width;
    m_nHeight = height;

    // color circle, on white background
    {
        PatBlt(m_hMemDC, 0, 0, width, height, WHITENESS);

        HBRUSH hBrush = CreateSolidBrush(color);
        SelectObject(m_hMemDC, hBrush);
        SelectObject(m_hMemDC, GetStockObject(NULL_PEN));

        Ellipse(m_hMemDC, 0, 0, width, height);

        SelectObject(m_hMemDC, GetStockObject(WHITE_BRUSH));
        DeleteObject(hBrush);
    }

    BYTE * pPixel = pBits;

    // compute alpha channel and premultiply it
    for (int y=0; y<height; y++)
        for (int x=0; x<width; x++, pPixel+=4)
    {
        // distance to center normalized to [0..255]
        int dis = (int) ( sqrt( (x-width/2) * (x-width/2) +
                               (y-height/2) * (y-height/2) ) *
                           255 / (max(width, height)/2) );

        BYTE alpha = (BYTE) max(min(255-dis, 255), 0);

        pPixel[0] = pPixel[0] * alpha / 255; // Blue
        pPixel[1] = pPixel[1] * alpha / 255; // Green
    }
}
```

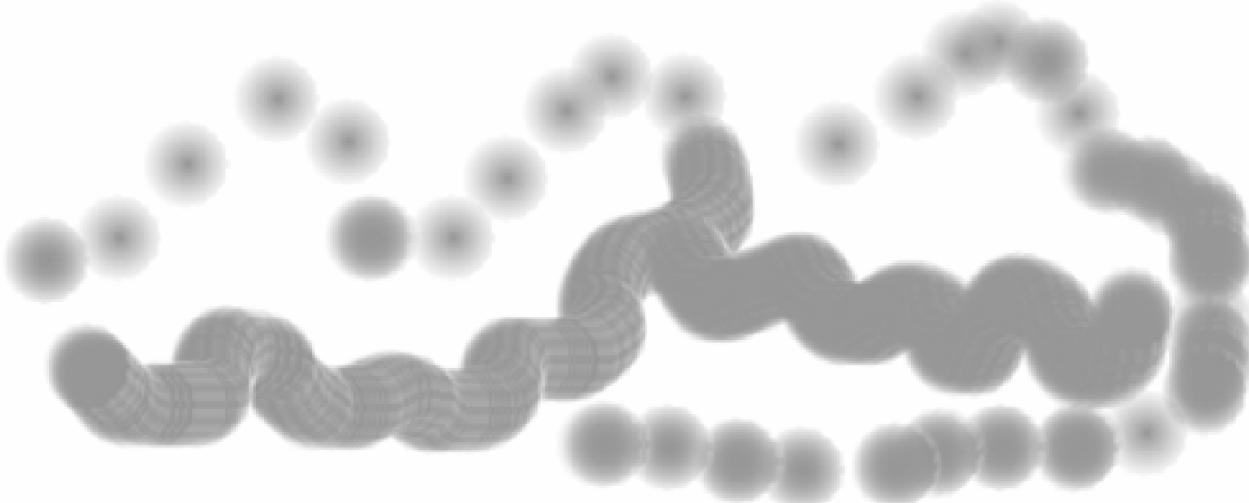
```
pPixel[2] = pPixel[2] * alpha / 255; // Red  
pPixel[3] = alpha;           // Alpha  
}  
}
```

The KAirBrush class uses a DIB section to store an airbrush, so we have a bitmap handle, memory DC, original bitmap handle, and brush dimension as its member variables. The KAirBrush::Create method creates the DIB section for an airbrush when given its dimension and color specifier. It creates a 32-bpp DIB section, and a memory DC in which the DIB section is selected. A white background and solid color circle are then drawn. Now we have a round-shaped brush with 100% hardness on a white background. The code that follows calculates the alpha channel and premultiplies it to the RGB value. For this the code accesses each pixel in the DIB section sequentially, calculating its distance from the center of the circle, maps it to an alpha value, premultiplies the alpha to RGB values, and stores the alpha in the alpha channel. Now we have a 32-bpp DIB section with the alpha channel premultiplied.

The KAirBrush::Apply method simply draws the brush centered around (x, y). Note that the constant alpha value is set to be 255 here, because we need only the alpha channel. If an application supports a pressure-sensitive tablet, the constant alpha value can be used to gradually change the brush along a line according to the brush or distance on the line.

Using the KAirBrush class is quite simple. Make sure an instance of KAirBrush is created—for example, during the initialization of a view. Toolbar buttons may be linked to code that changes the color or shape of the brush. When certain mouse events happen, apply the brush according to the mouse position. Here is a sample code fragment. [Figure 11-9](#) shows a sample display result.

Figure 11-9. Dots and lines drawn using airbrush.



```
switch ( uMsg )  
{  
case WM_CREATE:  
    m_brush.Create(32, 32, RGB(0, 0xFF, 0));  
    return 0;
```

```
case WM_LBUTTONDOWN:  
    wParam = MK_LBUTTON; // fall through  
  
case WM_MOUSEMOVE:  
    if ( wParam & MK_LBUTTON )  
    {  
        m_brush.Apply(m_hDCBitmap, LOWORD(lParam), HIWORD(lParam));  
        Refresh(LOWORD(lParam), HIWORD(lParam));  
    }  
    return 0;  
...  
}
```

The same technique can be applied to draw antialiased geometric shapes, or to blend multiple images together. For drawing antialiased shapes like lines, polygons, and circles, set alpha to be 255 for all inner pixels, 0 for all outside pixels, and set alpha for edge pixels to represent the level of coverage. For example, to draw a 45-degree line, certain pixels on the line are half covered by the line, so their alpha should be 127. The actual calculation can be quite complicated. One easy, but costly, way to do that is by creating a monochrome bitmap n -by- n the original size of the image, where an enlarged version of the geometric shape is drawn. The pixel sum of each n -by- n block is then converted to the alpha channel.

Simulating Alpha Blending

Like other nicer GDI features, alpha blending is not supported on all Win32 platforms, which restricts its usage. If you want to use alpha blending on real end-user applications, you have to simulate alpha blending yourself. Here is our try.

This time, we're not implementing the full features of the AlphaBlend function. Instead we just implement alpha blending between two 32-bpp DIBs. [Listing 11-10](#) shows the AlphaBlend3232 function.

Listing 11-10 Alpha Blending between Two 32-bpp DIBs

```
// Calculate DIB pixel offset  
inline int GetOffset(BITMAPINFO * pBMI, int x, int y)  
{  
    if ( pBMI->bmiHeader.biHeight > 0 ) // for bottom up, reflect y  
        y = pBMI->bmiHeader.biHeight - 1 - y;  
  
    return ( pBMI->bmiHeader.biWidth * pBMI->bmiHeader.biBitCount + 31 )  
        / 32 * 4 * y +  
        ( pBMI->bmiHeader.biBitCount / 8 ) * x;  
}  
  
// Alpha Blending between two 32-bpp DIBs  
BOOL AlphaBlend3232(BITMAPINFO * pBMIDst, BYTE * pBitsDst,  
                     int dx, int dy, int w, int h,
```

```
BITMAPINFO * pBMISrc, BYTE * pBitsSrc, int sx, int sy,
BLENDFUNCTION blend)

{
int alpha = blend.SourceConstantAlpha; // constant alpha
int beta = 255 - alpha; // constant beta
int format;

if (blend.AlphaFormat==0)
format = 0;
else if (alpha==255)
format = 1;
else
format = 2;

for (int j=0; j<h; j++)
{
BYTE * D = pBitsDst + GetOffset(pBMIDst, dx, j + dy);
BYTE * S = pBitsSrc + GetOffset(pBMISrc, sx, j + sy);

int i;

switch (format)
{
case 0: // constant alpha only
for (i=0; i<w; i++)
{
D[0] = (S[0] * alpha + beta * D[0] + 127) / 255;
D[1] = (S[1] * alpha + beta * D[1] + 127) / 255;
D[2] = (S[2] * alpha + beta * D[2] + 127) / 255;
D[3] = (S[3] * alpha + beta * D[3] + 127) / 255;
D += 4; S += 4;
}
break;

case 1: // alpha channel only
for (i=0; i<w; i++)
{
beta = 255 - S[3];
D[0] = S[0] + (beta * D[0] + 127) / 255;
D[1] = S[1] + (beta * D[1] + 127) / 255;
D[2] = S[2] + (beta * D[2] + 127) / 255;
D[3] = S[3] + (beta * D[3] + 127) / 255;
D += 4; S += 4;
}
break;

case 2: // both constant alpha and alpha channel
for (i=0; i<w; i++)
{
```

```
beta = 255 - ( S[3] * alpha + 127 ) / 255;  
  
D[0] = ( S[0] * alpha + beta * D[0] + 127 ) / 255;  
D[1] = ( S[1] * alpha + beta * D[1] + 127 ) / 255;  
D[2] = ( S[2] * alpha + beta * D[2] + 127 ) / 255;  
D[3] = ( S[3] * alpha + beta * D[3] + 127 ) / 255;  
D += 4; S += 4;  
}  
}  
}  
  
return TRUE;  
}
```

The destination for AlphaBlend3232 is a rectangle in a 32-bpp DIB; its source is a rectangle in a 32-bpp DIB of the same size. So we have a pixel address to both destination and source, and no scaling is supported. The routine divides alpha blending into three cases: constant alpha, alpha channel only, and both constant alpha and alpha channel. It then loops through each pixel in the destination rectangle to blend it with the source pixel.

For essential functionality like alpha blending, individual routines should be added for blending between different combinations of source and destination surface. For example, a routine AlphaBlend1632 should be written to accept a 16-bit destination and a 32-bit source; another routine AlphaBlend824 should be written to accept an 8-bpp palette-based destination that requires a color lookup table and a 24-bit source that does not support alpha channel.

[< BACK](#) [NEXT >](#)

11.5 SUMMARY

This chapter focuses on how to combine several operands together to form a new pixel when displaying an image. Under this big topic, we examined in great depth ternary raster operations, quaternary raster operations, transparent bitmap drawing using different techniques, and alpha blending, together with mapping a bitmap to a parallelogram. We discussed the basic designs behind raster operations, how to take them apart and analyze them, and deriving a raster operation to solve practical problems. All new fancy GDI bitmap-displaying functions like MaskBlt, PlgBlt, TransparentBlt, and AlphaBlend are provided with sample usages and routines to simulate them for Win32 platforms that do not support them.

After [Chapter 10](#) on GDI bitmap formats and [Chapter 11](#) on using GDI-provided functions, what's left is how to access a bitmap pixel array directly and implement features not provided by GDI, or to improve on GDI's implementation. The next chapter covers direct pixel access and its application in image processing.

Further Reading

Platform SDK contains a complete list of ternary raster operations. Search for "Ternary Raster Operations" in MSDN. *DDK Knowledge Base* has a good article (Q74508) on interpreting the encoding of lower-order words in a 32-bit ternary raster operation code. More detailed information can be found in Win95/98 DDK, under \display\samples\planar directory. File COMMENT.BLT describes history and technical notes concerning BitBLT, including exact bit-level encoding and assembly code.

Alpha blending is widely used in OpenGL, and for sure Windows graphics programming is going to use it more frequently. Chapter 7 of *OpenGL Programming Guide* discusses how blending is designed and used in OpenGL. OpenGL supports far more types of blending operations.

Sample Program

[Chapter 11](#) has one program which demonstrates all the topics covered here (see [Table 11-3](#)).

Table 11-3. Sample Program for [Chapter 11](#)

Directory	Description
Samples\Chapt_11\AdvBitmap	Demonstrates ROP chart, cursor display using raster operations, sprite animation, alpha blending, bitmap masking, bitmap fading, PlgBlt, airbrush, etc. Demo pages are accessible under the test menu or after opening a BMP file.

Chapter 12. Image Processing Using Windows Bitmaps

The benefit of device-independent bitmaps and DIB sections is that applications can directly access their pixel arrays and color tables. Often this direct access is a critical technique to implement features not provided by GDI, or to achieve improved performance not feasible using GDI calls.

So far, we have seen both cases. When we talked about simulating PlgBlt, we used GDI's pixel access API GetPixel and SetPixel. We found out that it's much slower than GDI's Windows 2000 implementation. Each individual GetPixel and SetPixel call has the overhead of parameter validation, user-mode to kernel-mode switching, color translation, etc. GetPixel even creates a temporary bitmap to call the internal implementation of BitBlt to perform the task. No wonder it's very slow. When using alpha blending, GDI does not provide adequate support for setting up the alpha channel in a 32-bpp bitmap, or to premultiply the RGB channels with the alpha channel. So you have to access the pixel array directly to perform these functions.

In this chapter, we are going to look at the direct manipulation of DIBs and DIB sections to implement various bitmap-handling algorithms. This chapter will cover direct pixel access, bitmap affine transformation using direct pixel access, bitmap color transformation, bitmap pixel transformation, and image processing using spatial filters.

12.1 GENERIC PIXEL ACCESS

As a first step, we need several generic functions to access individual pixels within a DIB or a DIB section. A DIB section can be treated in the same way as a DIB, once you have a complete BITMAPINFO and a pointer to its pixel array. So basically, we need functions very similar to the GDI functions GetPixel and SetPixel.

When working on a device-independent bitmap, accessing individual pixels within a compressed bitmap is quite hard. While reading a pixel from an RLE compressed bitmap is possible, although it could be very slow, writing a pixel to an RLE compressed bitmap is almost impossible. The reason is that new pixels may require expanding a scan line if it's not the same as its neighbors. So we're assuming all compressed bitmaps are uncompressed first; this includes RLE compressed bitmaps and JPEG or PNG compressed bitmaps.

Another thing to notice is that GetPixel and SetPixel deal with COLORREF, which could be RGB values, palette index, or palette RGB values. We want our basic pixel access routine to use the same pixel format as the bitmap it's accessing, which means color indexes for palette-based bitmaps, and 16-, 24-, or 32-bit RGB or RGB values for non-palette-based bitmaps. Routines dealing with COLORREF can be built on top of these basic routines.

[Listing 12-1](#) shows two methods added to our KDIB class: GetPixelIndex and SetPixelIndex. GetPixelIndex returns the color data for a pixel at a specific location. The data will be 1 bit for 1-bpp bitmaps, 2 bits for 2-bpp bitmaps, and so on. SetPixel Index is just the opposite; it replaces the color data for a pixel at a specific location with new color data.

Boundary checking is provided so that out-of-boundary access will be rejected. These illegal accesses are good candidates for GPFs (general protection faults). The routines return error values if the coordinates are out of bounds. After parameter validation, the address of the first pixel on the scan line is calculated according to the y-coordinate, based on the bitmap origin address and the address difference between two scan lines. These two values are precalculated to take care of the top-down and bottom-up scan-line order within a DIB. For example, for the default bottom-up DIB, the origin of the DIB is at the bottom of the bitmap data, and the scan-line delta is negative.

Real pixel access is done according to the pixel format. For 1-bpp, 2-bpp, and 4-bpp bitmaps, each pixel occupies a fraction of a byte. So the code needs to figure out the number of bits to shift and the right mask to use. Accessing an 8-bpp bitmap is the easiest; just read a single byte. Accessing a 16-bpp bitmap reads two bytes. The caller needs to figure out how to convert the 16-bpp pixel data to an RGB format.

Accessing a 24-bpp bitmap is a little tricky, because we can't access a 24-bpp pixel as a DWORD and then mask off the highest 8 bits. Although you do see published programs doing that, it's the wrong thing to do. For example, if you create a 64-by-64-pixel, 24-bpp DIB section, its pixel array's size will be $64 \times 64 \times 3 = 12$ KB. On an Intel CPU, exactly three pages of memory will be allocated. The offset of the last pixel in the bitmap is 0x2FFD. If you read it as a DWORD, the CPU needs to read

one byte past the 12-KB allocated region, thus creating a good chance of triggering a GPF. I once spent lots of time chasing exactly the same problem in a release build driver, because the debug build allocates extra bytes for tagging.

Listing 12-1 Generic DIB Pixel Access Routines

```
const BYTE Shift1bpp[] = { 7, 6, 5, 4, 3, 2, 1, 0 };
const BYTE Mask1bpp [] = { 0x7F,0xBF,0xDF,0xEF, 0xF7,0xFB,0xFD,0xFE };

const BYTE Shift2bpp[] = { 6, 4, 2, 0 };
const BYTE Mask2bpp [] = { ~0xC0, ~0x30, ~0x0C, ~0x03 };
const BYTE Shift4bpp[] = { 4, 0 };
const BYTE Mask4bpp [] = { ~0xF0, ~0x0F };

DWORD KDIB::GetPixelIndex(int x, int y) const
{
    if ( (x<0) || (x>=m_nWidth) )
        return -1;

    if ( (y<0) || (y>=m_nHeight) )
        return -1;

    BYTE * pPixel = m_pOrigin + y * m_nDelta;

    switch ( m_nImageFormat )
    {
        case DIB_1BPP:
            return ( pPixel[x/8] >> Shift1bpp[x%8] ) & 0x01;

        case DIB_2BPP:
            return ( pPixel[x/4] >> Shift2bpp[x%4] ) & 0x03;

        case DIB_4BPP:
            return ( pPixel[x/2] >> Shift4bpp[x%4] ) & 0x0F;

        case DIB_8BPP:
            return pPixel[x];

        case DIB_16RGB555:
        case DIB_16RGB565:
            return ((WORD *)pPixel)[x];

        case DIB_24RGB888:
            pPixel += x * 3;
            return (pPixel[0]) | (pPixel[1] << 8) | (pPixel[2] << 16);
    }
}
```

```
case DIB_32RGB888:  
case DIB_32RGBA8888:  
    return ((DWORD *)pPixel)[x];  
}  
  
return -1;  
}  
  
BOOL KDIB::SetPixelIndex(int x, int y, DWORD index)  
{  
if ( (x<0) || (x>=m_nWidth) )  
    return FALSE;  
  
if ( (y<0) || (y>=m_nHeight) )  
    return FALSE;  
  
BYTE * pPixel = m_pOrigin + y * m_nDelta;  
  
switch ( m_nImageFormat )  
{  
case DIB_1BPP:  
    pPixel[x/8] = (BYTE) ( ( pPixel[x/8] & Mask1bpp[x%8] ) |  
        ( (index & 1) << Shift1bpp[x%8] ) );  
    break;  
  
case DIB_2BPP:  
    pPixel[x/4] = (BYTE) ( ( pPixel[x/4] & Mask2bpp[x%4] ) |  
        ( (index & 3) << Shift2bpp[x%4] ) );  
    break;  
  
case DIB_4BPP:  
    pPixel[x/2] = (BYTE) ( ( pPixel[x/2] & Mask4bpp[x%2] ) |  
        ( (index & 15) << Shift4bpp[x%2] ) );  
    break;  
  
case DIB_8BPP:  
    pPixel[x] = (BYTE) index;  
    break;  
  
case DIB_16RGB555:  
case DIB_16RGB565:  
    ((WORD *)pPixel)[x] = (WORD) index;  
    break;
```

```
case DIB_24RGB888:  
    ((RGBTRIPLE *)pPixel)[x] = * ((RGBTRIPLE *) & index);  
    break;  
  
case DIB_32RGB888:  
case DIB_32RGBA8888:  
    ((DWORD *)pPixel)[x] = index;  
    break;  
default:  
    return FALSE;  
}  
  
return TRUE;  
}
```

The SetPixelIndex method has the same structure as GetPixelIndex. For 1-bpp, 2-bpp, and 4-bpp bitmaps, setting a pixel means to mask off the bits occupied by it first and then combine with the new data. For 24-bpp, we're casting the pointer to the RGBTRIPLE pointer and copying data as a RGBTRIPLE structure, so that the compiler will generate code to copy exactly three bytes.

If you're manipulating the arrangement of pixels in a bitmap, which needs random access to pixels—for example, in bitmap rotation, flipping, and copying between bitmaps having the same format—GetPixelIndex and SetPixelIndex are just what you want. They also work very well for non-palette-based bitmaps. If you want to set a pixel in an 8-bpp bitmap to red color, you need to consult the color table first. We will cover that later.

[< BACK](#) [NEXT >](#)

12.2 BITMAP AFFINE TRANSFORMATION

As a sample use for the GetPixelIndex and SetPixelIndex methods developed in the last section, let's implement a generic bitmap affine transformation algorithm. Actually, we already had the basic code working when we did the simulation for PlgBlt.

[Listing 12-2](#) shows two routines: KDIB::PlgBlt and KDIB::TransformBitmp. The KDIB::PlgBlt routine transforms a rectangle within a DIB to a parallelogram within another DIB. Three points on the destination surface define the parallelogram. Like the GDI function of the same name, the routine supports all 2D affine transformations, including translation, reflection, rotation, and shearing. KDIB::PlgBlt has the same structure as our simulation of the GDI PlgBlt function, except that now we use GetPixelIndex and SetPixelIndex to access pixels, in place of GDI's slow GetPixel and SetPixel functions.

Listing 12-2 Generic Bitmap Affine Transformation

```
BOOL KDIB::PlgBlt(const POINT * pPoint, KDIB * pSrc,
                   int nXSrc, int nYSrc, int nWidth, int nHeight)
{
    KReverseAffine map(pPoint);
    map.Setup(nXSrc, nYSrc, nWidth, nHeight);

    for (int dy=map.miny; dy<=map.maxy; dy++)
        for (int dx=map.minx; dx<=map maxx; dx++)
    {
        float sx, sy;
        map.Map(dx, dy, sx, sy);

        if ( (sx>=nXSrc) && (sx<(nXSrc+nWidth)) )
            if ( (sy>=nYSrc) && (sy<(nYSrc+nHeight)) )
                SetPixelIndex(dx, dy, pSrc->GetPixelIndex( (int)sx, (int)sy));
    }

    return TRUE;
}

HBITMAP KDIB::TransformBitmap(XFORM * xm, COLORREF crBack)
{
    int x0, y0, x1, y1, x2, y2, x3, y3;

    Map(xm, 0,      0,      x0, y0); // 0  1
    Map(xm, m_nWidth, 0,      x1, y1); //
    Map(xm, 0,      m_nHeight, x2, y2); // 2  3
    Map(xm, m_nWidth, m_nHeight, x3, y3);
```

```
int xmin, xmax;
int ymin, ymax;

minmax(x0, x1, x2, x3, xmin, xmax);
minmax(y0, y1, y2, y3, ymin, ymax);

int destwidth = xmax - xmin;
int destheight = ymax - ymin;

KBitmapInfo dest;
dest.SetFormat(destwidth, destheight,
    m_pBMI->bmiHeader.biBitCount, m_pBMI->bmiHeader.biCompression);

BYTE * pBits;
HBITMAP hBitmap = CreateDIBSection(NULL, dest.GetBMI(),
    DIB_RGB_COLORS, (void **) & pBits, NULL, NULL);

if ( hBitmap==NULL )
    return NULL;
{
    HDC hMemDC = CreateCompatibleDC(NULL);
    HGDIOBJ hOld = SelectObject(hMemDC, hBitmap);

    HBRUSH hBrush = CreateSolidBrush(crBack);
    RECT rect = { 0, 0, destwidth, destheight };
    FillRect(hMemDC, & rect, hBrush);
    DeleteObject(hBrush);

    SelectObject(hMemDC, hOld);
    DeleteObject(hMemDC);
}

KDIB destDIB;
destDIB.AttachDIB(dest.GetBMI(), pBits, 0);

POINT P[3] = { { x0-xmin, y0-ymin }, { x1-xmin, y1-ymin },
    { x2-xmin, y2-ymin } };

destDIB.PlgBlt(P, this, 0, 0, m_nWidth, m_nHeight);

return hBitmap;
}
```

The KDIB::PlgBlt assumes that a destination bitmap of the right size is already created, and it has the same pixel format as the source bitmap. When transforming a bitmap, the new bitmap to be generated is normally of a different size. Rotation and shearing may leave corners that should be filled by a background color, because it's out of boundary for the source image. The KDIB::TransformBitmap routine is actually responsible for setting the stage for KDIB::PlgBlt. It accepts a transformation matrix and background color. Based on the current DIB format, it calculates the exact size for the transformed bitmap, and creates a DIB section of that size with the current bitmap format. The

newly created DIB section is then cleared with the background color, before being passed to KDIB::PIgBlt for actual transformation.

[Figure 12-1](#) shows a picture of flowers rotated 15 degrees, using the KDIB::PIg Blt routine.

Figure 12-1. Bitmap rotation using KDIB::PIgBlt.



On a low-end Pentium 200-MHz machine, the KDIB::PIgBlt routine is able to rotate a 1024-by-768, 24-bpp image in about 1.062 seconds, or 0.7 megapixels per second. As a comparison, if you change the calls from GetPixelIndex/SetPixelIndex to GDI's GetPixel/SetPixel calls, after creating the proper memory device context handles, the time it takes goes up to 16.9 seconds, or 0.044 megapixels per second. This proves that direct pixel access is much faster than GDI's GetPixel/SetPixel functions. Considering that the KDIB::PIgBlt routine uses floating-point arithmetic for coordinate calculations, the performance ratio between a direct-pixel-access-based routine vs. GetPixel/SetPixel-based routine can even be bigger than 17 times, as we're seeing in this example.

12.3 FAST SPECIALIZED BITMAP TRANSFORMER

When performance is important, generic algorithms can be specialized and optimized for the ultimate performance goal. This is especially important for image-processing algorithms such as image transformation.

If we are willing to write specialized routines for different DIB formats, we can in-line a pixel access statement for a particular DIB format into the PltBlt routine, to save the overhead of function calls in the inner loop, testing for bitmap format, doing duplicate pixel address calculation, etc. Another area to improve is the floating-point usage in mapping destination bitmap coordinates to source bitmap coordinates. The default implementation of converting floating point to integer is very slow, because a function call is involved.

[Listing 12-3](#) shows an integer-only bitmap transformation routine that handles only 24-bpp images. The PlgBlt24 routine starts with the same code as the PlgBlt routine to set up a reverse transformation from the destination back to the source. The KReverseAffine::Setup routine returns the bounding box in the destination surface, which is then checked with the destination bitmap dimensions to make sure the values are valid. The source bitmap bounding box is converted to fixed-point numbers by multiplying with a constant-value FACTOR of 65,536. The transformation matrix is also converted in the same way. The fixed-point number format we use here uses a 16-bit integer and a 16-bit fractional value. It's large enough to handle quite large bitmap sizes, and also accurate enough.

For each scan line, the pixel address is calculated only once and stored in the pD Pixel variable, which is later incremented by three bytes (for 24-bpp bitmaps) for each pixel. The cost per pixel for destination pixel is therefore reduced to a single addition. The calculation for the source pixel corresponding to the current destination pixel is in-lined, converted to fixed-point arithmetic, and calculated incrementally. The initial values, sx, sy, are calculated outside the inner loop and incremented by the eM11 and eM12 factors in the transformation matrix. They are then compared with a source rectangle bounding box in fixed-point to make sure that only the pixel in the source bitmap is fetched. To get the actual source pixel address, fixed-point numbers sx and sy need to be converted to integer numbers; it's simply a division by the constant FACTOR. The compiler is smart enough to implement it by using a shift operation. The final pixel is again done using RGBTRIPLE copy.

[Listing 12-3 Optimized 24-bit DIB Transformer](#)

```
BOOL KDIB::PlgBlt24(const POINT * pPoint, KDIB * pSrc,
    int nXSrc, int nYSrc, int nWidth, int nHeight)
{
    // factor to change FLOAT to fixed point
    const int FACTOR = 65536;
```

```
// generate reverse transformation from destination to source
KReverseAffine map(pPoint);

map.Setup(nXSrc, nYSrc, nWidth, nHeight);

// make sure within destination bitmap dimension
if ( map.minx < 0 )      map.minx = 0;
if ( map.maxx > m_nWidth ) map.maxx = m_nWidth;
if ( map.miny < 0 )      map.miny = 0;
if ( map.maxy > m_nHeight ) map.maxy = m_nHeight;

// source rectangle in fixed point
int sminx = nXSrc * FACTOR;
int sminy = nYSrc * FACTOR;
int smaxx = ( nXSrc + nWidth ) * FACTOR;
int smaxy = ( nYSrc + nHeight ) * FACTOR;

// transformation matrix in fixed point
int m11 = (int) (map.m_xm.eM11 * FACTOR);
int m12 = (int) (map.m_xm.eM12 * FACTOR);
int m21 = (int) (map.m_xm.eM21 * FACTOR);
int m22 = (int) (map.m_xm.eM22 * FACTOR);
int mdx = (int) (map.m_xm.eDx * FACTOR);
int mdy = (int) (map.m_xm.eDy * FACTOR);

BYTE * SOrigin = pSrc->m_pOrigin;
int SDelta = pSrc->m_nDelta;

// in destination bitmap, scan from first to last scan line
for (int dy=map.miny; dy<map.maxy; dy++)
{
    // precalculate destination pixel address for first pixel
    BYTE * pDPixel = m_pOrigin + dy * m_nDelta + map.minx * 3;

    // source pixel address for the first pixel
    int sx = m11 * map.minx + m21 * dy + mdx;
    int sy = m12 * map.minx + m22 * dy + mdy;

    // go through each pixel on the scan line
    for (int dx=map.minx; dx<map.maxx; dx++, pDPixel+=3,
        sx+=m11, sy+=m12)
        if ( (sx>=sminx) && (sx<smaxx) )
            if ( (sy>=sminy) && (sy<smaxy) )
            {
                // source pixel address
```

```
BYTE * pSPixel = SOrigin + (sy/FACTOR) * SDelta;  
  
// copy three bytes  
* ((RGBTRIPLE *)pDPixel) =  
    ((RGBTRIPLE *)pSPixel)[sx/FACTOR];  
}  
}  
  
return TRUE;  
}
```

For the sample 1024-by-768 24-bpp bitmap we tested with KDIB::PlgBlt, our improved routine PlgBlt24 rotates it in 172 ms, or about 4.36 megapixels per second, a 520% improvement of the GetPixelIndex/SetPixelIndex-based generic solution. Compared with the GetPixel/SetPixel based solution, PlgBlt24 is 100 times faster. There are still areas in which PlgBlt24 can be improved. For example, instead of comparing with the source bitmap bounding box for every pixel, the intersections of the destination scan line with the source bounding box should be precalculated, so that the per-pixel test can be totally avoided.

[< BACK](#) [NEXT >](#)

12.4 BITMAP COLOR TRANSFORMATION

There are numerous algorithms that need to transform color for each pixel in the bitmap in a certain way. A color transformation is applied to each pixel independently and without a global context. Sample usage for this group of algorithms includes converting color bitmaps to grayscale, gamma correction, color space conversion, adjusting bitmap hue, lightness, or saturation, etc.

Such bitmap color transformation algorithms have the same pattern. If the bitmap has a color table, each entry in the color table needs to be transformed. Otherwise, we have to scan through each pixel in the pixel array and apply a transformation to each of them. An easy way to implement them would be to use a generic algorithm and a function pointer parameter. Each color transformation is a simple static function. To transform a bitmap with a given color transformation, just call the generic algorithm with the specific color transformation routine a parameter. A similar way is to define an abstract color transformation class, which has a virtual member function that does the color transformation.

Performance is normally considered critical for bitmap-handling algorithms, so calling a function or a virtual function for every pixel in a huge bitmap is normally unacceptable. We certainly do not want to repeat the same code over and over again, nor would we like to use macro-based programming. So the only alternative is a template function.

We would like to define these algorithms on the KDIB class. But the trouble with a template function is that our available C++ compiler does not support a template member function. So we have to use a static template function that accepts a pointer to a KDIB instance. To make things worse, our C++ compiler does not support a friend template function, which forces us to change some private members to be public members.

[Listing 12-4](#) shows a template-based DIB color transformation algorithm.

Listing 12-4 Template for Bitmap Color Transformation Algorithms

```
template <class Dummy>
bool ColorTransform(KDIB * dib, Dummy map)
{
    // OS/2 DIB color table: 1,4,8-bpp, include RLE compression
    if ( dib->m_pRGBTRIPLE )
    {
        for (int i=0; i<dib->m_nClrUsed; i++)
            map(dib->m_pRGBTRIPLE[i].rgbtRed,
                 dib->m_pRGBTRIPLE[i].rgbtGreen,
                 dib->m_pRGBTRIPLE[i].rgbtBlue);

        return true;
    }

    // Windows DIB color table: 1,2,4,8-bpp, include RLE compression
    if ( dib->m_pRGBQUAD )
    {
```

```
for (int i=0; i<dib->m_nClrUsed; i++)
    map(dib->m_pRGBQUAD[i].rgbRed,
        dib->m_pRGBQUAD[i].rgbGreen,
        dib->m_pRGBQUAD[i].rgbBlue);

    return true;
}

for (int y=0; y<dib->m_nHeight; y++)
{
    int width = dib->m_nWidth;
    unsigned char * pBuffer = (unsigned char *) dib->m_pBits +
        dib->m_nBPS * y;

    switch ( dib->m_nImageFormat )
    {
        case DIB_16RGB555: // 15-bit RGB color image, 5-5-5
            for (; width>0; width--)
            {
                BYTE red = ( (* (WORD *) pBuffer) & 0x7C00 ) >> 7;
                BYTE green = ( (* (WORD *) pBuffer) & 0x03E0 ) >> 2;
                BYTE blue = ( (* (WORD *) pBuffer) & 0x001F ) << 3;

                map( red, green, blue );
                * ( WORD * ) pBuffer = ( ( red >> 3 ) << 10 ) |
                    ( ( green >> 3 ) << 5 ) |
                    ( blue >> 3 );

                pBuffer += 2;
            }
            break;

        case DIB_16RGB565: // 16-bit RGB color image, 5-6-5
            for (; width>0; width--)
            {
                BYTE red = ( (* (WORD *) pBuffer) & 0xF800 ) >> 8;
                BYTE green = ( (* (WORD *) pBuffer) & 0x07E0 ) >> 3;
                BYTE blue = ( (* (WORD *) pBuffer) & 0x001F ) << 3;

                map( red, green, blue );

                * ( WORD * ) pBuffer = ( ( red >> 3 ) << 11 ) |
                    ( ( green >> 2 ) << 5 ) |
                    ( blue >> 3 );

                pBuffer += 2;
            }
            break;
    }
}
```

```
case DIB_24RGB888: // 24-bpp RGB
    for (; width>0; width--)
    {
        map( pBuffer[2], pBuffer[1], pBuffer[0] );
        pBuffer += 3;
    }
    break;

case DIB_32RGBA8888: // 32-bpp RGBA
case DIB_32RGB888: // 32-bpp RGB
    for (; width>0; width--)
    {
        map( pBuffer[2], pBuffer[1], pBuffer[0] );
        pBuffer += 4;
    }
    break;
default:
    return false;
}
}

return true;
}
```

The ColorTransform function accepts two parameters: a pointer to a KDIB instance and a function pointer. It's strange to pass a function pointer a template function without specifying the function prototype. But apparently, this method is supported and used by STL. The first part of the function handles the color table in the OS/2 BMP file format; each of the RGBTRIPLE structures is handled by calling the color transformation function through the map parameters. The color transformation routine accepts three reference parameters for the red, green, and blue channels, and returns the transformed color using the same variable. The code handling Windows color table is very similar, except now the RGBQUAD structure needs to be mapped. This part of the code handles all palette-based DIB formats, including RLE compressed and uncompressed formats.

The remaining code handles 16-bit high color bitmaps, 24-bit true color bitmaps, and 32-bit true color with alpha bitmaps. The logical order of the pixel array is not important here, so the code just goes through each pixel in the order they appear. For 16-bit bitmaps, two common formats are supported. The code needs to extract the RGB channels, convert them to 8 bits each, call the color transformation routine, and pack the results back into a 16-bit WORD. A 24-bit bitmap is very easy to handle. For a 32-bit bitmap, its alpha channel is untouched.

Any other rare DIB formats, like JPEG or PNG compressed bitmaps, or bitmaps with unusual bit fields, are not supported by the currently implemented ColorTransform routine.

Converting Bitmaps to Grayscale

The commonly used formula for converting color in RGB space to grayscale is

$$\text{gray} = 0.299 * \text{red} + 0.587 * \text{green} + 0.114 * \text{blue}$$

For computer implementation, we would like to implement the conversion without any floating-point computation. Based on the ColorTransform template, here is our RGB-bitmap to grayscale-bitmap conversion method.

```
// 0.299 * red + 0.587 * green + 0.114 * blue
inline void MaptoGray(BYTE & red, BYTE & green, BYTE & blue)
{
    red = (red * 77 + green * 150 + blue * 29 + 128) / 256;
    green = red;
    blue = red;
}

class KImage : public KDIB
{
public:
    bool ToGreyScale(void);
    ...
};

bool KImage::ToGreyScale(void)
{
    return ColorTransform(this, MaptoGray);
}
```

A new class KImage is derived from the KDIB class to encapsulate image-processing algorithms developed in this chapter. The KImage class has no extra member variables. Its only method shown here is ToGreyScale. More methods will be added later in this chapter.

The KImage::ToGreyScale method transforms the current color DIB to a grayscale DIB. It simply calls the ColorTransform template function with the MaptoGray routine as the color transformation routine. MaptoGray uses integer operation to calculate the lightness of a color and assign it to all three RGB channels.

In a debug build, MaptoGray is compiled as a subroutine and its pointer is passed to ColorTransform. In a release build, all calls to MaptoGray are in-lined to get the best performance.

Gamma Correction

Image displaying suffers from photometric distortions caused by the nonlinear response of display devices to lightness. The photometric response of a displaying device is known as the gamma response characteristic. Display monitors for different operating systems use different gammas. When an image prepared on a Macintosh screen is displayed on a PC screen, it normally looks too dark. On the other hand, an image downloaded from a PC server to a Macintosh machine may look too bright. To compensate for these differences, the gamma of the image needs to be corrected.

Gamma correction is normally done in the three RGB channels independently. Three arrays of 256 bytes each are precalculated and passed to either the software gamma transformer or the hardware display card. Each of the three arrays is applied to one of the channels.

Gamma correction can be easily implemented using the ColorTransform template function.

```
BYTE redGammaRamp[256];
BYTE greenGammaRamp[256];
BYTE blueGammaRamp[256];

inline void MapGamma(BYTE & red, BYTE & green, BYTE & blue)
{
    red = redGammaRamp[red];
    green = greenGammaRamp[green];
    blue = blueGammaRamp[blue];
}

BYTE gamma(double g, int index)
{
    return min(255, (int) ( (255.0 * pow(index/255.0, 1.0/g)) + 0.5 ) );
}

bool KImage::GammaCorrect(double redgamma, double greengamma,
                           double bluegamma)
{
    for (int i=0; i<256; i++)
    {
        redGammaRamp[i] = gamma( redgamma, i);
        greenGammaRamp[i] = gamma(greengamma, i);
        blueGammaRamp[i] = gamma( bluegamma, i);
    }

    return ColorTransform(this, MapGamma);
}
```

The user level routine provided here is KDIB::GammaCorrect. It accepts three independent gamma values, which normally range between 0.2 and 5.0. The routine pre-calculates three gamma ramps for each of the RGB channels, according to the definition of gamma. It then calls ColorTransform with the MapGamma routine as the color transformer. MapGamma does a simple table lookup for each pixel.

A gamma correction with gamma equal to 1.0 is an identity color transformation. Gamma less than one makes a picture look “darker,” and gamma larger than one makes a picture look “lighter.”

If you apply a gamma correction of 2.2 to images prepared on a Macintosh, they will look exactly as they originally appeared on their creator’s screen. [Figure 12-2](#) shows a tiger picture with the wrong gamma and its gamma-corrected version.

Figure 12-2. Gamma correction.



The table lookup mechanism implemented by MapGamma can also be used to adjust color according to other criteria. Actually, the only condition is that the RGB channels are independent of each other. For example, you can define a red Gamma Ramp to reduce the red channel by 10%, while leaving the other two as identity transformations.

Win32 GDI supports setting a graphics device's gamma ramp if the hardware and device driver supports downloadable gamma ramps. The related function is SetDeviceGammaRamp, which is part of ICM 2.0. DirectDraw also supports gamma ramps through the IDirectDrawGammaControl interface. Newer PC hardware should all support downloadable gamma ramps.

[< BACK](#) [NEXT >](#)

12.5 BITMAP PIXEL TRANSFORMATION

The template-based bitmap color-transformation algorithm presented in the last section walks through every color in a bitmap, not every pixel. The difference is mainly with palette-based bitmaps, for which the color-transformation algorithm just needs to walk through all entries in the color lookup table, instead of every pixel in the bitmap.

There is another large class of image-processing algorithms that need to process every pixel. For example, a histogram algorithm needs to sample every pixel to find the true distribution of colors in the bitmap. To split a color bitmap into different channels, we would prefer the result to be a separate grayscale image, which can then be operated on. The actual channel-splitting algorithms used by image editors need to walk through every pixel.

In this section, we will develop a generic bitmap pixel-transformation algorithm, and show several sample usages. This time, we choose to use function pointer, virtual function, and abstract base class, instead of template. A template-based solution as shown in the last section offers very good performance, at the cost of multiple copies of binary code for each instance of the template. Another restriction imposed by limited compiler support is that the operator that instantiated the template is a simple function. We can't use a C++ class, which encapsulates both data and code with the template function. For example, we have to use global variables for the gamma correction implementation.

Generic Pixel Transformation Class

[Listing 12-5](#) shows the KPixelMapper, an abstract base class from which different pixel-transformation algorithms can be derived. The class is designed to handle the transformation of a single pixel and a single scan line.

The processing methods in the class are all virtual methods. The MapRGB method is a pure virtual function, which handles a single pixel in its RGB form. The KPixelMapper class does not implement it, because a derived class is supposed to provide the actual implementation for whatever algorithm is intended. The MapIndex handles a pixel in its color-table index form. We do have a default implementation that converts color index to RGB values and calls MapRGB. The Map1bpp, Map2bpp,..., Map32bpp methods handle all the commonly seen DIB scan-line formats. Their default implementation loops through each pixel in a scan line and calls MapRGB or Map Index with each pixel.

These routines are provided as virtual functions, so that a derived class can customize their implementation. For example, a derived class may decide that a 24-bpp image is really important, so it overrides Map24bpp to in-line every call to its Map RGB implementation for maximum performance. Note that the innermost loop is most critical to performance. [Listing 12-5](#) shows two of the scan-line handlers, for handling 1-bpp and 24-bpp scan lines.

Both MapRGB and MapIndex return a Boolean value to indicate whether the parameters passed through references are changed. Callers may decide whether to modify the original pixel based on this return value.

[Listing 12-5 Abstract Base Class KPixelMapper](#)

```
// Abstract class for mapping single pixel and scan line
class KPixelMapper
{
    BYTE * m_pColor; // point to color table BGR...
    int   m_nSize;   // size of an entry in the table 3 or 4
    int   m_nCount;  // number of entries in the table
```

```
// return true if data changed
virtual bool MapRGB(BYTE & red, BYTE & green, BYTE & blue) = 0;

// return true if data changed
virtual bool MapIndex(BYTE & index)
{
    MapRGB(m_pColor[index*m_nSize+2],
           m_pColor[index*m_nSize+1],
           m_pColor[index*m_nSize]);

    return false;
}

public:

KPixelMapper(void)
{
    m_pColor = NULL;
    m_nSize = 0;
    m_nCount = 0;
}

virtual ~KPixelMapper()
{
}

void SetColorTable(BYTE * pColor, int nEntrySize, int nCount)
{
    m_pColor = pColor;
    m_nSize = nEntrySize;
    m_nCount = nCount;
}

virtual bool StartLine(int line)
{
    return true;
}

virtual void Map1bpp(BYTE * pBuffer, int width);
virtual void Map2bpp(BYTE * pBuffer, int width);
virtual void Map4bpp(BYTE * pBuffer, int width);
virtual void Map8bpp(BYTE * pBuffer, int width);
virtual void Map555(BYTE * pBuffer, int width);
virtual void Map565(BYTE * pBuffer, int width);
virtual void Map24bpp(BYTE * pBuffer, int width);
virtual void Map32bpp(BYTE * pBuffer, int width);
};
```

```
void KPixelMapper::Map1bpp(BYTE * pBuffer, int width)
{
    BYTE mask = 0x80;
    int shift = 7;

    for (; width>0; width--)
    {
        BYTE index = ( ( pBuffer[0] & mask ) >> shift ) & 0x1;

        if ( MapIndex(index) )
            pBuffer[0] = ( pBuffer[0] & ~mask ) ||
                (( index & 0x0F ) << shift);

        mask >>= 1; shift -= 1;

        if ( mask==0 )
        {
            pBuffer++; mask = 0x80; shift = 7;
        }
    }
}

void KPixelMapper::Map24bpp(BYTE * pBuffer, int width)
{
    for (; width>0; width--)
    {
        MapRGB( pBuffer[2], pBuffer[1], pBuffer[0] );
        pBuffer += 3;
    }
}
```

To use the KPixelMapper class to transform a DIB, a new function KImage ::PixelTransform is added, which needs to set up an instance of the KPixelMapper class, and feed the scan lines to it. Here is the PixelTransform function—a very straightforward piece of code.

```
bool KDIB::PixelTransform(KPixelMapper & map)
{
    if ( m_pRGBTRIPLE )
        map.SetColorTable((BYTE *) m_pRGBTRIPLE,
                          sizeof(RGBTRIPLE), m_nClrUsed);
    else if ( m_pRGBQUAD )
        map.SetColorTable((BYTE *) m_pRGBQUAD,
                          sizeof(RGBQUAD), m_nClrUsed);

    for (int y=0; y<m_nHeight; y++)
    {
        unsigned char * pBuffer = (unsigned char *) m_pBits + m_nBPS * y;
```

```
if ( ! map.StartLine(y) )
break;

switch ( m_nImageFormat )
{
case DIB_1BPP:
map.Map1bpp(pBuffer, m_nWidth);
break;

case DIB_2BPP:
map.Map2bpp(pBuffer, m_nWidth);
break;

case DIB_4BPP:
map.Map4bpp(pBuffer, m_nWidth);
break;

case DIB_8BPP:
map.Map8bpp(pBuffer, m_nWidth);
break;

case DIB_16RGB555: // 15-bit RGB color image, 5-5-5
map.Map555(pBuffer, m_nWidth);
break;

case DIB_16RGB565: // 16-bit RGB color image, 5-6-5
map.Map565(pBuffer, m_nWidth);
break;

case DIB_24RGB888: // 24-bpp RGB
map.Map24bpp(pBuffer, m_nWidth);
break;

case DIB_32RGBA8888: // 32-bpp RGBA
case DIB_32RGB888: // 32-bpp RGB
map.Map32bpp(pBuffer, m_nWidth);
break;
default:
return false;
}

}

return true;
}
```

We have a clear separation of duties here; a derived class of the KPixelMapper class handles single-pixel transformation, the KPixelMapper class itself handles scan-line transformation, and the KImage::PixelTransform method handles transforming a whole DIB. If you support a bitmap format other than a BMP format, you just need to write your own PixelTransform routine. If you want a new image-processing algorithm that falls in the

pixel-transformation category, just implement a derived class of KPixelMapper.

Generic Channel-Splitting Class

Now let's do something useful—implementing color channel algorithm splitting in the right way—that is, generating grayscale per-channel images from a color image. The basic idea is to map an RGB pixel to a byte, which will be stored in an 8-bpp bitmap using a grayscale color table. As always, we want to make the implementation as generic as possible, so that different types of channel splitting can be implemented. This time, we need a simple function to control the channel splitting class, which is defined as Operator in [Listing 12-6](#).

Listing 12-6 Image Channel Splitting Based on KPixelMapper

```
typedef BYTE (* Operator)(BYTE red, BYTE green, BYTE blue);

// Generic channel splitting class derived from KPixelMapper
// Controlled by an operator function passed to KChannel::Split
class KChannel : public KPixelMapper
{
    Operator m_Operator;
    int     m_nBPS;

    BYTE *  m_pBits;
    BYTE *  m_pPixel;

    // return true if data changed
    virtual bool MapRGB(BYTE & red, BYTE & green, BYTE & blue)
    {
        * m_pPixel ++ = m_Operator(red, green, blue);

        return false;
    }

    virtual bool StartLine(int line)
    {
        m_pPixel = m_pBits + line * m_nBPS; // first pixel of a scan line

        return true;
    }
}

public:

BITMAPINFO * Split(CDIB & dib, Operator oper);
};

BITMAPINFO * KChannel::Split(KImage & dib, Operator oper)
{
    m_Operator = oper;
    m_nBPS    = (dib.GetWidth() + 3) / 4 * 4; // 8-bpp scan-line size
    int headsize = sizeof(BITMAPINFOHEADER) + 256 * sizeof(RGBQUAD);
```

```
BITMAPINFO * pNewDIB = (BITMAPINFO *) new BYTE
[headsize + m_nBPS * abs(dib.GetHeight())];

memset(pNewDIB, 0, headsize);

pNewDIB->bmiHeader.biSize      = sizeof(BITMAPINFOHEADER);
pNewDIB->bmiHeader.biWidth     = dib.GetWidth();
pNewDIB->bmiHeader.biHeight    = dib.GetHeight();
pNewDIB->bmiHeader.biPlanes    = 1;
pNewDIB->bmiHeader.biBitCount  = 8;
pNewDIB->bmiHeader.biCompression = BI_RGB;

for (int c=0; c<256; c++)
{
    pNewDIB->bmiColors[c].rgbRed  = c;
    pNewDIB->bmiColors[c].rgbGreen = c;
    pNewDIB->bmiColors[c].rgbBlue = c;
}

m_pBits = (BYTE*) & pNewDIB->bmiColors[256];

if ( pNewDIB==NULL )
    return NULL;

dib.PixelTransform(* this);

return pNewDIB;
}

BITMAPINFO * KDIB::SplitChannel(Operator oper)
{
    KChannel channel;

    return channel.Split(* this, oper);
}
```

The KChannel class is derived from the KPixelMapper class. Its main entrance is the Split method, which accepts a reference to a DIB and an Operator. The Split method creates a 256-color DIB with the same dimensions as the source bitmap, fills a grayscale palette, and sets the address of the pixel array to a member variable `m_pBits`, which will be used to locate each pixel. The KDIB::PixelTransform method is then called to scan through each pixel in the bitmap, which finally calls KChannel::MapRGB. Our MapRGB implementation calls the operator to map RGB to a byte and store it as the pixel value for the new 256-color bitmap being created. The StartLine method will be called at the start of every scan line, so the code can properly set the destination scan line's starting point.

This class handles only one channel at a time. Multiple channels can either be handled one by one, or with a new implementation that creates multiple 8-bpp DIBs and accepts a new type of operator returning multiple results.

Sample Channel Splitting

Using the KChannel class is really easy; you just need to supply an operator. Here are a few sample operators for commonly used RGB, CMYK, and HLS channel splitting:

```
// Red in RGB channel splitting
inline BYTE TakeRed(BYTE red, BYTE green, BYTE blue)
{
    return red;
}

// Black in KCMY channel splitting
inline BYTE TakeK(BYTE red, BYTE green, BYTE blue)
{
    // min ( 255-red, 255-green, 255-blue)
    if ( red < green )
        if ( green < blue )
            return 255 - blue;
        else
            return 255 - green;
    else
        return 255 - red;
}

// Hue in HLS channel splitting
inline BYTE TakeH(BYTE red, BYTE green, BYTE blue)
{
    KColor color(red, green, blue);
    color.ToHLS();

    return (BYTE) (color.hue * 255 / 360);
}
```

Here is how these operators are used in the KDIBView class, a class for a MDI child window displaying a DIB:

```
LRESULT KDIBView::OnCommand(int nId)
{
    switch( nId )
    {
        case IDM_COLOR_SPLITRGB:
            CreateNewView(m_DIB.SplitChannel(TakeRed), "Red Channel");
            CreateNewView(m_DIB.SplitChannel(TakeGreen), "Green Channel");
            CreateNewView(m_DIB.SplitChannel(TakeBlue), "Blue Channel");
            return 0;

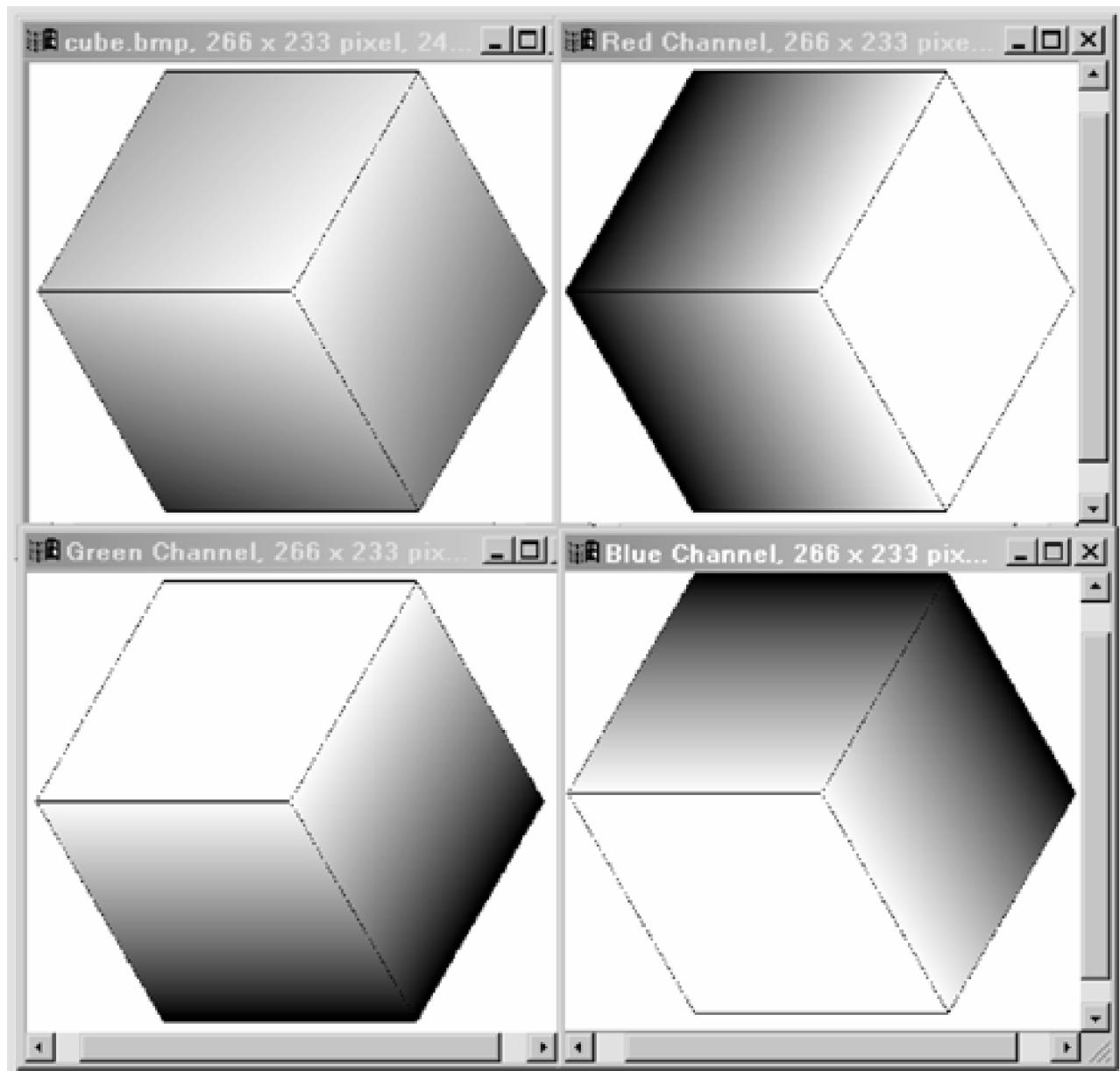
        case IDM_COLOR_SPLITHLS:
            CreateNewView(m_DIB.SplitChannel(TakeH), "Hue Channel");
            CreateNewView(m_DIB.SplitChannel(TakeL), "Lightness Channel");
            CreateNewView(m_DIB.SplitChannel(TakeS), "Saturation Channel");
    }
}
```

```
return 0;

case IDM_COLOR_SPLITKCMY:
    CreateNewView(m_DIB.SplitChannel(TakeK), "Black Channel");
    CreateNewView(m_DIB.SplitChannel(TakeC), "Cyan Channel");
    CreateNewView(m_DIB.SplitChannel(TakeM), "Magenta Channel");
    CreateNewView(m_DIB.SplitChannel(TakeY), "Yellow Channel");
    return 0;
...
}
```

The SplitChannel routine returns a pointer to a packed DIB. The KDIBView ::CreatenewView routine then takes it to create a new DIB MDI child window to display it. If one of the channel-splitting commands is selected, a few new grayscale DIB windows will be added to the MDI frame window, each displaying a new bitmap. [Figure 12-3](#) shows splitting an RGB cube into RGB channels. Note that in grayscale images, light colors have higher intensity, while darker colors have lower intensity. This explains why one of the three diamond shapes in each grayscale image is pure white, because it corresponds to the surface in the original color cube where the corresponding color channel is having maximum intensity (255).

[Figure 12-3. RGB color channel splitting.](#)



Histogram

Just to make the point that the KPixelMapper is a generic pixel-transformation class, we're throwing in a different derived class—a histogram generator.

```
// RGB Histogram class derived from KPixelMapper
class KHistogram : public KPixelMapper
{
    int      m_FreqRed[256];
    int      m_FreqGreen[256];
    int      m_FreqBlue[256];
    int      m_FreqGray[256];

    // return true if data changed
    virtual bool MapRGB(BYTE & red, BYTE & green, BYTE & blue)
    {
```

```
m_FreqRed[red]   ++;
m_FreqGreen[green]++;
m_FreqBlue[blue] ++;
m_FreqGray[(red * 77 + green * 150 + blue * 29 + 128 ) / 256]++;
return false;
}
```

public:

```
void Sample(CDIB & dib);
};
```

```
void KHistogram::Sample(CDIB & dib)
{
    memset(m_FreqRed, 0, sizeof(m_FreqRed));
    memset(m_FreqGreen, 0, sizeof(m_FreqGreen));
    memset(m_FreqBlue, 0, sizeof(m_FreqBlue));
    memset(m_FreqGray, 0, sizeof(m_FreqGray));

    dib.PixelTransform(* this);
}
```

The KHistogram class calculates RGB and gray-level frequency in four integer arrays. The MapRGB implementation just increases the frequency counts for the RGB channels and the grayscale level. After calling KHistogram::Sample, the collected histograms can be displayed in a graphical form to guide the image editor user on how the image should be adjusted.

[< BACK](#) [NEXT >](#)

12.6 BITMAP SPATIAL FILTERS

A pixel transformation determines the output for each pixel by looking at only a single input pixel. There is another class of image-processing algorithms that work by looking at neighboring pixels of a pixel to determine the output pixel. Such algorithms are normally called spatial filters. For example, an image-smoothing filter may take an average of a 3×3 pixel block to generate the output pixel, in order to filter out high-frequency noise.

A spatial filter takes a source bitmap and produces a destination bitmap. It normally looks at an $N \times N$ pixel block, where N is an odd number. Most common spatial filters use $N = 3$. The center of the pixel block is then pixel-aligned with the pixel under consideration; the other pixels are its neighboring pixels. An odd N makes the block symmetric around the center pixel. When processing a whole image, such a spatial filter can't process $(N - 1)/2$ -pixel-wide edges around the bitmap, because some of the pixels in the $N \times N$ pixel block will be out of the source image. To solve this problem, we can either copy the source pixel directly to the destination pixel for the edges, or fill the edges with a uniform color.

Our previous classes and functions can't handle spatial filters; we need something new that can handle an $N \times N$ pixel block as input data for each pixel. [Listing 12-7](#) shows an abstract KFilter class.

The KFilter class looks much simpler than the KPixelTransform class, mainly because the former now handles only grayscale 8-bpp, 24-bpp, and 32-bpp bitmaps. Spatial filters involve arithmetic operations on pixel values, which can't be well expressed in a palette-based image. Supporting 15-bpp and 16-bpp images is possible, but we choose to skip the extra coding here.

The KFilter works on one single-channel grayscale image at a time. 24-bpp and 32-bpp images are taken as multiple grayscale channels and operated on independently. The KFilter::Kernel pure virtual function determines how a spatial filter works. To KFilter::Kernel, a pixel is a single byte in the range of 0...255, representing the per-channel intensity value. It accepts a pointer to the current pixel, an address offset to the next pixel in the same row, and an address offset to the next pixel in the same column. From these three pixels, a Kernel implementation has access to all its neighboring pixels through addition and subtraction. The function returns a byte value that will be written to an output image by its caller. Now you can see that 15-bpp and 16-bpp scan lines won't easily fit into this model. The m_nHalf member variable holds the value $(N - 1)/2$, so it's normally 1 for 3×3 filters.

Listing 12-7 Class KFilter for Spatial Filters

```
// Abstract class for image spatial filter, handling pixel and scan line
class KFilter
{
    int    m_nHalf;

    virtual BYTE Kernel(BYTE * pPixel, int dx, int dy) = 0;

public:
    cdt
```

```
int GetHalf(void) const { return m_nHalf; }

KFilter(void) { m_nHalf = 1; }

virtual ~KFilter() {}

virtual void Filter8bpp (BYTE * pDst, BYTE * pSrc, int nWidth, int dy);
virtual void Filter24bpp(BYTE * pDst, BYTE * pSrc, int nWidth, int dy);
virtual void Filter32bpp(BYTE * pDst, BYTE * pSrc, int nWidth, int dy);

virtual void DescribeFilter(HDC hDC, int x, int y);
};

void KFilter::Filter8bpp(BYTE * pDst, BYTE * pSrc, int nWidth, int dy)
{
    memcpy(pDst, pSrc, m_nHalf);
    pDst += m_nHalf; pSrc += m_nHalf;

    for (int i=nWidth - 2 * m_nHalf; i>0; i--)
        * pDst ++ = Kernel(pSrc++, 1, dy);

    memcpy(pDst, pSrc, m_nHalf);
}

void KFilter::Filter24bpp(BYTE * pDst, BYTE * pSrc, int nWidth, int dy)
{
    memcpy(pDst, pSrc, m_nHalf * 3);
    pDst += m_nHalf * 3; pSrc += m_nHalf * 3;

    for (int i=nWidth - 2 * m_nHalf; i>0; i--)
    {
        * pDst ++ = Kernel(pSrc++, 3, dy);
        * pDst ++ = Kernel(pSrc++, 3, dy);
        * pDst ++ = Kernel(pSrc++, 3, dy);
    }
    memcpy(pDst, pSrc, m_nHalf * 3);
}

void KFilter::Filter32bpp(BYTE * pDst, BYTE * pSrc, int nWidth, int dy)
{
    memcpy(pDst, pSrc, m_nHalf * 4);
    pDst += m_nHalf * 4; pSrc += m_nHalf * 4;

    for (int i=nWidth - 2 * m_nHalf; i>0; i--)
    {
        * pDst ++ = Kernel(pSrc++, 4, dy);
        * pDst ++ = Kernel(pSrc++, 4, dy);
        * pDst ++ = Kernel(pSrc++, 4, dy);
        * pDst ++ = * pSrc++;           // copy alpha channel
    }
}
```

```
    memcpy(pDst, pSrc, m_nHalf * 4);
}
```

The Filter8bpp, Filter24bpp, and Filter32bpp methods handle the three scan-line types we support here: 8-bpp, 24-bpp, and 32-bpp. They accept a pointer to the destination and source scan lines, scan-line pixel width, and offset to next scan line. For each scan line, the first and last m_nHalf pixels are simply copied. The rest of the pixels are passed to the Kernel method one channel at a time, with the results written to the destination scan line. These methods are declared as virtual functions so that they can be customized in the derived classes.

A new method KImage::SpatialFilter is added to the KImage class to feed the whole DIB to the KFilter class. It allocates a new destination pixel array, copies over the first and last few unprocessable scan lines, and calls one of the KFilter filtering methods to process the rest. It finally replaces the old pixel array with the new pixel array. The implementation here can be changed to either generate a new bitmap, or use the processed source pixel array to store the result, so that only a few scan lines' worth of space need to be allocated.

```
bool KImage::SpatialFilter(KFilter & filter)
{
    BYTE * pDestBits = new BYTE[m_nImageSize];

    if ( pDestBits==NULL )
        return false;

    for (int y=0; y<m_nHeight; y++)
    {
        BYTE * pBuffer = (BYTE *) m_pBits + m_nBPS * y;
        BYTE * pDest = (BYTE *) pDestBits + m_nBPS * y;

        if ( (y>=filter.GetHalf()) && (y<(m_nHeight- filter.GetHalf())) )
            switch ( m_nImageFormat )
            {
                case DIB_8BPP:
                    filter.Filter8bpp(pDest, pBuffer, m_nWidth, m_nBPS);
                    break;

                case DIB_24RGB888: // 24-bpp RGB
                    filter.Filter24bpp(pDest, pBuffer, m_nWidth, m_nBPS);
                    break;

                case DIB_32RGBA8888: // 32-bpp RGBA
                case DIB_32RGB888: // 32-bpp RGB
                    filter.Filter32bpp(pDest, pBuffer, m_nWidth, m_nBPS);
                    break;

                default:
                    delete [] pDestBits;
                    return false;
            }
        else
            memcpy(pDest, pBuffer, m_nBPS);
```

```
}

memcpy(m_pBits, pDestBits, m_nImageSize);
delete [] pDestBits;

return true;
}
```

A 3×3 spatial filter can normally be described using a 3×3 matrix and a weight. The numbers in the 3×3 matrix are multiplied with the color value of corresponding pixels in the 3×3 block, and their product divided by the weight is the final result. The result could be out of the 0...255 range we use to store a color channel level, so truncation may be needed. Certain filters need to add a constant number to the result before truncation. Here is a template class that supports 3×3 spatial filters with an extra weight and additive value:

```
template <int k00, int k01, int k02,
          int k10, int k11, int k12,
          int k20, int k21, int k22,
          int weight, int add, bool checkbound, TCHAR * name>
class K33Filter : public CFilter
{
    virtual BYTE Kernel(BYTE * P, int dx, int dy)
    {
        int r = ( P[-dy-dx]*k00 + P[-dy]*k01 + P[-dy+dx]*k02 +
                  P[ -dx]*k10 + P[0] *k11 + P[ +dx]*k12 +
                  P[ dy-dx]*k20 + P[dy] *k21 + P[ dy+dx]*k22 )
                / weight + add;

        if ( checkbound )
            if ( r < 0 )
                return 0;
            else if ( r > 255 )
                return 255;

        return r;
    }
};
```

The K33Filter class has 12 parameters. The first nine define a 3×3 coefficient matrix, followed by a weight, a value to add, and a Boolean value to control whether bounds checking is needed. One way of doing this would be to use a bunch of floating-point numbers, which can share the same piece of code by just switching the data. But this implementation will require nine floating-point multiplications and converting floating point to integer. The K33Filter template can significantly improve the performance of applying a spatial filter by using integer parameters only. We are using integers only, and each filter will be instantiated with its own set of template parameters. For the compiler, the nine multiplications and one division are all operations with constants, which can easily be optimized or even totally skipped. If the bounds-checking variable is false, the bounds-checking code does not need to be generated, either. The cost in code-size increase is minimum, because each filter needs to overwrite only a single function.

With these classes, we're ready to try to define several spatial filters to see what wonders they can do to bitmaps.

Smoothing and Sharpening Filters

[Figure 12-4](#) shows four small pictures: one original picture, and three pictures demonstrating the smooth filter, the Gaussian smooth filter, and the sharpening spatial filter.

Figure 12-4. Smoothing and sharpening filters.



To make the filtering result more visible, the pictures in [Figure 12-4](#) are displayed using 3:1 scale.

The three filters demonstrated in [Figure 12-4](#) can be defined by:

```
TCHAR szSmooth[]      = _T("Smooth");
TCHAR szGaussianSmooth[] = _T("Gaussian Smooth");
TCHAR szSharpening[]   = _T("Sharpening");

K33Filter < 1, 1, 1, 1, 1, 1, 1, 1, 1, 9, 0, false,
           szSmooth > filter33_smooth;
K33Filter< 0, 1, 0, 1, 4, 1, 0, 1, 0, 8, 0, false,
           szGaussianSmooth > filter33_gaussiansmooth;
K33Filter< 0, -1, 0, -1, 9, -1, 0, -1, 0, 5, 0, true,
           szSharpening > filter33_sharpening;
```

The original picture is shown as the first picture. The second picture shows the effect of the smooth filter. It has all ones in the 3×3 matrix and a weight of 9. So the filter assigns to a pixel the average pixel value of the 3×3 block it is in. A smooth filter is called a low-pass filter, which preserves low-frequency features and filters out high-frequency

noises. A smooth filter can be used to simulate a kind of antialiasing effect to smooth the lines, shapes, and bitmaps drawn by GDI. [Figure 12-4](#) shows how a jagged edge in the original picture is smoothed when applying a smooth filter. After a smooth filter is applied, grayscale pixels are introduced along the edges of the glyph to make them smoother.

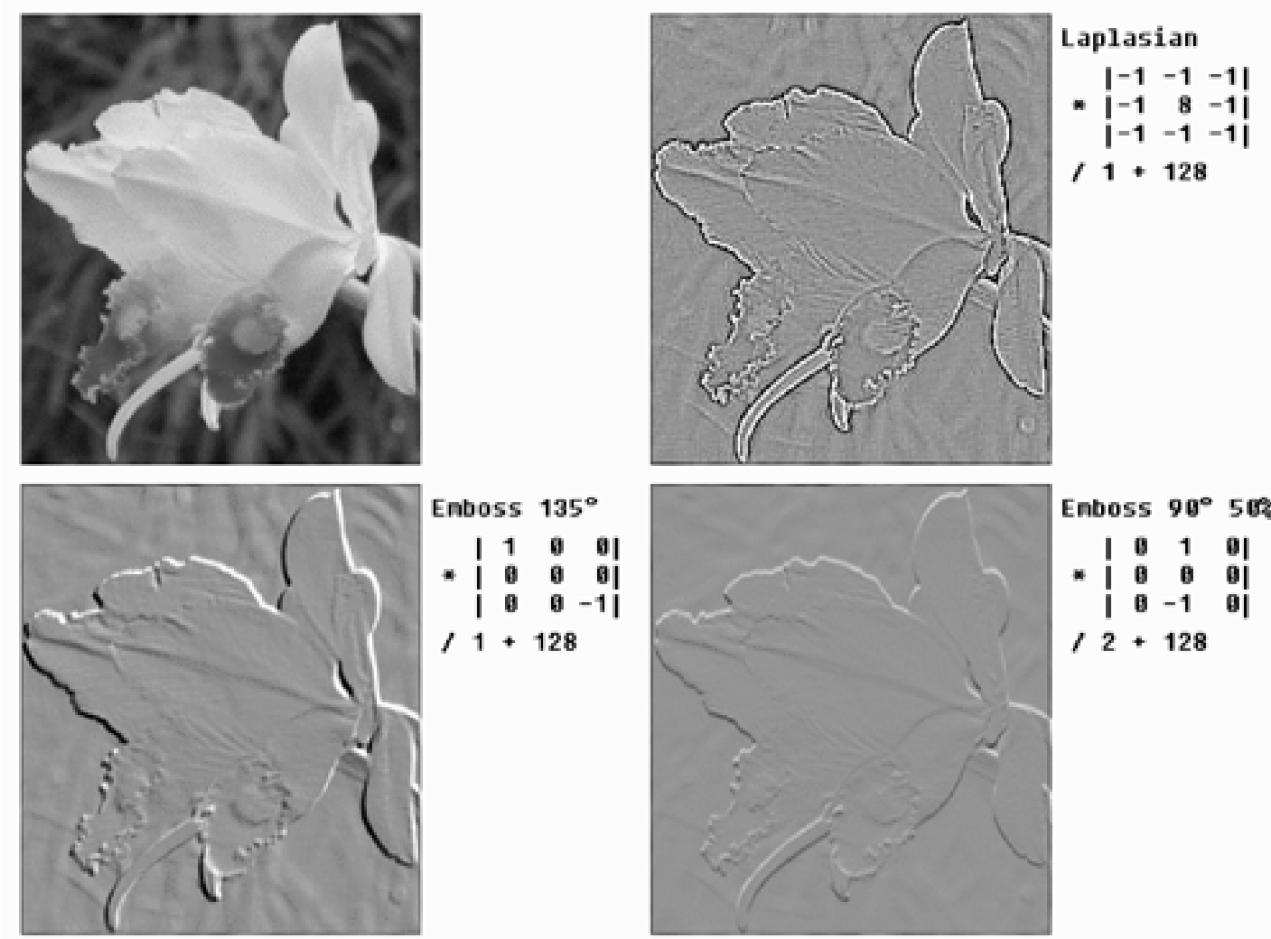
The Gaussian smooth filter is another kind of low-pass filter. Instead of using a uniform distribution, the Gaussian smooth filter uses a Gaussian distribution that puts more weight on the center pixel. Gaussian filters can be defined for a larger radius. What's shown here is a 3×3 filter.

The sharpening filter subtracts the neighboring pixels from the current pixel to emphasize variations in the image. It belongs to high-pass filters that accentuate high-frequency components of an image while leaving the low-frequency parts unchanged. Adjusting the weight on the center pixel can control the level of sharpening. On the monochrome picture shown in [Figure 12-4](#), the sharpening filter does not make much of a difference.

Edge-Detection and Embossing Filters

[Figure 12-5](#) illustrates a Laplacian edge-detection filter and two embossing filters. They can be defined by:

Figure 12-5. Edge-detection and embossing filters.



```
TCHAR szLaplacian[] = _T("Laplacian");
TCHAR szEmboss135[] = _T("Emboss 135°");
```

```
TCHAR szEmboss90[] = _T("Emboss 90° 50%");

K33Filter<-1, -1, -1, -1, 8, -1, -1, -1, 1, 128, true,
szLaplacian > filter33_laplacian;
K33Filter< 1, 0, 0, 0, 0, 0, 0, 0, -1, 1, 128, true,
szEmboss135 > filter33_emboss135;
K33Filter< 0, 1, 0, 0, 0, 0, 0, -1, 0, 2, 128, true,
szEmboss90 > filter33_emboss90;
```

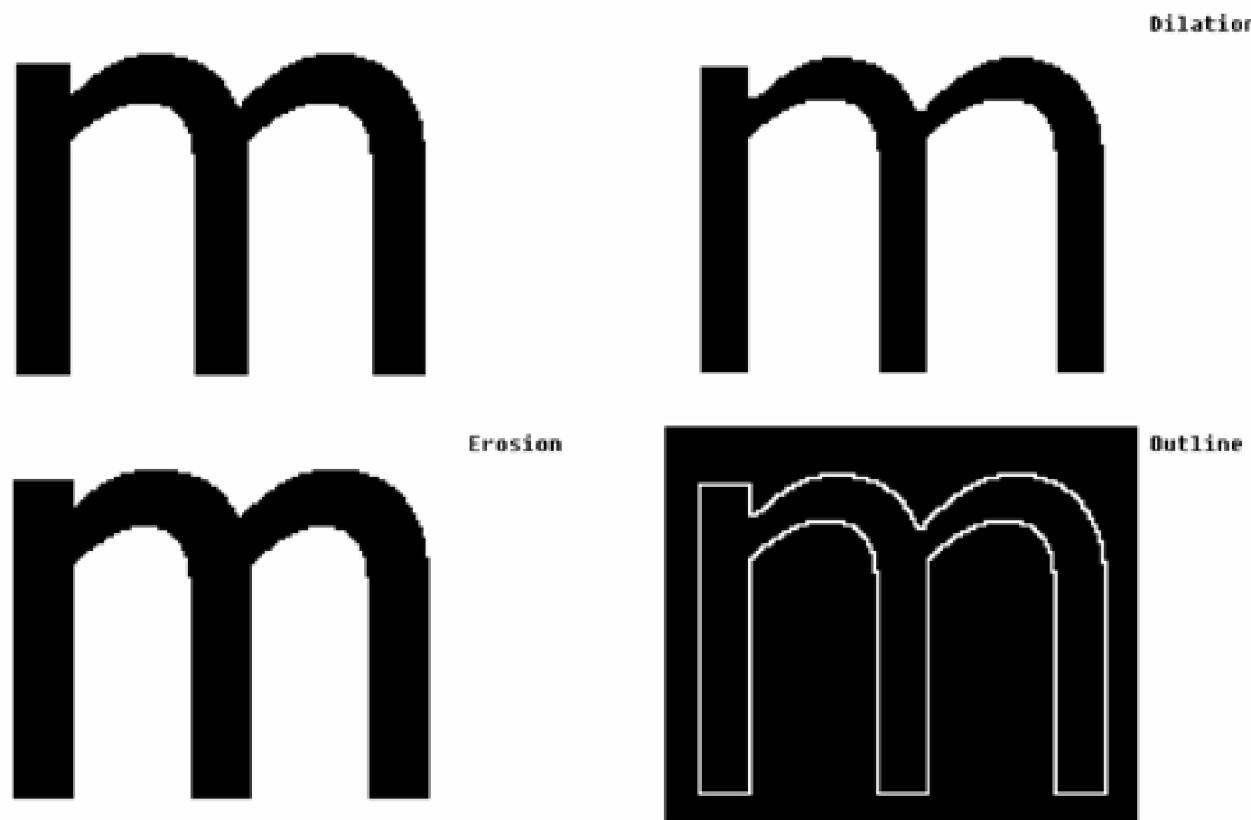
The Laplacian filter looks similar to a high-pass filter, but it generates a totally different image. It belongs to the edge-detection filters, which have zero-sum coefficients in the matrix. An edge-detection filter turns areas with uniform colors to black, and areas with changes to nonblack colors. The filter we're using here adds 128 to each channel to avoid having a negative result changed to 0. Adding 128 turns areas with uniform colors to gray.

The next two filters, called emboss filters, can turn a color image to a mostly grayscale image with some interesting 3D effects. For an emboss filter, its matrix has 1 on one corner and -1 on the opposite corner. Applying an emboss filter can be thought of as subtracting an image displaced by a certain amount from the original image. We're adding 128 to the result to move the zero point to a medium grayscale level. The relative positions of the pixel with 1 and the pixel with -1 determine the direction of embossing. Two directions are shown here. The second emboss filter divides the matrix multiplication result by 2 to reduce the amount of embossing.

Morphological Filters

[Figure 12-6](#) illustrates three new spatial filters: dilation filter, erosion filter, and outline filter. The pictures are displayed at 2:1 scale to make the effects more visible.

Figure 12-6. Morphological filters.



These three filters differ from the previous filters, which are based on a linear combination of pixels. They are what's called morphological filters. A morphological filter uses an $N \times N$ matrix to match the neighboring pixels of a pixel. The matching determines the resulting value for the pixel in the center.

The dilation operator generates black only when all pixels in a structure pattern are black, and white otherwise. So after a dilation filter is applied, the white area in an image expands.

The erosion operator generates white only when all pixels in a structure pattern are white, and black otherwise. So after an erosion filter is applied, the white area in an image shrinks.

The outline operator does a dilation first and then subtracts the original image from it. For smooth areas, the outline operator generates black (0) because the dilated image is the same as the original image. For new white pixels generated due to dilation, the outline operator gives white. The result is an outline of the original image.

[Figure 12-6](#) shows the result of applying these three operators on a monochrome image of a text character. Note that the foreground color of the character is black and the background color is white, so dilation grows background white color and shrinks foreground black color, while erosion shrinks background white color and grows foreground black color. So for this test case, dilation makes the letter "m" in the picture smaller, while erosion makes it bigger. The outline operator generates an outline in white color.

These morphological filters are originally defined for monochrome images. For color image channels in multiple grayscale images, we can use the minimum function to simulate erosion, and the maximum function to simulate dilation. Here is our implementation of the erosion filter. The KErosion::Kernel function finds the minimum value of nine pixels in the 3×3 block and returns it as the function result. For color images, it could also find the minimum value of eight pixels around the center pixel, and use the average of the central pixel and the minimum value as the result. This gives a slowed-down erosion effect. To get the dilation filter, just switch minimum to maximum.

```
// Minimum to grow the darker region
class KErosion : public KFilter
{
    inline void smaller(BYTE &x, BYTE y)
    {
        if ( y < x ) x = y;
    }

    BYTE Kernel(BYTE * pPixel, int dx, int dy)
    {
        BYTE m = pPixel[-dy-dx];
        smaller(m, pPixel[-dy]);
        smaller(m, pPixel[-dy+dx]);
        smaller(m, pPixel[ -dx]);
        smaller(m, pPixel[ +dx]);
        smaller(m, pPixel[ dy-dx]);
        smaller(m, pPixel[dy]);
        smaller(m, pPixel[ dy+dx]);

        return min( pPixel[0] , m ); // /2;
    }
};
```

Image processing is an exciting subject, but because this is a graphics programming book, our focus here is to show how direct access to the DIB and DIB section pixel array can allow you to achieve all these cool effects. By doing that, we've developed several generic classes and templates to allow applications to add their own pieces for their own purposes.

[< BACK](#) [NEXT >](#)

12.7 SUMMARY

The main focus of this chapter has been the direct accessing of the pixel array of a DIB or a DIB section. With this ability to access individual pixels in a bitmap, lots of interesting algorithms can be implemented and endless effects can be achieved.

This chapter has demonstrated how to use direct pixel access to implement a generic bitmap affine transformation algorithm that does not rely on GDI's bitmap rotation functions, which are available only on NT-based systems. With a specialized, highly optimized, integer-only image affine transformation algorithm, it has been demonstrated that millions of pixels can be processed within a second.

With direct pixel accessing, cool image-processing algorithms can be implemented to provide features not directly provided by GDI. This chapter built a generic framework to implement image color-transformation algorithms, pixel-transformation algorithms, and spatial filters. Lots of other image-processing algorithms can be implemented based on the template classes and abstract classes developed in this chapter.

Techniques developed in this chapter can also be applied to achieve effects like antialiasing or drop shadow, or used in a DirectDraw surface, which is just like a DIB section with hardware acceleration.

[Chapter 13](#) will cover palette, image color reduction, and halftoning. [Chapter 17](#) will discuss JPEG image decoding and printing. [Chapter 18](#) will use similar direct pixel access methods on DirectDraw surfaces.

Further Reading

With direct pixel accessing, the sky is the limit. So there are endless books to read. Just grab any good computer graphics or image-processing book and try to implement the algorithms in it.

Compared with GDI, the Java 2D graphics API provides much more powerful image-processing capabilities. Its `BufferedImage` class, `BufferedImageOp` interface, etc., are very similar to what this chapter is trying to do. *Java 2D Graphics*, by Jonathan Knudsen, is a good read on Java 2D graphics.

As always, the *Graphics Gems* books, Volumes I–V, are a good source of graphics and image-processing algorithms.

Sample Program

[Chapter 12](#) has only one program, which demonstrates all the topics covered here (see [Table 12-1](#)).

Table 12-1. Sample Program for Chapter 12

Directory	Description
Samples\Chapt_12\Imaging	Demonstrates direct pixel access, converting to grayscale, gamma correction, image affine transformation, image color transformation, image pixel transformation, and various spatial filters.
	Opens a BMP file and tries menu options under “color” and “View.”

[< BACK](#) [NEXT >](#)

Chapter 13. Palettes

Up to this chapter, we have been using lots of colors in our programs; we talked about color pens, color brushes, 16-, 24-, and 32-bit bitmaps, gradient fill, alpha blending, and image processing. But if we run these programs on a 256-color display monitor, suddenly we go back to the dark ages. All the colorful displays turn grayish with ugly dithering patterns.

The problem lies in the palette, a tool that Windows GDI and display hardware borrowed from graphics artists to map the color index in a palette-based frame buffer to RGB colors.

In this chapter, we discuss what happens if you don't pay attention to the palette at all, what's the least you should do if you want your program to run reasonably well on palette-based displays, and what you can do to take full advantage of the palette. This chapter also discusses color quantization, an algorithm to convert high-color or true-color images to indexed color images with an optimal color table.

13.1 SYSTEM PALETTE

To see what happens when you switch from Windows to a 256-color display mode, first display some colorful screen using basic operating-system programs. For example, the default window screen has several colorful icons on it, the start menu is quite bright, and the color selection display box is supposed to display lots of colors. Now try to figure out how many colors you are actually seeing after you switch to a 256-color display mode. To get the correct answer, you can capture the screen and use an image processing program to count the exact number of colors in the captured bitmap. The answer is not more than 20 colors.

The whole Windows operating system user interface is built using 20 colors when running in a 256-color display mode. If an application program does not handle any palette, it normally can use only the same 20 colors. Icons and toolbars will be displayed in 20 colors. LoadBitmap converts any color bitmap to a 256-color DDB, but with only 20 colors actually being used. DIBs and DIB sections are also displayed in 20 colors. Any other colors are dithered using a combination of these 20 colors. The most frustrating fact is that even 256-color bitmaps loaded with LoadBitmap are displayed using 20 colors.

To understand the problem and the solution, we have to understand the system palette, the logical palette, and the realization of the logical palette.

Display Settings

Due to the price drop in RAM, most display monitors should not be in a 256-color display mode anymore. But it's still possible that some old software will ask you to switch to a 256-color mode. To test the palette-related code in this chapter, your display mode needs to be switched to a 256-color mode, which can be done using the control panel's display applet.

To check if a device really supports the hardware palette, a program should query the device's RASTERCAPS flag and check for the RC_PALETTE bit. If this bit is on, your graphic device is set up in a palette-based mode. The detailed information about a display card's current settings and available settings can be queried using the Enum DisplaySettings function. If an application wants to change the display setting, call the Change DisplaySettings function. Here is a Switch8bpp routine used in this chapter's sample program, "Palette," which prompts the user to switch to a 256-color display mode if the current display setting does not support the hardware palette.

```
BOOL Switch8bpp(void)
{
    HDC hDC = GetDC(NULL);
    int hasPalette = (GetDeviceCaps(hDC, RASTERCAPS) & RC_PALETTE);
    ReleaseDC(NULL, hDC);

    if ( hasPalette ) // palette supported
        return TRUE;

    int rslt = MessageBox(NULL, _T("Switch to 256 color mode?"),
        _T("Palette"), MB_YESNOCANCEL);
```

```
if ( rslt==IDCANCEL )
    return FALSE;

if ( rslt==IDYES ) // switch to 8-bpp allowed
{
    DEVMODE dm;
    dm.dmSize      = sizeof(dm); // important, avoid GPF
    dm.dmDriverExtra = 0;
    EnumDisplaySettings(NULL,ENUM_CURRENT_SETTINGS,&dm); // current
    dm.dmBitsPerPel = 8;           // 8-bpp
    ChangeDisplaySettings(&dm, 0); // SWITCH
}
return TRUE;
}
```

The Switch8bpp routine uses GetDeviceCaps to check if the current primary display supports the palette; if not, the user is prompted with a change-display-setting request. If the user agrees to change the setting, EnumDisplaySettings is used to query the current DEVMODE structure, which holds the information device setting. After changing DEVMODE's dmBitsPerPel field to 8, it calls ChangeDisplaySettings with the changed DEVMODE to switch the screen to a 8-bpp, 256-color display mode. A WM_DISPLAYCHANGE message will be sent to all top-level windows.

Querying the System Palette

When running in a 256-color display mode, each pixel is stored using a single byte in the display card's frame buffer. One byte gives a maximum of 256 different colors that can be displayed at the same time. The exact combination of displayable colors is managed by the hardware palette, which is presented to the user application as the system palette.

A system palette in a 256-color display mode is a table of 256 entries, each being a PALETTEENTRY structure. GDI provides a few functions to query and control the system palette.

```
typedef struct {
    BYTE peRed;
    BYTE peGreen;
    BYTE peBlue;
    BYTE peFlags;
} PALETTEENTRY;

UINT GetSystemPaletteEntries(HDC hDC, UINT istartIndex, UINT nEntries,
    LPPALETTEENTRY lppe);
UINT GetSystemPaletteUse(HDC hDC);
UINT SetSystemPaletteUse(HDC hDC, UINT uUsage);
```

The PALETTEENTRY structure specifies a color using its RGB components. The peFlags is used when creating a logical palette, which will be discussed in the next section. GetSystemPaletteEntries returns a block of entries from

the current system palette for a graphics device. The first parameter is a handle to a device context, the next two parameters specify where to start and how many to copy, and the last parameter is a pointer to receiving an array. If you're not sure how many entries are in the system palette, calling GetSystemPaletteEntries(hDC, 0, 0, NULL) will return the total number of entries.

The system palette is a per-graphics-device resource, shared by all the device contexts created on it. For a graphics device card, all the windows use the same system palette. Applications can modify the system palette in a controlled manner, so the system palette is a volatile piece of data. After the system palette is changed, the operating system broadcasts a WM_PALETTECHANGED message to every top-level window in the system, to give them a chance to respond to the changes. When needed, top-level windows should themselves forward the message to their child windows that may need it.

To help understand the dynamic nature of the system palette, we can write a small popup window to display the system palette and monitor every change to it. [Listing 13-1](#) shows the KPaletteWnd class. Its CreatePaletteWindow method creates a popup window to display all the colors in the system palette. An initialized 256-color device-dependent bitmap (DDB) is used to display the system palette. We're making the assumption that the display card is using a single plane, 8 bits per pixel, to represent the DDB, which seems to be universal. The data to initialize the DDB is set to be 16-by-16 blocks of uniform color ranging from 0 to 255. Because the data is supposed to be in the internal DDB format, no color conversion is done when creating and displaying the DDB. So a byte with 0 in the DDB will show the color of the first entry in the system palette. The message handler handles the WM_PALETTE CHANGED message by refreshing the display.

Listing 13-1 Class for Visualizing System Palette Changes

```
class KPaletteWnd : public KWindow
{
    virtual LRESULT WndProc(HWND hWnd, UINT uMsg,
                           WPARAM wParam, LPARAM lParam)
    {
        switch ( uMsg )
        {
            case WM_PAINT:
            {
                PAINTSTRUCT ps;
                HDC hDC = BeginPaint(hWnd, & ps);
                HDC hMemDC = CreateCompatibleDC(hDC);
                BYTE data[80][80]; // initialization data for 8-bpp DDB

                for (int i=0; i<80; i++)
                    for (int j=0; j<80; j++)
                {
                    data[i][j] = (i/5) * 16 + (j/5);
                    if ( ((i%5)==0) || ((j%5)==0) )
                        data[i][j] = 255;
                }

                HBITMAP hBitmap = CreateBitmap(80, 80, 1, 8, data);
                HGDIOBJ hOld = SelectObject(hMemDC, hBitmap);
```

```
StretchBlt(hDC,10,10,256,256,hMemDC,0,0,80,80, SRCCOPY);
SelectObject(hMemDC, hOld);
DeleteObject(hBitmap);
DeleteObject(hMemDC);
EndPaint(hWnd, & ps);
}

return 0;

case WM_PALETTECHANGED:
    InvalidateRect(hWnd, NULL, TRUE);
    return 0;
}

return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

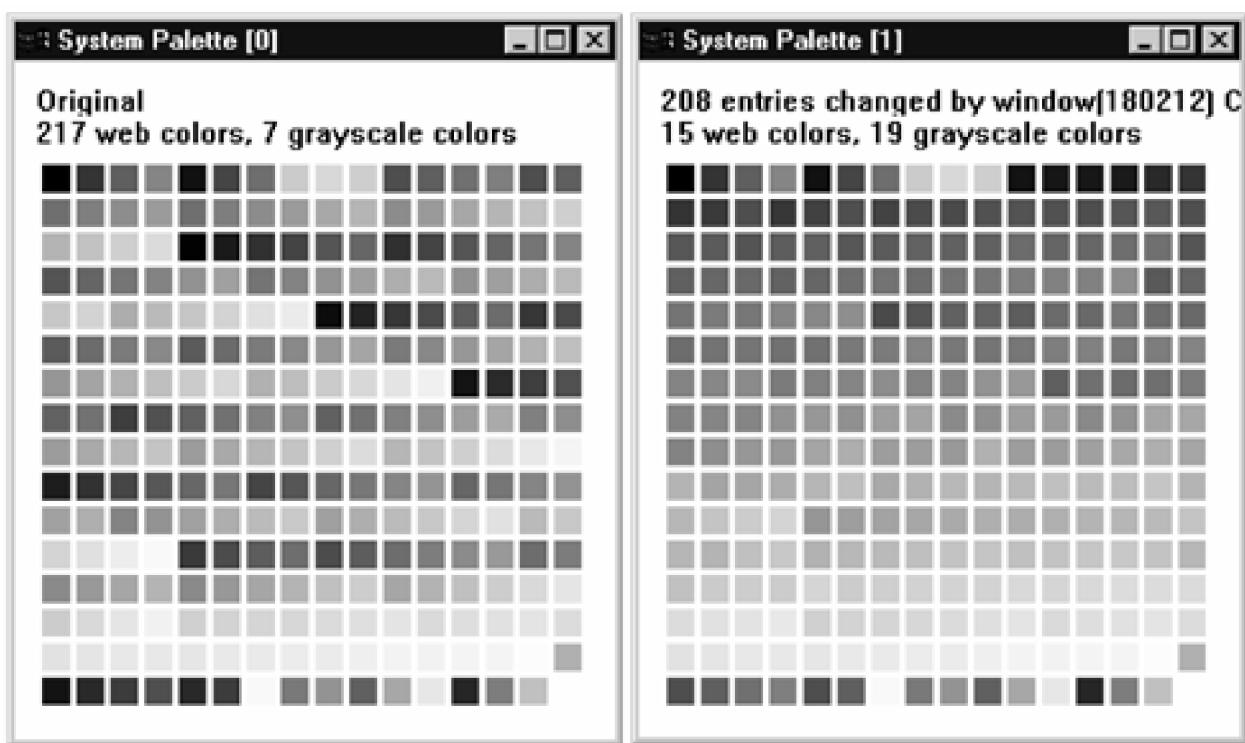
public:

void CreatePaletteWindow(HINSTANCE hInst)
{
if ( ! Switch8bpp() ) // if cancelled
    return;

CreateEx(0, _T("SysPalette"), _T("System Palette"),
    WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN, CW_USEDEFAULT,
    CW_USEDEFAULT, 290, 340, NULL, NULL, hInst);
ShowWindow(nShow);
UpdateWindow();
}
};
```

The actual code on the CD is more complicated than what's shown in [Listing 13-1](#). When processing the WM_PALETTECHANGED message, it uses the GetSystemPaletteEntries call to retrieve all the entries in the system palette, so that we can do a simple analysis on the colors in the palette and compare with the previous palette when the palette-changing message is received. [Figure 13-1](#) shows two sample screens of the system palette-monitoring window.

Figure 13-1. Monitoring system palette changes.



When running a program with the system palette-monitoring window open, you can find that the system palette starts with a quite uniform color distribution in the RGB color space. The RGB color space has three channels. If you want a uniform color distribution, each channel will have roughly $256^{1/3} = 6.34$ levels, with 6 being the closest integer value. Each RGB channel has values ranging from 0 to 255; the range can be approximated using 6 levels using 0, 51, 102, 153, 204, and 255. If each of the RGB channels has 6 levels as such, we only need 216 colors. These are called web-safe colors, because they are supported both by Microsoft's and Netscape's browsers. Images with these colors can be displayed without dithering on both browsers. Besides these web-safe colors, the initial palette also contains extra grayscale colors—that is, colors with the same red, green, and blue values.

[Figure 13-1](#) shows two screens generated by the system palette-monitoring window. The first screen shows the default system palette with 217 web-safe colors (one color is duplicated) and 7 other grayscale colors. After an application with a fancy splash window is displayed, the palette changes dramatically: 208 colors within the system palette have been changed to create a palette to display the splash window in its best possible way. When you open and close other applications, especially graphics applications, you will find the system palette changes frequently, even when switching between two MDI child windows within the same MDI frame window.

Static Colors

The top part and the bottom part of the system palette never seem to change. These two parts hold the system static colors—colors reserved by the operating system to display its user interface. The Windows operating system normally reserves 20 static colors, although it's possible to reduce the number.

`GetSystemPaletteUse` returns a flag indicating the number of static colors used by the system. If the return value is `SYSPAL_NOSTATIC`, the system uses two static colors, black and white; if the return value is `SYSPAL_STATIC`, 20 static colors are used. Windows 2000 supports a new flag, `SYSPAL_NOSTATIC256`, for absolutely no reserved static color.

`SetSystemPaletteUse` changes the current static color setting using the same flag mentioned above. Static colors are used as system colors, as in the `GetSysColor`, `SetSysColor` API. So if an application wants to reduce the static

colors to SYSPAL_NOTSTATIC, it must save all the current system colors, change the number of static colors, reset the system colors to a reasonable value, and restore the original system colors when finishing. Changing the number of static colors should be done only in extreme cases. For example, if a medical image application wants to display 256 shades of an x-ray image in a 256-color display mode, it has to use the SYSPAL_NOSTATIC or SYSPAL_NOSTATIC256 options.

The arrangement of the 20 static colors is quite interesting. Sixteen of them are from the 16-color VGA palette; the other four are defined by the current window color scheme. [Table 13-1](#) lists the static colors under two different color schemes: the traditional window color scheme and the spruce color scheme.

Of the 20 static colors, 8 dark colors are put in the first 8 slots in the system palette, and 8 light colors are put in the last 8 slots. These 16 colors are used for “pure” colors: red, green, blue, cyan, magenta, yellow, their dark versions, and four grayscale colors. These 16 colors are always there in the same spot, if the 20 static colors are used. They are put into opposite corners of the system palette to make sure the basic raster operations on them make sense. For example, it's critical to put black at slot 0 and white at slot 255. The masking raster operation depends on this to function. Entry 1 is dark red (RGB(0x80, 0x00, 0x00)); if you invert the index, it goes to entry 0xFE cyan (RGB(0x00, 0xFF, 0xFF)), which is not exactly red's complement color in RGB color, RGB(0x7F, 0xFF, 0xFF), but quite close. If you combine red and green, you get the perfect yellow, because $0xF9 | 0xFA = 0xFB$.

Table 13-1. Static Colors

Index	RGB Value	Color Name	Default System Color Usage
0x00	0x00, 0x00, 0x00	Black	COLOR_WINDOWFRAME, COLOR_MENUTEXT, COLOR_WINDOWTEXT, COLOR_3DDKSHADOW, COLOR_INFOTEXT
0x01	0x80, 0x00, 0x00	Dark red	
0x02	0x00, 0x80, 0x00	Dark green	
0x03	0x80, 0x80, 0x00	Dark yellow	
0x04	0x00, 0x00, 0x80	Dark blue	
0x05	0x80, 0x00, 0x80	Dark magenta	
0x06	0x00, 0x80, 0x80	Dark cyan	
0x07	0xC0, 0xC0, 0xC0	Light gray	
0x08	0xC0, 0xDC, 0xC0 0x59, 0x97, 0x64	Money green	COLOR_ACTIVECAPTION, COLOR_HIGHLIGHT, COLOR_BTNSHADOW, COLOR_GRAYTEXT
0x09	0xA6, 0xCA, 0xF0 0xA2, 0xC8, 0xA9	Sky blue	COLOR_MENU, COLOR_ACTIVEBORDER, COLOR_INACTIVEBORDER, COLOR_BTNFACE, COLOR_3DLIGHT
0xF6	0xFF, 0xFB, 0xF0 0xD0, 0xE3, 0xD3	Cream	COLOR_SCROLLBAR, COLOR_APPWORKSPACE, COLOR_INACTIVECAPTIONTEXT, COLOR_BTNHIGHLIGHT
0xF7	0x3A, 0x6E, 0xA5 0x21, 0x3F, 0x21		COLOR_BACKGROUND
0xF8	0x80, 0x80, 0x80	Dark gray	
0xF9	0xFF, 0x00, 0x00	Red	
0xFA	0x00, 0xFF, 0x00	Green	
0xFB	0xFF, 0xFF, 0x00	Yellow	
0xFC	0x00, 0x00, 0xFF	Blue	
0xFD	0xFF, 0x00, 0xFF	Magenta	
0xFE	0x00, 0xFF, 0xFF	Cyan	
0xFF	0xFF, 0xFF, 0xFF	White	COLOR_WINDOW, COLOR_CAPTIONTEXT, COLOR_HIGHLIGHTTEXT, COLOR_INFOBK

The four colors in the middle of the static colors can change according to the current color scheme. [Table 13-1](#) shows their actual RGB color in two color schemes. These colors are widely used in the Windows user interface. The last column of the table shows their default usage in various system colors. Under a 256-color display mode, the Windows operating system always tries to use static colors as system colors.

13.2 THE LOGICAL PALETTE

Although the system palette can handle 256 colors, your application can use only the 20 static colors if you don't do something special. The system palette is a systemwide resource, not a per-device-context resource. For a device context, the feature corresponding to the system palette is a logical palette. Logical palettes control how colors used in the GDI drawing commands are translated to color indexes in the drawing surface frame buffer.

Each device context has a logical palette attribute. Logical palettes form a class of GDI objects, similar to logical brushes, logical fonts, logical pens, etc. Here are the data structures and functions that deal with logical palettes.

```

UINT GetPaletteEntries(HPALETTE hpal, UINT iStartIndex, UINT nEntries,
    LPPALETTEENTRY lppe);
HPALETTE CreateHalftonePalette(HDC hDC);

HPALETTE SelectPalette(HDC hDC, HPALETTE hpal, BOOL bForceBackground);
UINT RealizePalette(HDC hDC);
BOOL ResizePalette(HPALETTE hpal, UINT nEntries);
BOOL UnrealizeObject(HGDIOBJ hgdiObj);
BOOL ResizePalette(HPALETTE hpal, UINT nEntries);

typedef struct tagLOGPALETTE {
    WORD    palVersion;
    WORD    palNumEntries;
    PALETTEENTRY palPalEntry[1];
} LOGPALETTE;

HPALETTE CreatePalette(CONST LOGPALETTE * lplgpl);

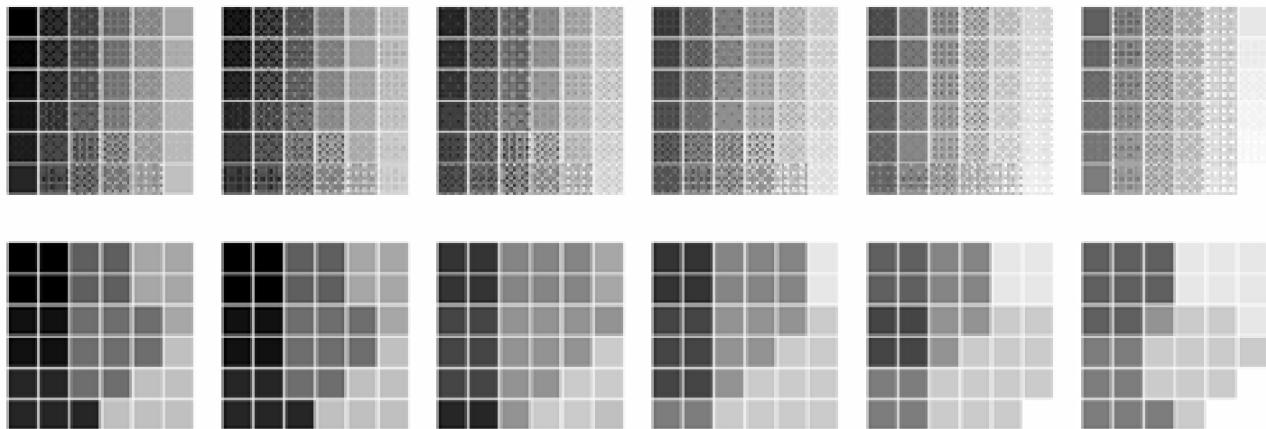
```

The Default Palette

The logical palette associated with a device context can be accessed using `GetCurrent Object(hDC, OBJ_PAL)`. For a new device context, its default logical palette is the stock default palette returned by `GetStockObject(DEFAULT_PALETTE)`. The default palette contains exactly the 20 system static colors, as shown in [Table 13-1](#). It limits the number of colors an application can use. For example, if you use `PALETTE INDEX` or `PALETTERGB` macros to specify colors in a device context with the default palette, the only available solid colors are the 20 static colors.

The following routine shows how to display the 216 web-safe colors. [Figure 13-2](#) shows the display result with the default palette. For the picture on the top, which uses the `RGB` macro to specify colors, most of the colors are displayed as dithered color, except for black, red, green, blue, yellow, cyan, magenta, and white, which are present in the system palette. For the picture on the bottom, which uses the `PALETTERGB` macro, the colors are matched with the 20 colors in the palette, and the matched colors are used as solid colors. Neither of them displays the expected colors.

Figure 13-2. Displaying web-safe colors using the default palette.



```
void WebColors(HDC hDC, int x, int y, int crttyp)
{
    for (int r=0; r<6; r++)
    for (int g=0; g<6; g++)
    for (int b=0; b<6; b++)
    {
        COLORREF cr;

        switch ( crttyp )
        {
            case 0: cr = RGB(r*51, g*51, b*51); break;
            case 1: cr = PALETTERGB(r*51, g*51, b*51); break;
        }

        HBRUSH hBrush = CreateSolidBrush(cr);

        RECT rect = { r * 110 + g*16+ x, b*16+ y,
                      r * 110 + g*16+15+x, b*16+15+y};
        FillRect(hDC, &rect, hBrush);

        DeleteObject(hBrush);
    }
}
```

The Halftone Palette

An easy way to increase the number of colors you can use is using the halftone palette. The `CreateHalftonePalette` function is provided to create a logical halftone palette. It's strange that the halftone palette is not provided as a stock object, which can then be shared by all processes running on the system. Once created, a halftone palette is just like other GDI objects; it needs to be deleted after using it.

SelectPalette needs to be called to select a logical palette into a device context. Note that the generic SelectObject function can't be used to do that, because selecting a palette needs an extra parameter, the force background flag. If the last parameter of Select Palette is TRUE, the palette will be treated later as a *background palette*; otherwise, it will be treated as a *foreground palette* if other conditions are met.

Before a palette can be used, it needs to be "realized." Realizing a logical palette is the process of allocating spaces in the system palette to accommodate an application's requirements and build a table of mapping from logical palette indexes to system palette indexes. If the palette being realized is a foreground palette, nonstatic colors in the system palette are removed; any colors not in the static colors are added into the system palette, until all 256 entries are used up. A map is then built to translate colors in the logical palette to indexes in the system palette, which will be used in translating pixel color into color indexes in the frame buffer. A background palette is treated with less respect; nothing is removed from the system palette, color can be added only when there is an unused slot in the system palette. So the foreground palette is meant for the current foreground window having the input focus, and the background palette is meant for all other windows that are still on the screen and want to have a decent display.

Here is a code fragment to create a halftone palette, select it, realize it, and display the web-color chart once more.

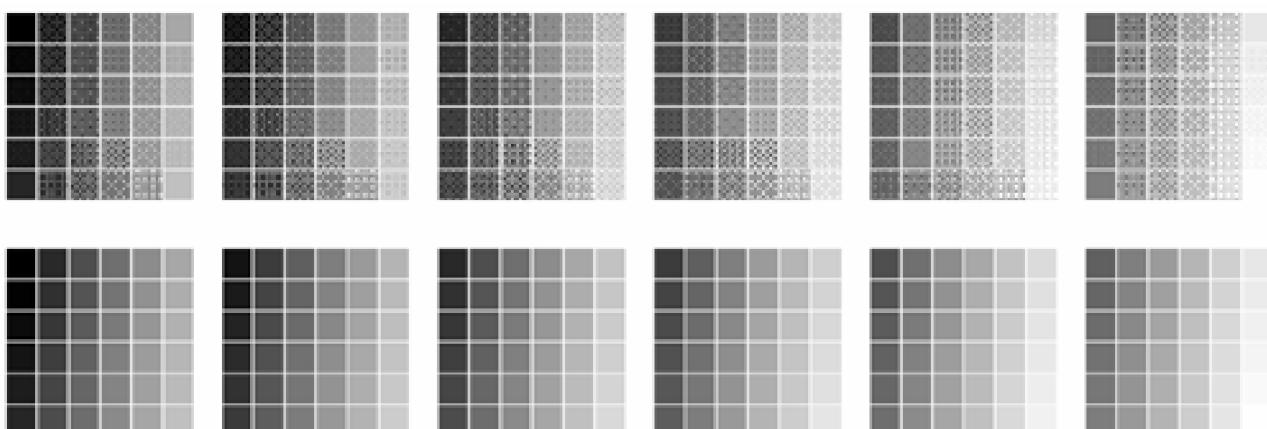
```
void TestHalftonePalette(HDC hDC, HINSTANCE hInstance)
{
    HPALETTE hPal = CreateHalftonePalette(hDC);
    HPALETTE hOld = SelectPalette(hDC, hPal, FALSE);
    RealizePalette(hDC);

    WebColors(hDC, 10, 10, 0);
    WebColors(hDC, 10, 130, 1);

    SelectPalette(hDC, hOld, TRUE);
    DeleteObject(hPal);
}
```

The results are shown in [Figure 13-3](#). The top part, using the RGB macro, still uses dithering with a few solid colors. GDI is making use of only the 20 static colors. The bottom part, using the PALETTERGB macro, displays all 216 colors as solid colors.

Figure 13-3. Displaying web-safe colors using the halftone palette.



If the bForceBackground flag is TRUE and colors in the halftone palette are not the current system palette, the system palette will not be changed; GDI only tries to approximate the request using the closest match with colors currently in the system palette.

As with the system palette, GDI allows an application to query the color table of an existing logical palette. This can be done using the GetPaletteEntries function, which returns the same PALETTEENTRY table as for GetSystemPaletteEntries.

The halftone palette contains 256 colors; 216 of them belong to the web-safe color family—that is, colors with an RGB value divisible by 51, including 6 grayscale colors. Twenty-five more grayscale colors are present in the halftone palette, so the half tone palette can handle 31 different grayscale levels. The remaining 13 colors are from static colors for several color schemes.

Creating a Customized Palette

An application is not limited to using only the default palette and the halftone palette. It can create customized palettes using the CreatePalette function. CreatePalette returns a handle to a logical palette, which can then be selected and realized in the same way as a halftone palette.

CreatePalette accepts a pointer to a LOGPALETTE structure, which contains a version number, a count, and a variable-size array of PALETTEENTRY structures. The palette version still is 0x0300, not changed after it was first introduced in Windows 3.0. The number of entries in the LOGPALETTE structure can vary greatly. If you're creating a palette for a monochrome DIB, only two colors are needed. An 8-bpp DIB may need 256 colors. Windows NT-based systems limit the number to 1024. A PALETTE ENTRY structure is used to describe each color in the palette. Its first three fields normally specify the color's RGB intensity. The peFlags field specifies how the entry should be treated in realizing the palette. [Table 13-2](#) summarizes the four possible values for peFlags.

Table 13-2. PALETTEENTRY peFlags Field

Value	Meaning
0	Normal. Match RGB color with the system palettes; add to it if missing.
PC_RESERVED	Reserve one slot in the system palette, which may be used for animation. Don't match other colors with this entry.
PC_EXPLICIT	No change to the system palette. The first two bytes of the PALETTEENTRY structure form an index to the system palette.
PC_NOCOLLAPSE	Match with the system palette colors only if no empty slot is available; otherwise, use a new slot.

Recall that normally the system palette contains the 20 static colors, which can't be replaced. So if an application wants to realize a 256-entry logical palette, it's possible that some colors can't be added to the system palette. GDI fulfills the requirement in the order in which entries appear in the LOGPALETTE structure. It's advisable to put important colors first in the LOGPALETTE structure.

The following routine creates a 256-color grayscale palette, with no special requirement. If such a palette is selected and realized when the system is using the 20 static colors, the 16 colors at the end of the table can't be realized using solid colors. If an application, say, a medical application displaying an x-ray image, wants to display the true 256 grayscale image, it could use SetSystemPaletteUse(hDC, SYSPAL_NOSTATIC) to reduce the static colors to

contain black and white only. After doing so, the whole Windows UI is now in grayscale; an application needs to make sure the system colors are set properly to make the screen still readable.

```
HPALETTE CreateGrayscalePalette(void)
{
    LOGPALETTE * pLogPal = (LOGPALETTE *) new BYTE[sizeof(LOGPALETTE)
        + 255 * sizeof(PALETTEENTRY)];

    pLogPal->palVersion = 0x0300;
    pLogPal->palNumEntries = 256;
    for (int i=0; i<256; i++)
    {
        PALETTEENTRY entry = { i, i, i, 0 };

        pLogPal->palPalEntry[i] = entry;
    }

    HPALETTE hPal = CreatePalette(pLogPal);

    delete [] (BYTE *) pLogPal;

    return hPal;
}
```

Creating a logical palette with the PC_EXPLICIT flag is interesting. Its purpose is not to change the system palette, but to allow colors in the system palette to be used as an index to a logical palette. When a logical palette with a PC_EXPLICIT flag is selected and realized, colors specified using the PALETTEINDEX macro will be mapped to the system palette indexes given in the LOGPALETTE structure; even the PALETTERGB macro behaves the same way as the PALETTEINDEX.

After a palette is created, ResizePalette can be used to increase or decrease the number of colors in it. When the size of a palette is reduced, removed entries are not available for use, but the remaining entries are still unchanged. When the size of a palette is increased, new entries are set to black. SetPaletteEntries can be used to initialize these new entries.

13.3 PALETTE MESSAGES

When a window realizes a foreground logical palette, nonstatic colors are removed from the system palette, and new colors from the logical palette are added. If any application that uses nonstatic colors in the old system palette is still visible on the screen, even as an inactive window, its display is totally messed up. Red may be turned to green; green may be turned to yellow. To allow multiple windows to share the system palette nicely, the Windows palette manager broadcasts several palette messages to top-level windows to notify them of important changes.

The WM_QUERYNEWPALETTE

During the time when a window is inactive, other windows may have changed the system palette, so its display may have been compromised. When a window is about to receive a keyboard focus, Windows sends a WM_QUERYNEWPALETTE to it to give it a chance to make things right.

If the window is using a nondefault palette, it should realize the palette as a foreground palette and repaint the whole window to restore it to its best condition. The palette the window is using should be precreated and stored in a window class member variable, or global variable. The routine below demonstrates how to handle the message.

```
BOOL KWindow::OnQueryNewPalette(void)
{
    if ( m_hPalette==NULL )
        return FALSE;

    HDC     hDC = GetDC(m_hWnd);
    HPALETTE hOld= SelectPalette(hDC, m_hPalette, FALSE);

    BOOL changed = RealizePalette(hDC) != 0;
    SelectPalette(hDC, hOld, FALSE);
    ReleaseDC(m_hWnd, hDC);

    if ( changed )
        InvalidateRect(m_hWnd, NULL, TRUE); // repaint
    return changed;
}
```

A member variable m_hPalette is added to our top-level window class, which is initialized to NULL, unless a derived window wants to use a palette. If you're running in high color or true color mode, or you want to use only the static colors, m_hPalette can stay as NULL. When a WM_QUERYNEWPALETTE message is received by the window's message procedure, it's forwarded to the KWindow::OnQueryNewPalette, or a routine overriding it. The OnQueryNewPalette method creates a new device context handle, selects the palette as a foreground palette and realizes it. If the realization was successful, which means the device supports the palette, the window's client is invalidated so that it will be repainted with the right colors. The routine returns TRUE if the palette is realized, and FALSE otherwise.

WM_PALETTEISCHANGING

When an application is about to realize its logical palette, Windows broadcasts a WM_PALETTEISCHANGING message to top-level windows, informing them that the system palette is about to change. But in no way is the realization delayed for any confirmation.

When a foreground window realizes its palette, changes in the system palette can make background windows turn really ugly. The WM_PALETTEISCHANGING message is supposed to give the background window a chance to prepare for the coming changes in the system palette. For example, a window can just clear its display to a background color using a static color, such that the display will stay the same during the system palette change, and then repaint the screen using a background palette.

One MSDN article claims that this message is a holdover from an earlier design and should be ignored. We've noticed that this message is never posted on Windows 2000. For even the professional software, you will see the screen turn ugly briefly during palette switching.

WM_PALETTECHANGED

After an application changes the system palette, the display of all windows except the foreground window may be totally messed up, and the WM_PALETTECHANGED message is broadcast to all overlapped and popup windows in the system. Now these windows should really respond to this message and try to repair their screen display as much as possible.

The wParam parameter of the WM_PALETTECHANGED message is the handle of the window changing the system palette. The window handling this message should check the handle to see if it, itself, is the window making the change; if so, no action is needed. Otherwise, there are two ways to repair its display.

For a window, a quick way to repair the damage is to realize its logical palette as the background palette and call the GDI function UpdateColors to do a pixel-level repair.

BOOL UpdateColors(hDC);

UpdateColors goes through every pixel in the device surface, mapping color indexes to the original system palette to a new color index best matching its color according to the new system palette. You can imagine that the internal implementation of UpdateColors builds a mapping table from the original system palette to the new system palette, and then loops through each pixel to translate them.

Because UpdateColors works from the device frame buffer, which holds an approximation of the intended picture, a multiple application of UpdateColors will gradually degrade the picture significantly. For example, if the original picture being displayed is a color picture, after an application switches to a grayscale palette, UpdateColors maps everything to grayscale. But when another window realizes a halftone palette, UpdateColors will not be able to map the grayscale image back to a color image.

The second way to respond to a WM_PALETTECHANGED message is to repaint the window, with its palette realized as the background palette. Recall that the background palette can't remove any entry from the system palette, but it can make use of unused entries and match its logical colors to the existing entries. If the new system palette is quite balanced, a decent display is still possible.

Here is our implementation of WM_PALETTECHANGED message processing. The routine checks if the current window made the system change by comparing its handle with wParam. If they are not the same and the window does have a palette, it's selected and realized. A count is kept of the number of times UpdateColors has been called. If the number is small enough, UpdateColors is called for a faster update; otherwise, a new repaint is requested to give a better display.

```
LRESULT KWindow::OnPaletteChanged(HWND hWnd, WPARAM wParam)
{
    if ( ( hWnd != (HWND) wParam ) && m_hPalette )
    {
        HDC hDC = GetDC(hWnd);
        HPALETTE hOld = SelectPalette(hDC, m_hPalette, FALSE);

        if ( RealizePalette(hDC) )
            if ( m_nUpdateCount >= 2 )
            {
                InvalidateRect(hWnd, NULL, TRUE);
                m_nUpdateCount = 0;
            }
            else
            {
                UpdateColors(hDC);
                m_nUpdateCount++;
            }
    }

    SelectPalette(hDC, hOld, FALSE);
    ReleaseDC(hWnd, hDC);
}

return 0;
}
```

A Test Program

To put everything together, here is a simple class window that displays a DIB using the halftone palette. The class shows how a logical palette is created, realized, and used in displaying the bitmap, and how palette messages are handled with the routines shown above.

[Listing 13-2](#) shows a complete DIB window derived from the KWindow class. A popup window is created by the CreateDIBWindow method, which accepts a nonpacked DIB. An option parameter is used to demonstrate the difference between using a palette vs. not using a palette, response to palette messages, and stretch bitbltng mode. The WM_CREATE message handling creates a halftone palette depending on an option parameter. The WM_PAINT message handling uses the palette to display the bitmap. The WM_PALETTECHANGED message handling does the damage control based on the same option. The WM_QUERYNEWPALETTE message handling realizes the half tone palette; and finally the WM_NCDESTROY message handling de letes the palette.

[Listing 13-2 The Palette-Correct Handling of Bitmap](#)

```
typedef enum
{
    pal_no      = 0x00, // no palette
    pal_halftone = 0x01, // use halftone palette
    pal_bitmap   = 0x02, // use DIB/DIB section palette
```

```
pal_react    = 0x04, // response to WM_PALETTECHANGED message
pal_stretchHT = 0x08, // use STRETCH_HALFTONE stretchmode
};

class KDIBWindow : public KWindow
{
const BITMAPINFO * m_pBMI;
const BYTE      * m_pBits;
int           m_nOption;

virtual LRESULT WndProc(HWND hWnd, UINT uMsg,
                      WPARAM wParam, LPARAM lParam)
{
switch ( uMsg )
{
case WM_CREATE:
m_hWnd    = hWnd;
{
HDC hDC = GetDC(m_hWnd);
if ( (m_nOption & 3)==pal_bitmap )
m_hPalette = CreateDIBPalette(m_pBMI);
else if ( (m_nOption & 3)==pal_halftone )
m_hPalette = CreateHalftonePalette(hDC);
else
m_hPalette = NULL;
ReleaseDC(m_hWnd, hDC);
}
return 0;

case WM_PAINT:
{
PAINTSTRUCT ps;

HDC hDC = BeginPaint(hWnd, & ps);
HPALETTE hOld = SelectPalette(hDC, m_hPalette, FALSE);
RealizePalette(hDC);

if ( m_nOptions & pal_stretchHT )
SetStretchBltMode(hDC, STRETCH_HALFTONE);
else
SetStretchBltMode(hDC, STRETCH_DELETESCANS);

StretchDIBits(hDC, 10, 10, m_pBMI->bmiHeader.biWidth,
m_pBMI->bmiHeader.biHeight,
0, 0, m_pBMI->bmiHeader.biWidth,
m_pBMI->bmiHeader.biHeight,
m_pBits, m_pBMI, DIB_RGB_COLORS, SRCCOPY);

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

```
}

return 0;

case WM_PALETTECHANGED:
    if ( m_nOption & pal_react )
        return OnPaletteChanged(hWnd, wParam);
    break;

case WM_QUERYNEWPALETTE:
    return OnQueryNewPalette();

case WM_NCDESTROY:
    DeleteObject(m_hPalette);
    m_hPalette = NULL;
    return 0;
}

return DefWindowProc(hWnd, uMsg, wParam, lParam);
}

public:

void CreateDIBWindow(HINSTANCE hInst, const BITMAPINFO * pBMI,
                     const BYTE * pBits, int option)
{
    if ( pBMI==NULL )
        return;

    m_nOption = option;
    m_pBMI   = pBMI;
    m_pBits  = pBits;

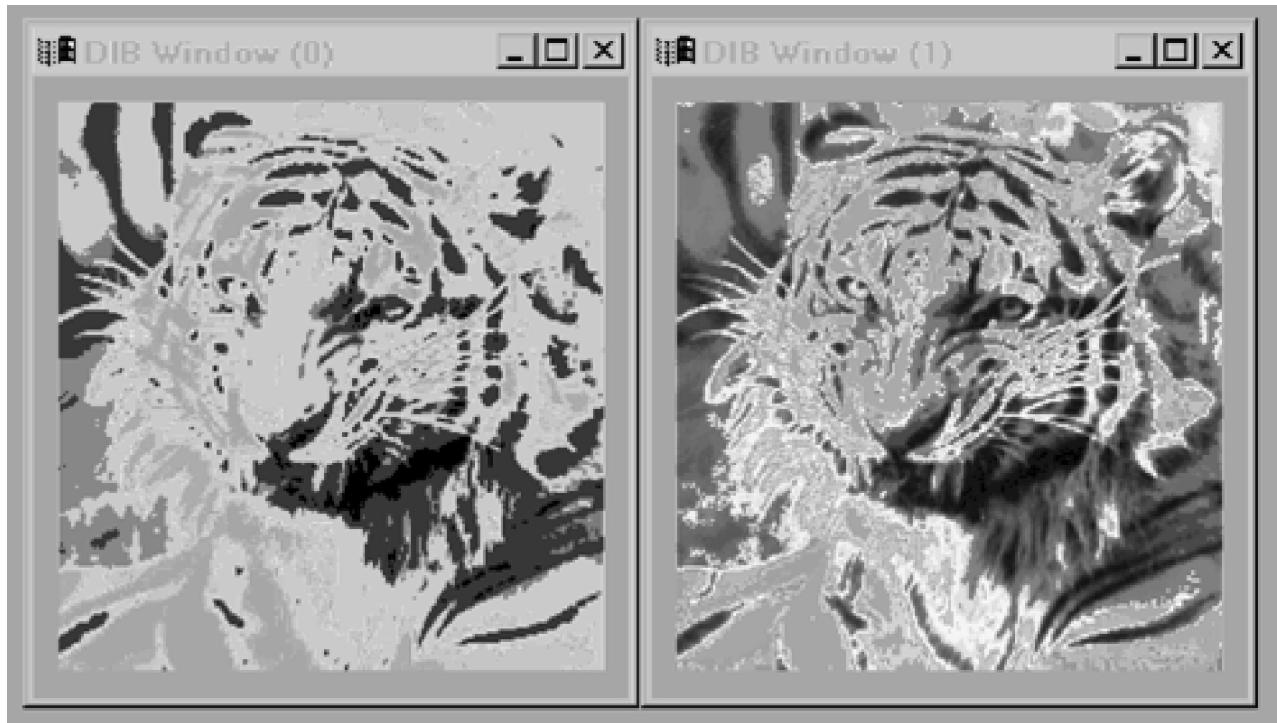
    TCHAR title[32];
    wsprintf(title, _T("DIB Window (%d)"), m_nOption);

    CreateEx(0, _T("DIBWindow"), title,
             WS_OVERLAPPEDWINDOW | WS_CLIPCHILDREN,
             CW_USEDEFAULT, CW_USEDEFAULT, m_pBMI->bmiHeader.biWidth + 28,
             m_pBMI->bmiHeader.biHeight + 48, NULL, NULL, hInst);
    ShowWindow(SW_NORMAL);
    UpdateWindow();
}

};
```

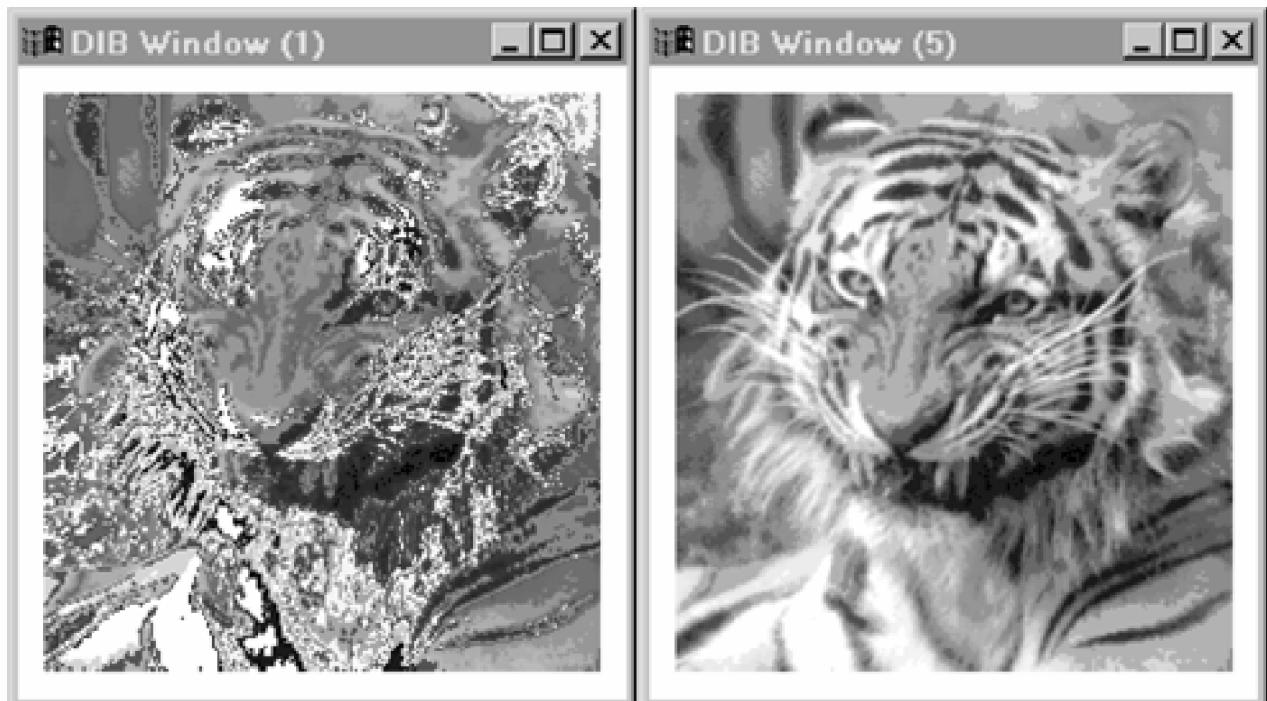
Figure 13-4 shows two pictures. The picture on the left shows the display result without the halftone palette, that is, option=pal_no. The color pixels in the bitmap are matched with the 20 static colors, generating a very grayish and flat picture. The picture on the right shows the result using the halftone palette (option=pal_halftone), a very colorful and quite smooth display; although we can't see the color here in the book, we can see clearly the level of colors.

Figure 13-4. Displaying a DIB without and with the halftone palette.



[Figure 13-5](#) compares the palette message handling. The picture on the left (option=pal_halftone) ignores the WM_PALETTECHANGED message, thus giving up the chance to repair the display when the system palette is changed. The picture on the right (option=pal_halftone | pal_react) takes the opportunity to either update the color or redraw the picture using the background palette. The difference is remarkable. On the screen, you can notice some small difference using a foreground palette vs. a background palette, but again the pictures on the paper are almost the same.

Figure 13-5. Displaying a DIB without and with WM_PALETTECHANGED handling.



When a top-level window has child windows, especially child windows for MDI (multiple document interface) application, the palette messages need to be forwarded properly, because the child windows will not get these messages themselves.

The WM_QUERYNEWPALETTE message is sent to a top-level window only when it gets input focus. For a MDI main frame window, this message should be forwarded to the active MDI child window. If the MDI child windows use different palettes, any child window getting the focus should have a chance to realize its palette as a foreground palette.

The WM_PALETTECHANGED message is also sent only to top-level windows. For a MDI frame window, it should forward the message to all of its child windows, to give all of them a chance to respond to the system palette change.

[< BACK](#) [NEXT >](#)

13.4 PALETTE AND BITMAPS

Compared with vector graphics, which uses pens and brushes, bitmap graphics has more problems with palette-based display modes. For example, the usual loading of a bitmap using LoadBitmap is not acceptable, because only system static colors are used to approximate an image that can have thousands of colors. For bitmaps that come with a color table, it needs to be translated to the Windows logical palette and used properly to generate the best result. To display a high color or true color bitmap on a 256-color display, the best result can be achieved only by generating the best palette and halftoning the image according to the palette. This section will discuss the common issues with displaying bitmaps in palette-based display modes.

The Device-Dependent Bitmap and Palette

The easiest way to convert a bitmap in a BMP format to a DDB is using LoadBitmap or LoadImage. LoadBitmap converts a BMP file attached as a resource in an EXE/DLL to a DDB. LoadImage converts either a bitmap resource or an external BMP file to a DDB, although LoadImage can load the image as a DIB section, too. Neither of them has any device context as an input parameter, or a logical palette. When used to generate a DDB, LoadBitmap and LoadImage use only the 20 static colors. All color pixels in the bitmap are replaced with the closest matching color from this small pool of colors. The picture on the left in [Figure 13-4](#) shows an example.

To generate a colorful DDB, a logical palette is needed to control the color conversion from a DIB to a DDB. The palette could be the halftone palette, a custom palette, or generated from the system palette. [Listing 13-3](#) shows a new bitmap loading routine that handles a palette.

Listing 13-3 A Palette-Controlled DDB Loading

```
BYTE * GetDIBPixelArray(BITMAPINFO * pDIB)
{
    return (BYTE *) & pDIB->bmiColors[GetDIBColorCount(pDIB->bmiHeader)];
}

// create a logical palette with all current system palette colors
HPALETTE CreateSystemPalette(void)
{
    LOGPALETTE * pLogPal = (LOGPALETTE *) new char[sizeof(LOGPALETTE)
        + sizeof(PALETTEENTRY) * 255];

    pLogPal->palVersion = 0x300;
    pLogPal->palNumEntries = 256;

    HDC hDC = GetDC(NULL);

    GetSystemPaletteEntries(hDC, 0, 256, pLogPal->palPalEntry);

    ReleaseDC(NULL, hDC);
```

```
HPALETTE hPal = CreatePalette(pLogPal);
delete [] (char *) pLogPal;

return hPal;
}

// Load a DIB from resource or from file
BITMAPINFO * LoadDIB(HINSTANCE hInst, LPCTSTR pBitmapName,
    bool & bNeedFree)
{
    HRSRC     hRes = FindResource(hInst, pBitmapName, RT_BITMAP);
    BITMAPINFO * pDIB;

    if ( hRes )
    {
        HGLOBAL hGlobal = LoadResource(hInst, hRes);
        pDIB = (BITMAPINFO *) LockResource(hGlobal);

        bNeedFree = false;
    }
    else
    {
        HANDLE handle = CreateFile(pBitmapName, GENERIC_READ,
            FILE_SHARE_READ, NULL, OPEN_EXISTING,
            FILE_ATTRIBUTE_NORMAL, NULL);

        if ( handle == INVALID_HANDLE_VALUE )
            return NULL;
        BITMAPFILEHEADER bmFH;

        DWORD dwRead = 0;
        ReadFile(handle, & bmFH, sizeof(bmFH), & dwRead, NULL);

        if ( (bmFH.bfType == 0x4D42) && (bmFH.bfSize<=
            GetFileSize(handle, NULL)) )
        {
            pDIB = (BITMAPINFO *) new BYTE[bmFH.bfSize];

            if ( pDIB )
            {
                bNeedFree = true;
                ReadFile(handle, pDIB, bmFH.bfSize, & dwRead, NULL);
            }
        }
        CloseHandle(handle);
    }

    return pDIB;
}
```

```
// load a BMP resource or file under the control of a palette
HBITMAP PaletteLoadBitmap(HINSTANCE hInst, LPCTSTR pBitmapName,
    HPALETTE hPalette)
{
    bool bDIBNeedFree;
    BITMAPINFO * pDIB = LoadDIB(hInst, pBitmapName, bDIBNeedFree);

    int width    = pDIB->bmiHeader.biWidth;
    int height   = pDIB->bmiHeader.biHeight;

    HDC hMemDC   = CreateCompatibleDC(NULL);
    HBITMAP hBmp = CreateBitmap(width, height,
        GetDeviceCaps(hMemDC, PLANES),
        GetDeviceCaps(hMemDC, BITSPIXEL), NULL);

    HGDIOBJ hOldBmp = SelectObject(hMemDC, hBmp);

    HPALETTE hOld = SelectPalette(hMemDC, hPalette, FALSE);
    RealizePalette(hMemDC);
    SetStretchBltMode(hMemDC, HALFTONE);
    StretchDIBits(hMemDC, 0, 0, width, height, 0, 0, width, height,
        GetDIBPixelArray(pDIB), pDIB, DIB_RGB_COLORS, SRCCOPY);

    SelectPalette(hMemDC, hOld, FALSE);
    SelectObject(hMemDC, hOldBmp);
    DeleteObject(hMemDC);

    if ( bDIBNeedFree )
        delete [] (BYTE *) pDIB;

    return hBmp;
}
```

The `PaletteLoadBitmap` has an extra parameter compared with `LoadBitmap`, a logical palette handle. A bitmap is loaded as a DIB first. The logical palette handle is then selected into a memory DC before a loaded DIB is converted to a DDB, so the DDB generated can use all the colors in the logical palette. Routine `LoadDIB` handles loading a bitmap from a resource or from an external file as a packed DIB. Routine `Create SystemPalette` is a helper routine to create a logical palette with all the colors in the current system palette.

The logical palette handle passed to `PaletteLoadBitmap` should be the palette used when the bitmap is going to be displayed. For example, if an application is a game program that uses the halftone palette, bitmaps in the game should be loaded with the halftone palette. The main window of the program should handle the palette messages to make sure the halftone palette is selected when the bitmaps are drawn.

DDB bitmaps are also widely used on toolbars, buttons, controls, menus, etc. These bitmaps are different from the ones an application draws in its window client area. Their drawing is normally handled by the operating system, unless they are set to be owner drawn. The operating system is using the default palette to draw them, so if an application wants to use more than the 20 static colors, the colors in the bitmap must match the current system

palette. In other words, whenever the system palette changes, these bitmaps need to be regenerated and reset.

Here is a routine that enables displaying a toolbar with more than 20 colors. It is implemented in the KToolbarB class, which is derived from the KToolbar class. The KToolbarB::SetBitmap routine should be called every time the system palette changes. It loads a bitmap using the current system palette, and uses the TB_REPLACE BIT MAP message to replace the toolbar bitmap currently being used. Now you can display more colors in a toolbar for a 256-color display mode.

```
BOOL KToolbarB::SetBitmap(HINSTANCE hInstance, int resourceId)
{
    HPALETTE hPal = CreateSystemPalette();
    HBITMAP hBmp = PaletteLoadBitmap(hInstance,
        MAKEINTRESOURCE(resourceId), hPal);
    DeleteObject(hPal);

    if ( hBmp )
    {
        TBREPLACEBITMAP rp;

        rp.hInstOld = m_ResInstance;
        rp.nIDOld = m_ResId;
        rp.hInstNew = NULL;
        rp.nIDNew = (UINT) hBmp;
        rp.nButtons = 40;

        SendMessage(m_hWnd, TB_REPLACEBITMAP, 0, (LPARAM) & rp);

        if ( m_ResInstance==NULL )
            DeleteObject( (HBITMAP) m_ResId);

        m_ResInstance = NULL;
        m_ResId = (UINT) hBmp;

        return TRUE;
    }
    else
        return FALSE;
}
```

Device-Independent Bitmaps and Palette

Unlike device-dependent bitmaps, each device-independent bitmap has complete color information that allows it to be displayed on any device. For high color or true color images, each pixel has complete color information; for other bitmaps, indexes are mapped to RGB values through a color table. The main problem in displaying DIBs on a palette-based system is what palette to use when displaying such a bitmap.

Displaying DIBs using the default palette allows only the 20 static colors. A half-tone palette is good for displaying

business graphics with highly saturated and well-distributed colors. For bitmaps with uneven color distribution in the RGB color space, a customized palette can do a better job than general-purpose palettes like the halftone palette.

For bitmaps with no more than 256 colors, the color table in the bitmap can be easily converted to a logical palette. For high color or true color bitmaps, Windows allows an application to attach a color table for displaying on palette-based devices, although it's not known that any such application does exist.

[Listing 13-4](#) shows a routine that generates a logical palette from a DIB's color table.

Listing 13-4 Convert DIB Color Table to a Logical Palette

```
HPALETTE CreateDIBPalette(BITMAPINFO * pDIB)
{
    BYTE * pRGB;
    int nSize;
    int nColor;

    if ( pDIB->bmiHeader.biSize==sizeof(BITMAPCOREHEADER) ) // OS/2
    {
        pRGB = (BYTE *) pDIB + sizeof(BITMAPCOREHEADER);
        nSize = sizeof(RGBTRIPLE);
        nColor = 1 << ((BITMAPCOREHEADER *) pDIB)->bcBitCount;
    }
    else
    {
        nColor = 0;

        if ( pDIB->bmiHeader.biBitCount<=8 )
            nColor = 1 << pDIB->bmiHeader.biBitCount;

        if ( pDIB->bmiHeader.biClrUsed )
            nColor = pDIB->bmiHeader.biClrUsed;

        if ( pDIB->bmiHeader.biClrlImportant )
            nColor = pDIB->bmiHeader.biClrlImportant;

        pRGB = (BYTE *) & pDIB->bmiColors;
        nSize = sizeof(RGBQUAD);

        if ( pDIB->bmiHeader.biCompression==BI_BITFIELDS )
            pRGB += 3 * sizeof(RGBQUAD);
    }

    if ( nColor>256 )
        nColor = 256;

    if ( nColor==0 )
        return NULL;
```

```
LOGPALETTE * pLogPal = (LOGPALETTE *) new BYTE[sizeof(LOGPALETTE)
+ sizeof(PALETTEENTRY) * (nColor-1)];
HPALETTE hPal;

if ( pLogPal )
{
    pLogPal->palVersion = 0x0300;
    pLogPal->palNumEntries = nColor;

    for (int i=0; i<nColor; i++)
    {
        pLogPal->palPalEntry[i].peBlue = pRGB[0];
        pLogPal->palPalEntry[i].peGreen = pRGB[1];
        pLogPal->palPalEntry[i].peRed = pRGB[2];
        pLogPal->palPalEntry[i].peFlags = 0;

        pRGB += nSize;
    }

    hPal = CreatePalette(pLogPal);
}

delete [] (BYTE *) pLogPal;

return hPal;
}
```

The routine tries to locate the color table within a DIB and finds the number of colors needed to display the bitmap. Recalling that under normal conditions, only 236 colors other than the static colors can be realized, it's a good habit not to use more than 236 nonstatic colors in a DIB color table. The biClrImportant field is designed to reduce the number of colors needed. It's also recommended to sort the colors in the color table according to their frequency of usage. In case some of them need to be dropped, the least-used color should be the first to go.

The CreateDIBPalette uses a DIB color table to generate only a logical palette. We will discuss how to generate an optimal palette for high color or true color images in the next section, which covers the more general topic of reducing the number of colors in a bitmap. For the moment, our displaying code uses a halftone palette if a DIB does not contain a color table.

The effect of using a DIB color table as a palette could be drastic. [Figure 13-6](#) shows the comparison. The first picture is displayed using the halftone palette, without halftone stretching; there is visible color distortion. The second picture is displayed using the halftone palette and halftone stretching; there is a much better display with a visible halftone pattern. The picture on the bottom is displayed using a color-table generated customized palette, without halftone stretching; this is a very smooth picture.

Figure 13-6. Displaying a DIB using a halftone palette, a half tone palette with HALFTONE stretch, and a customized palette.



One big surprise is that when using a color-table-generated palette, the halftone stretching mode does not improve the display at all; instead, the result is almost the same as using a halftone palette with halftone stretching enabled.

A Palette Index in a DIB Color Table

When we display a DIB, we normally use the DIB_RGB_COLORS flag in calls like StretchDIBits. This flag tells GDI that the DIB color table really contains RGB values. GDI will match the RGB values in the color table with the colors in the logical palette, and then translate logical palette indexes to system palette indexes, which will be written to the frame buffer.

Color matching with a palette is a slow process. GDI provides two functions for an application to do color matching on its own:

```
UINT GetNearestPaletteIndex(HPALETTE hpal, COLORREF crColor);
COLORREF GetNearestColor(HDC hDC, COLORREF crColor);
```

GetNearestPaletteIndex searches through all colors in a logical palette to find the closest match for a color reference. Distance here is defined as the distance between two colors in the RGB color space. For two colors $\text{RGB}(r_1, g_1, b_1)$ and $\text{RGB}(r_2, g_2, b_2)$, their distance is defined as $\sqrt{(r_1 - r_2)^2 + (g_1 - g_2)^2 + (b_1 - b_2)^2}$. For the purpose of finding the closest match, GDI can just use the square of the distance to save the slow square-root calculation. GetNearestColor matches a color reference with the colors in the current system palette, and returns the matched color.

It would be silly for GDI to do color matching for every pixel. You can imagine that for every DIB call with DIB_RGB_COLORS, GDI will color-match every color in the color table, and use the result to handle all the pixels in the bitmap.

If the current logical palette is generated from the color table in the bitmap, GDI provides a way to eliminate the first step of color matching with the logical palette. To use this optimization, the application has to modify the DIB color table from RGB values to indexes to a logical palette, and then pass the DIB_PAL_COLORS in place of the DIB_RGB_COLORS when using the DIB. With the DIB_PAL_COLORS flag, the color table in a DIB is interpreted as a WORD-array of indexes to the logical palette.

Here is a routine that creates a new BITMAPINFO structure with a palette-index color table.

```
BITMAPINFO * IndexColorTable(BITMAPINFO * pDIB, HPALETTE hPal)
{
    int nSize;
    int nColor;

    const BYTE * pRGB = GetColorTable(pDIB, nSize, nColor);

    if ( pDIB->bmiHeader.biBitCount>8 ) // no change
        return pDIB;

    // allocate a new BITMAPINFO for modification
    BITMAPINFO * pNew = (BITMAPINFO *) new BYTE[sizeof(BITMAPINFOHEADER)
                                                + sizeof(RGBQUAD)*nColor];

    pNew->bmiHeader = pDIB->bmiHeader;

    WORD * plIndex = (WORD *) pNew->bmiColors;
    for (int i=0; i<nColor; i++, pRGB+=nSize)
        if ( hPal )
            plIndex[i] = GetNearestPaletteIndex(hPal,
                                                 RGB(pRGB[2], pRGB[1], pRGB[0]));
        else
            plIndex[i] = i;

    return pNew;
}
```

The routine accepts a BITMAPINFO pointer and a logical palette. It allocates a new BITMAPINFO structure, copies over the format and dimension information, and generates a color table with palette indexes. If a logical palette handle is not given, it's assumed that the bitmap will be displayed using the logical palette created from the color table, so we can just map entries in the color table to their sequential position in the table. If a logical palette is given, the RGB color values in the original color table are matched with this palette. The routine avoids modifying the original color table by allocating a new BITMAPINFO structure; it can be used to handle the read-only DIB read from the resource file. But the caller needs to check if a new BITMAPINFO is allocated and free it when it's not needed.

The DIB Section and Palette

After creating a DIB section, either through CreateDIBSection or through LoadImage, a GDI DIB section handle is returned. Unlike DIB, where we always know a pointer to its BITMAPINFO structure, from which we can find its color table, when given only a DIB section handle, how to access its color table is not so apparent. The only way to access its color table is to select it into a memory DC and use the following two functions:

```
UINT GetDIBColorTable(HDC hDC, UINT uStartIndex, UINT cEntries,
                      RGBQUAD * pColors);
```

```
UINT SetDIBColorTable(HDC hDC, UINT uStartIndex, UINT cEntries,
    CONST RGBQUAD * pColors);
```

GetDIBColorTable copies the color table of a DIB section to a user-supplied RGBQUAD array. SetDIBColorTable does the opposite, getting a DIB section's color table from a user-supplied RGBQUAD array. Besides the misleading names, it's hard to understand why the two functions do not accept a DIB section handle as an input parameter.

Once you have the color table, a logical palette can be generated as shown below.

```
HPALETTE CreateDIBSectionPalette(HDC hDC, HBITMAP hDIBSec)
{
    HDC hMemDC = CreateCompatibleDC(hDC);
    HGDIOBJ hOld = SelectObject(hMemDC, hDIBSec);

    RGBQUAD Color[256];

    int nEntries = GetDIBColorTable(hMemDC, 0, 256, Color);

    HPALETTE hPal = LUTCreatePalette((BYTE *) Color, sizeof(RGBQUAD),
        nEntries);

    SelectObject(hMemDC, hOld);
    DeleteObject(hMemDC);

    return hPal;
}
```

If CreateDIBSection is used to create the DIB, the application does have a valid BITMAPINFO structure, which can be used to create a logical palette in the same way as creating a logical palette from a DIB.

[< BACK](#) [NEXT >](#)

13.5 COLOR QUANTIZATION

High color or true color bitmaps normally do not have a color table embedded in their bitmap information header. So far we are using the halftone palette to display them, which normally does not generate optimal display results. The process of generating an optimal palette from a color image is called color quantization.

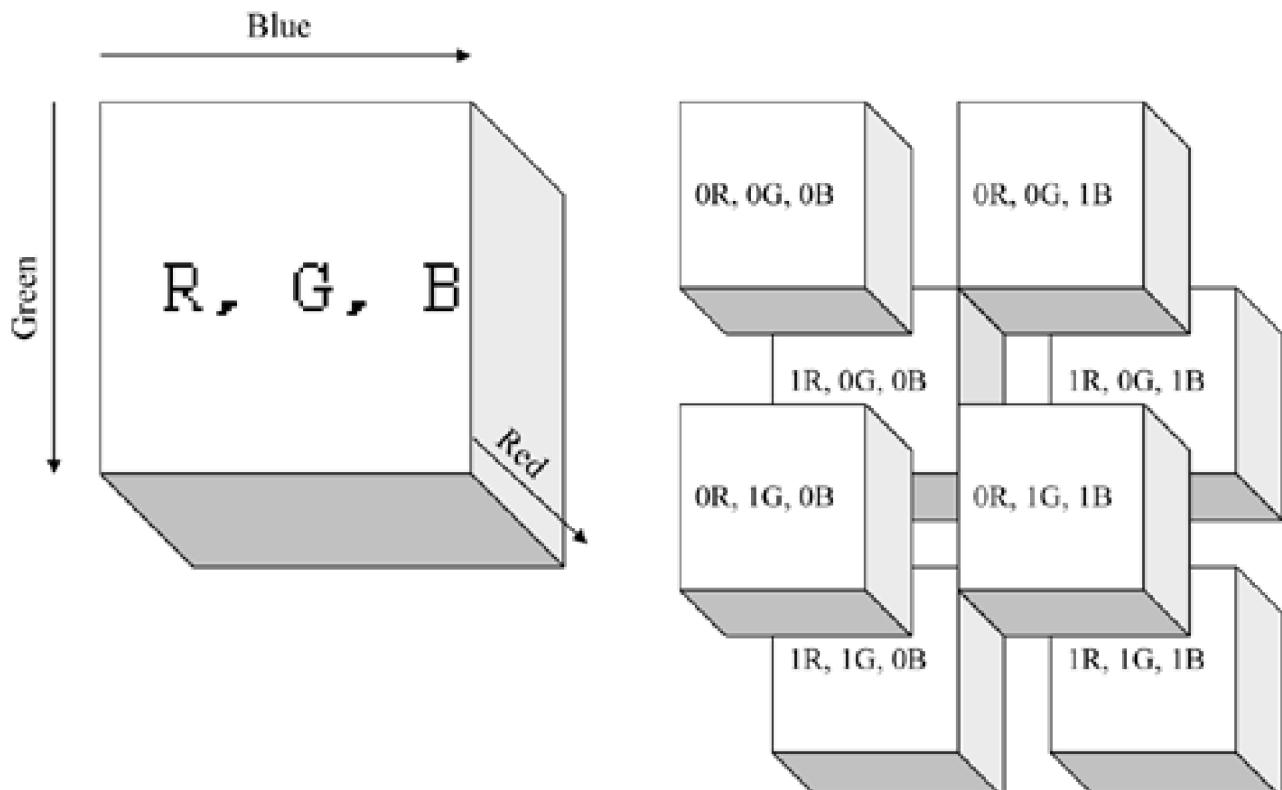
Color quantization is the process of generating a limited set of colors for an image, such that when the image is displayed using this set of colors, it's as close as possible to the original image. If the set of colors is limited to 2^N in size, each pixel in the original image can be represented using N bits of information. So color quantization is also an image-compression technique that reduces the size of images, and often a lossy one. For example, the .GIF file format supports only up to 256 colors. A high color or true color image needs to be converted to an 8-bpp format using an optimal palette.

Color quantization is a hot research area in computer graphics, so there are lots of algorithms but no optimal solution. M. Gervautz and W. Purgathofer of Austria published a paper in 1988 on using octree for color quantization, which is now considered a simple and premium-quality palette-generation method.

The octree color quantization algorithm works in three stages. In the first stage, a tree structure is built to gather information about the color statistics in the image. In the second stage, the tree is reduced by combining small nodes in it into big nodes, until the number of nodes is kept within a given limit. In the last stage, the nodes in the tree are traversed to generate a color table.

The root of the tree represents the whole RGB color space; for our purposes, this means the set of points $\text{RGB}(r, g, b)$, with r , g , and b being in $[0..255]$. The root node has 8 child nodes, each representing $\frac{1}{8}$ of the RGB color space. The partition is done by dividing the R, G, and B planes into two equal halves. [Figure 13-7](#) illustrates the division of the root node into 8 subnodes. The division is based on the first bit of the RGB components. All the pixels whose highest RGB bits are 0 are put into the first sub node, labeled "0R, 0G, 0B," where R, G, B are limited to 7 bits. All the pixels whose highest RGB bits are 1 are put into the last subnode, labeled "1R, 1G, 1B."

Figure 13-7. An octree representation of RGB color space.



The nodes in each layer of the tree are divided accordingly until the ninth level is reached. The second-level nodes, immediately below the root, are divided according to the second bits of RGB values; the third-level nodes are divided according to the third bits of RGB values; and so on.

Using an octree to represent the 24-bit RGB space is potentially huge. It has one root node, 8 second-level nodes, 64 third-level nodes, and 16.7 million ninth-level nodes. A complete octree has 19,173,961 nodes. If 50 bytes of storage are used to represent each node, such a tree needs 914 MB of memory, or mostly hard disk space. The trick is to grow the tree only when needed and trim it when memory is low. That's the main reason we are using a tree, anyway; otherwise, a huge array would be a much simpler data structure.

Besides maintaining the tree structure, each node holds information on the color pixels it represents. When a leaf node is first added, it represents a single pixel. Gradually, more pixels having exactly the same RGB values can be added to it. Nodes can be merged into a big node when running out of space, or when reducing the tree for palette generation. So each node could represent multiple pixels of different RGB values. [Listing 13-5](#) shows the KNode class.

Listing 13-5 The KNode Class for Nodes in Octree Quantization

```
class KNode
{
public:
    bool     IsLeaf;
    KNode *  Child[8];

    unsigned Pixels;
    unsigned SigmaRed;
    unsigned SigmaGreen;
    unsigned SigmaBlue;
```

```
KNode(bool leaf)
{
    IsLeaf = leaf;
    Pixels = 0;
    SigmaRed = 0;
    SigmaGreen = 0;
    SigmaBlue = 0;
    memset(Child, 0, sizeof(Child));
}

RemoveAll(void);

int PickLeaves(PALETTEENTRY * pEntry, int * pFreq, int size);
};

KNode::RemoveAll(void)
{
    for (int i=0; i<8; i++)
        if ( Child[i] )
        {
            Child[i]->RemoveAll();
            Child[i] = NULL;
        }

    delete this;
}

int KNode::PickLeaves(PALETTEENTRY * pEntry, int * pFreq, int size)
{
    if ( size==0 )
        return 0;

    if ( IsLeaf )
    {
        * pFreq = Pixels;

        pEntry->peRed = ( SigmaRed + Pixels/2 ) / Pixels;
        pEntry->peGreen = ( SigmaGreen + Pixels/2 ) / Pixels;
        pEntry->peBlue = ( SigmaBlue + Pixels/2 ) / Pixels;
        pEntry->peFlags = 0;
        return 1;
    }
    else
    {
        int sum = 0;

        for (int i=0; i<8; i++)
            if ( Child[i] )
```

```
    sum += Child[i]->PickLeaves(pEntry+sum, pFreq+sum,
                                  size-sum);

    return sum;
}

}
```

The IsLeaf member variable tells if a node is a leaf node. A leaf node is defined as a node without child nodes. Initially, only nodes on the ninth layer of the tree are leaf nodes. When nodes get merged together, an upper-layer node can become a leaf node too. The Child array holds 8 pointers to 8 subnodes of a nonleaf node. The rest of the member values hold the number of pixels and the sums of their RGB values for all the pixels in the sub-tree rooted at the current node. For example, the pixels member for the root node holds the total number of pixels in the entire tree. Note that we are using 32-bit unsigned integers for the sums, so an octree is limited to hold 2^{24} pixels.

The constructor of the KNode class is quite simple; it just resets all the member variables. The RemoveAll method removes all the nodes in the current subtree using a simple recursion. PickLeaves is the final harvesting routine. It fills a PALETTE ENTRY array with RGB values, and an integer array with color-frequency information. The routine does a simple traverse of the tree in a depth-first order and converts each leaf node in the tree to a PALETTEENTRY structure, whose RGB value is calculated as the average of all pixels' RGB values. The number of pixels represented by each node is also stored in the frequency array. This extra piece of information can be used to sort the PALETTEENTRY array according to color frequency.

[Listing 13-6](#) shows the octree class.

Listing 13-6 The Octree Class for Octree Quantization

```
class KOctree
{
    typedef enum { MAXMODE = 65536 };

    KNode * pRoot;
    int TotalNode;
    int TotalLeaf;

    void Reduce(KNode * pTree, unsigned threshold);

public:

    KOctree()
    {
        pRoot = new CNode(false);
        TotalNode = 1;
        TotalLeaf = 0;
    }

    ~KOctree()
    {
        if ( pRoot )
```

```
{  
    pRoot->RemoveAll();  
    pRoot = NULL;  
}  
}  
  
void AddColor (BYTE r, BYTE g, BYTE b);  
void ReduceLeaves(int limit);  
int GenPalette(PALETTEENTRY *entry, int * Freq, int size);  
  
void Merge(CNode * pNode, CNode & target);  
};  
  
void KOctree::AddColor (BYTE r, BYTE g, BYTE b)  
{  
    KNode * pNode = pRoot;  
  
    for (BYTE mask=0x80; mask!=0; mask>>=1) // follow path until leaf node  
    {  
        // add pixel to it  
        pNode->Pixels ++;  
        pNode->SigmaRed += r;  
        pNode->SigmaGreen += g;  
        pNode->SigmaBlue += b;  
  
        if ( pNode->IsLeaf )  
            break;  
  
        // take one bit each of RGB to form an index  
        int index = ( (r & mask) ? 4 : 0 ) +  
                    ( (g & mask) ? 2 : 0 ) +  
                    ( (b & mask) ? 1 : 0 );  
  
        // create a new node if it's a new branch  
        if ( pNode->Child[index]==NULL )  
        {  
            pNode->Child[index] = new KNode(mask==2);  
            TotalNode ++;  
  
            if ( mask==2 )  
                TotalLeaf ++;  
        }  
  
        // follow the path  
        pNode = pNode->Child[index];  
    }  
  
    for (int threshold=1; TotalNode>MAXMODE; threshold++)  
        Reduce(pRoot, threshold);
```

```
}

// Combine node with leaf-only subnodes, and no more than threshold pixels
// Combine leaf node with no more than threshold into sibling node
void KOctree::Reduce(CNode * pTree, unsigned threshold)
{
    if ( pTree==NULL )
        return;

    bool childallleaf = true;

    // recursively call all nonleaf child nodes
    for (int i=0; i<8; i++)
        if ( pTree->Child[i] && ! pTree->Child[i]->IsLeaf )
    {
        Reduce(pTree->Child[i], threshold);

        if ( ! pTree->Child[i]->IsLeaf )
            childallleaf = false;
    }

    // if all children are leaves, combined is not big enough, combine
    if ( childallleaf & (pTree->Pixels<=threshold) )
    {
        for (int i=0; i<8; i++)
            if ( pTree->Child[i] )
        {
            delete pTree->Child[i];
            pTree->Child[i] = NULL;
            TotalNode--;
            TotalLeaf--;
        }
    }

    pTree->IsLeaf = true;
    TotalLeaf++;

    return;
}

// merge small child leaf nodes
for (i=0; i<8; i++)
    if ( pTree->Child[i] && pTree->Child[i]->IsLeaf &&
        (pTree->Child[i]->Pixels<=threshold) )
    {
        KNode temp = * pTree->Child[i];

        delete pTree->Child[i];
        pTree->Child[i] = NULL;
        TotalNode--;
    }
```

```
TotalLeaf --;

for (int j=0; j<8; j++)
    if ( pTree->Child[j] )
    {
        Merge(pTree->Child[j], temp);
        break;
    }
}

void KOctree::Merge(KNode * pNode, KNode & target)
{
    while ( true )
    {
        pNode->Pixels    += target.Pixels;
        pNode->SigmaRed  += target.SigmaRed;
        pNode->SigmaGreen += target.SigmaGreen;
        pNode->SigmaBlue += target.SigmaBlue;

        if ( pNode->IsLeaf )
            break;

        KNode * pChild = NULL;

        for (int i=0; i<8; i++)
            if ( pNode->Child[i] )
            {
                pChild = pNode->Child[i];
                break;
            }

        if ( pChild==NULL )
        {
            assert(FALSE);
            return;
        }
        else
            pNode = pChild;
    }
}

void KOctree::ReduceLeaves(int limit)
{
    for (unsigned threshold=1; TotalLeaf>limit; threshold++)
        Reduce(pRoot, threshold);
}

int KOctree::GenPalette(PALETTEENTRY entry[], int * pFreq, int size)
```

```
{  
    ReduceLeaves(size);  
  
    return pRoot->PickLeaves(entry, pFreq, size);  
}
```

The member variables of the KOctree class are quite simple. The root of the tree is in pRoot, which links to the entire tree. The total numbers of nodes and leaf nodes in the tree are kept in TotalNode and TotalLeaf. The tree starts as a tree with only the root node allocated by the constructor, and the destructor removes everything in the tree.

The AddColor method is the main routine responsible for generating the tree. It accepts the red, green, and blue components of a pixel in RGB space. The color is added to the root node first; then the first bits from RGB are used to form an index to a subnode on the second layer of the tree. The color pixel is added to all the layers until we meet a leaf node. If a subnode on the path is not yet created, it's created. Note that a merged leaf node will not be expanded again.

The KOctree class has a population limitation set by the MAXNODE constant. Currently, it's set to be 65,536, roughly enough to represent the most complicated 16-bpp image precisely. The maximum allowed tree occupies around 3 MB of memory. If there are too many nodes in the tree, AddColor will call Reduce to trim the tree. The trimming is done gradually using a threshold value, which starts with one. In the first round of trimming, all leaf nodes with a single pixel will be merged. After one round of trimming, if there are still too many nodes, the threshold value is increased, and another round of trimming begins.

The Reduce method implements the trimming algorithm in three steps. First, all nonleaf subnodes are trimmed by calling Reduce recursively. After that, if all the sub nodes of the current node are leaf nodes and the total number of pixels is no more than the threshold, all the subnodes are deleted, and the current node is marked as a leaf node. Recall that AddColor adds color information to each layer in the tree, so each node holds the summary information about its subnodes. The final step of Reduce checks for any small leaf subnodes, and merges them with one of their sibling nodes.

The Merge method does the merging of sibling nodes. It simply finds a branch to a leaf node in a sibling node and adds the RGB values to the branch. A more sophisticated algorithm should check for the closest match.

The previously mentioned routines handle the construction of the tree and any trimming needed if the tree is getting too big. After a tree is generated, the ReduceLeaves method gradually trims the tree until the number of leaf nodes is within a limit. The same Reduce method is called to trim the tree with an increasing threshold value. Gradually, smaller nodes get merged into upper-level big nodes, and bigger nodes do not get merged until the threshold is raised rather high. The idea is to use limited leaf nodes to represent the color distribution in an image as accurately as possible. So nodes with a large number of pixels are favored over those with fewer pixels.

The GenPalette method is the final method to fill a PALETTEENTRY array and a frequency array. It calls ReduceLeaves to merge the trim to reduce the number of leaf nodes and KNode::PickLeaves to fill the two arrays.

What's missing now is sending each pixel in a bitmap to the KOctree class to grow a tree and then pick the leaf nodes to generate a palette. [Listing 13-7](#) shows the KPalette Gen class.

Listing 13-7 The KPaletteGen Class: The Octree Palette Generation

```
class KPaletteGen : public KPixelMapper  
{
```

```
KOctree octree;

// return true if data changed
virtual bool MapRGB(BYTE & red, BYTE & green, BYTE & blue)
{
    octree.AddColor(red, green, blue);

    return false;
}
public:

void AddBitmap(KImage & dib)
{
    dib.PixelTransform(* this);
}

int GetPalette(PALETTEENTRY * pEntry, int * pFreq, int size)
{
    return octree.GenPalette(pEntry, pFreq, size);
}

};

int GenPalette(BITMAPINFO * pDIB, PALETTEENTRY * pEntry, int * pFreq,
               int size)
{
    KImage dib;
    KPaletteGen palgen;

    dib.AttachDIB(pDIB, NULL, 0);

    palgen.AddBitmap(dib);

    return palgen.GetPalette(pEntry, pFreq, size);
}
```

The KPaletteGen class is derived from the KPixelMapper class developed in [Chapter 12](#), which handles pixel transformation of a DIB. Recall that a KPixelMapper needs only to implement the MapRGB method, which will be called for every pixel in the bitmap. The KPaletteGen::MapRGB method just adds a color pixel to a KOctree class instance. The AddBitmap method enumerates every pixel in a bitmap and calls MapRGB eventually. The GetPalette method returns the final color table.

A global function, GenPalette, is also provided to generate a color table from a packed DIB. It uses both the KImage class and the KPaletteGen class to perform its job.

The quality of the palette generated by the octree quantization algorithm is very good, even in comparison with professional image-processing software. Here is a sample 16-color color table generated for the tiger image in [Figure 13-4](#). It shows the RGB value for each entry in the color table and the number of pixels represented by the entry. As you can see, the number of pixels allocated for each entry is quite balanced.

PALETTEENTRY Pal16[] = // for the tiger picture shown in Figure 13-4

```
{  
    { 59, 52, 47 }, // 0, 3874  
    { 55, 41, 41 }, // 1, 1792  
    { 76, 51, 42 }, // 2, 2893  
    { 99, 77, 54 }, // 3, 2823  
    { 101, 97, 87 }, // 4, 5567  
    { 113, 108, 84 }, // 5, 1652  
    { 153, 113, 84 }, // 6, 5417  
    { 140, 119, 110 }, // 7, 2475  
    { 166, 136, 113 }, // 8, 4136  
    { 206, 148, 115 }, // 9, 2521  
    { 170, 154, 150 }, // 10, 2312  
    { 173, 149, 142 }, // 11, 1899  
    { 212, 173, 148 }, // 12, 3749  
    { 234, 207, 170 }, // 13, 1610  
    { 232, 222, 209 }, // 14, 2659  
    { 250, 244, 235 }, // 15, 2781  
};
```

[Figure 13-8](#) shows displaying the tiger using a 16-color, 64-color, and 236-color palette generated using the octree quantization algorithm, without halftoning.

Figure 13-8. A bitmap displayed with a 16-, 64-, and 236-color palette.



The octree color quantization algorithm can also be used for other purposes. Image editors normally provide a way to count the number of colors used in the bitmap, to provide guidance in how the image could be compressed. For true color images, counting the exact number of colors used is not an easy task, because 16.7 million different colors are possible. The octree provides a good data structure to this task. The number of colors in an image is just the number of leaf nodes in its octree representation, unless we run out of memory before the image is fully scanned. If used for color counting only, the KNode class can be simplified to reduce memory usage.

An alternative method of color counting is building a 256-by-256-by-256-bit array, each bit representing a color in the 8-8-8 RGB color space. A total of 2 MB memory is needed.

13.6 BITMAP COLOR-DEPTH REDUCTION

Now that we have a good algorithm to find the “optimal” palette of a bitmap, the next fun thing to try is reducing high-color or bitmap-color bitmaps to index-based bitmaps, or generally to reduce the color depth of a bitmap. For example, we can convert a true color image to an 8-bpp image, reducing it to one-third of its original size plus any gain from RLE compression. We can also convert an 8-bpp to a 4-bpp image.

With a color table, or a palette, the easiest way to reduce the color depth is the simple nearest-color-matching algorithm. For every pixel in a bitmap, its color is compared with all the colors in the color table; the index of the closest match becomes the pixel value in the new bitmap.

[Listing 13-8](#) shows a KColorMatch class, which implements brute-force linear searching color matching. The KColorMatch::ColorMatch method searches through a RGBQUAD array to find the closest match in RGB color space.

Listing 13-8 The KColorMatch Class: Simple Color Matching

```
class KColorMatch
{
public:
    RGBQUAD * m_Colors;
    int      m_nEntries;

    int square(int i)
    {
        return i * i;
    }

public:
    BYTE ColorMatch(int red, int green, int blue)
    {
        int dis = 0xFFFFFFFF;
        BYTE best = 0;

        if ( red<0 ) red=0; else if ( red>255 ) red=255;
        if ( green<0 ) green=0; else if ( green>255 ) green=255;
        if ( blue<0 ) blue=0; else if ( blue>255 ) blue=255;

        for (int i=0; i<m_nEntries; i++)
        {
            int d = square(red - m_Colors[i].rgbRed);
            if ( d>dis ) continue;

            d += square(green - m_Colors[i].rgbGreen);
            if ( d>dis ) continue;
        }
    }
}
```

```
d += square(blue - m_Colors[i].rgbBlue);
if ( d < dis )
{
    dis = d;
    best = i;
}
}
return best;
}

void Setup(int nEntry, RGBQUAD * pColor)
{
    m_nEntries = nEntry;
    m_Colors = pColor;
}
};
```

Based on the KColorMatch and KPixelMapper class, here is a simple class for bitmap color reduction. The KColorReduction class only supports the generation of an 8-bpp DIB, but it can be easily extended to other formats. Its main method is Convert8bpp, which allocates a new 8-bpp bitmap, calls the octree color quantization algorithm to generate an optimal color table, and then uses the KImage::PixelTransform method to call the color-matching algorithm shown in [Listing 13-9](#) for the conversion.

Listing 13-9 KColorReduction: A Nearest-Match Color-Depth Reduction

```
class KColorReduction : public KPixelMapper
{
protected:
    int      m_nBPS;
    BYTE    * m_pBits;
    BYTE    * m_pPixel;
    KColorMatch m_Matcher;

    // return true if data changed
    virtual bool MapRGB(BYTE & red, BYTE & green, BYTE & blue)
    {
        *m_pPixel ++ = m_Matcher.ColorMatch(red, green, blue);
        return false;
    }

    virtual bool StartLine(int line)
    {
        m_pPixel = m_pBits + line * m_nBPS; // first pixel of a scanline
        return true;
    }

public:
```

```
BITMAPINFO * Convert8bpp(BITMAPINFO * pDIB);
};

BITMAPINFO * CColorReduction::Convert8bpp(BITMAPINFO * pDIB)
{
    m_nBPS      = (pDIB->bmiHeader.biWidth + 3) / 4 * 4; // 8bpp scanline
    int headsize = sizeof(BITMAPINFOHEADER) + 256 * sizeof(RGBQUAD);

    BITMAPINFO * pNewDIB = (BITMAPINFO *) new BYTE[headsize +
        m_nBPS * abs(pDIB->bmiHeader.biHeight)];

    memset(pNewDIB, 0, headsize);

    pNewDIB->bmiHeader.biSize      = sizeof(BITMAPINFOHEADER);
    pNewDIB->bmiHeader.biWidth     = pDIB->bmiHeader.biWidth;
    pNewDIB->bmiHeader.biHeight    = pDIB->bmiHeader.biHeight;
    pNewDIB->bmiHeader.biPlanes    = 1;
    pNewDIB->bmiHeader.biBitCount  = 8;
    pNewDIB->bmiHeader.biCompression = BI_RGB;

    memset(pNewDIB->bmiColors, 0, 256 * sizeof(RGBQUAD));

    int freq[236];
    m_Matcher.Setup(GenPalette(pDIB, pNewDIB->bmiColors, freq, 236),
                    pNewDIB->bmiColors);

    m_pBits = (BYTE*) & pNewDIB->bmiColors[256];

    if ( pNewDIB==NULL )
        return NULL;

    KImage dib;
    dib.AttachDIB(pDIB, NULL, 0);

    dib.PixelTransform(* this);
    return pNewDIB;
}
```

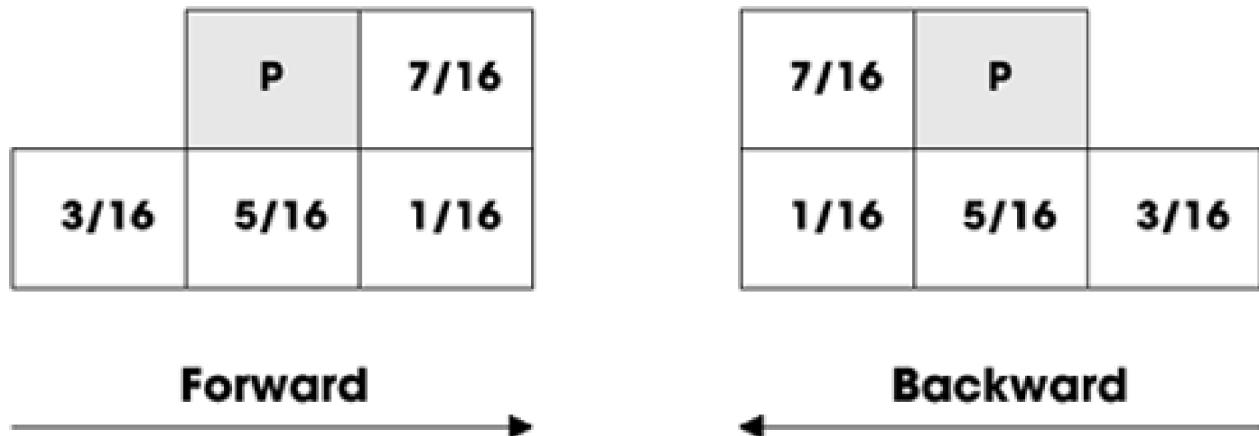
The KColorReduction class generates almost exactly the same result as GDI's bitmap display when HALFTONE stretch mode is not used. This should be no surprise, because GDI basically uses the same algorithm with a better-optimized implementation. In the HALFTONE stretch mode, GDI is able to use a dithering/halftoning pattern to generate smooth transitions between color shades. The closest-color-matching algorithm tries to find the best match for each individual pixel independently, while a halftone algorithm tries to generate pixel blocks whose average color approximates the color of the source image. The HALFTONE stretch mode is supported only on NT-based systems.

The halftone algorithm used by GDI is a simple-ordered dithering algorithm. A better algorithm for halftone is the Floyd-Steinberg error-diffusion algorithm. In an error-diffusion algorithm, each pixel's color is added with an accumulated error value, which is initially zero. The color is then matched with the color table in the usual way, and the index returned is stored in the destination bitmap. What's different is that now the error between the original color and the matched color is distributed (diffused) to neighboring pixels to influence their color matching. The

surrounding pixels will be biased by the error to generate an image that on the average is more accurate than per-pixel color matching.

In the Floyd-Steinberg error-diffusion algorithm, error is divided into four uneven parts— $3/16$, $5/16$, $1/16$, and $7/16$ —and added to four neighboring pixels. To reduce the visual patterns in dithering, odd and even scan lines are scanned in alternate directions. [Figure 13-9](#) shows the error distribution for the Floyd-Steinberg error diffusion.

[Figure 13-9. The error distribution in the Floyd-Steinberg error-diffusion algorithm.](#)



[Listing 13-10](#) shows our implementation of the error-diffusion algorithm. The KErrorDiffusionColorReduction class is derived from the KColorReduction class, basically to share the color matching and 8-bpp bitmap-generation code. Instead of overriding the pixel mapping routine, it overrides the 24-bpp scan-line processing line. The error-diffusion algorithm needs extra data to store the accumulated error and different scan orders for alternative lines; both are awkward to implement in the pixel-mapping routine. Implementing the algorithm at the scan-line level is much easier and could be faster, but we have to provide implementation for the scan lines in other formats to provide a complete solution.

[Listing 13-10 The Floyd-Steinberg Error-Diffusion Algorithm](#)

```
class KErrorDiffusionColorReduction : public KColorReduction
{
    int      * red_error;
    int      * green_error;
    int      * blue_error;

    bool     m_bForward;

    virtual bool StartLine(int line)
    {
        m_pPixel = m_pBits + line * m_nBPS; // first pixel of a scanline

        m_bForward = (line & 1) == 0;
        return true;
    }

    virtual void Map24bpp(BYTE * pBuffer, int width);
```

```
public:
    BITMAPINFO * Convert8bpp(BITMAPINFO * pDIB);
};

inline void ForwardDistribute(int error, int * curerror, int & nexterror)
{
    if ( (error<-2) || (error>2) ) // -2..2, not big enough
    {
        nexterror = curerror[1] + error * 7 / 16;

        curerror[-1] += error * 3 / 16; //      X 7/16
        curerror[ 0] += error * 5 / 16; // 3/16 5/16 1/16
        curerror[ 1] += error / 16;
    }
    else
        nexterror = curerror[1];
}

inline void BackwardDistribute(int error, int * curerror, int & nexterror)
{
    if ( (error<-2) || (error>2) ) // -2..2, not big enough
    {
        nexterror = curerror[-1] + error * 7 / 16;

        curerror[ 1] += error * 3 / 16; // 7/16 X
        curerror[ 0] += error * 5 / 16; // 1/16 5/16 3/16
        curerror[-1] += error / 16;
    }
    else
        nexterror = curerror[-1];
}

BITMAPINFO * KErrorDiffusionColorReduction::Convert8bpp(BITMAPINFO * pDIB)
{
    int extwidth = pDIB->bmiHeader.biWidth + 2;

    int * error = new int[extwidth*3];
    memset(error, 0, sizeof(int) * extwidth * 3);
    red_error = error + 1;
    green_error = red_error + extwidth;
    blue_error = green_error + extwidth;

    BITMAPINFO * pNew = CColorReduction::Convert8bpp(pDIB);

    delete [] error;
    return pNew;
}
```



```
BackwardDistribute(red - m_Matcher.m_Colors[match].rgbRed,
    red_error +i, next_red);
BackwardDistribute(green - m_Matcher.m_Colors[match].rgbGreen,
    green_error+i, next_green);
BackwardDistribute(blue - m_Matcher.m_Colors[match].rgbBlue,
    blue_error +i, next_blue);

* m_pPixel == match;

pBuffer -= 3;
}
}
}
```

The error-diffusion class has four extra member variables. Three of them are arrays of error values for the RGB channels. They are allocated from the heap in Convert8bpp and initialized to zero. Note that the initialization makes sure red_error[-1] and red_error[width] are addressable, so as to avoid boundary checking when distributing error. The m_bForward member variable tells whether we need to go forward or backward, which is set by StartLine.

Two in-line functions, ForwardDistribute and BackwardDistribute, handle error distribution for all three channels. They accept the current error value, a pointer to the current position in the error array, and return the next error value to use.

For each scan line, Map24Bpp adds per-channel error values to each pixel's color for color matching; the per-channel error is then distributed, and the routine moves to the next pixel.

The error-diffusion algorithm generates much better results than the closest-matching algorithm, and in most of the cases better results than GDI's halftone algorithm. Its added advantage is that it can use any palette, whereas GDI's halftone algorithm normally uses fewer colors.

13.7 SUMMARY

This chapter discussed how to display good-quality color graphics or documents on a graphics device with limited colors. For this we have to deal with palettes, sharing palettes with other applications, generating palettes from a bitmap color table, color quantization, and color-depth reduction.

In the near future, the palette will remain a concern for applications targeting mass markets. If an application uses more than 20 colors, the palette should be taken into consideration in design and implementation. This chapter has shown with lots of details how to display bitmaps nicely on palette-based systems. Line art should be easier than bitmaps because it normally uses limited colors. For normal applications, a halftone palette with a uniform distribution of colors is good enough. But for high-quality applications, or applications that need lots of colors at the same time, a customized optimal palette can improve the display quality a lot when compared with a halftone palette.

The palette is also supported in a DirectDraw surface to allow game programs to achieve special animation effects using palette animation techniques, to reduce memory usage, or to just improve game performance on low-end machines in general.

Finally, we're done with bitmaps and the palette, and we are ready to move to a totally new area in the next two chapters: fonts and texts.

Further Reading

Graphics GEMS, Volume I, contains an article by Michael Gervautz and Werner Purgathofer on octree quantization. *Graphics GEMS*, Volume II, has articles on half toning and color dithering techniques, optimal color quantization algorithms, color mapping, etc.

Color quantization, halftoning, and dithering algorithms are widely used in image compression, printing, etc. For example, Independent JPEG group's free JPEG compression/decompression library contains source code for color quantization, dithering, and the Floyd-Steinberg error-diffusion algorithm. A slightly modified version of it is included in the CD, and used in Chapter 17 for JPEG image displaying and printing.

Sample Program

[Chapter 13](#) has one program, which demonstrates all the topics covered here (see [Table 13-3](#)).

Table 13-3. Sample Program for Chapter 13

Directory	Description
-----------	-------------

Samples\Chapt_13\Palette	Demonstrates the system palette, the palette message handling, dithering, the halftone palette, web-safe colors, the grayscale palette, changing display mode, creating a palette from bitmap, octree quantization, error diffusion, etc.
--------------------------	---

[< BACK](#) [NEXT >](#)

Chapter 14. Fonts

Starting with this chapter, we are going to explore fonts and texts in Windows graphics programming. Fonts and their use in text printing have a long and interesting history. Long ago, in 2400 B.C., Indians started to use engraved seals; in 450 A.D. Chinese stamped seals with ink on paper, the beginning of true printing; later, in 1049, Chinese used movable type made from clay; in 1241 Koreans started to use metal type. Two centuries later, by 1452, Gutenberg opened a new era in printing with the development of a punch-and-mold system, which allowed the mass production of movable type used to reproduce pages of text. A complete set of type of one face and one size was called a font in the printing industry after that.

In 1976, a professor wanted to publish the second edition of his book published several years earlier using the same Gutenberg-style molten lead type technology. To his surprise, he found out that the old technology was phasing out and the new technology, photo-optical typesetting machines, produced disappointing results at that time. The professor refused to use immature technology to present the product of his 15 years of hard labor, so he started tackling the old typography problem using computer science. Four years later, he invented a whole new way of designing fonts using mathematical formulas implemented by computer programs, a complete computer-based typesetting system. With it he published the revision he had delayed for 4 years.

This professor is Donald E. Knuth, his font design software is METAFONT, and his typesetting package is TEX. Better still, all his hard work together with the full source code is free to anyone, so people all over the world can use his software to design fonts for any language to print books electronically.

As you can imagine, fonts and text are very complicated topics. We're devoting this chapter to fonts and the next chapter to text. This chapter will cover character sets, code pages, glyphs, fonts, TrueType fonts, and font embedding.

14.1 WHAT'S A FONT?

Since the early days of personal computing, word processing has been one of the major reasons for people using computers. During your school years and throughout your life, you have stories to write, reports to finish, resumes to edit, or books to publish.

Word processing relies mainly on the operating systems' font and text support capabilities. Fonts and text, however, are not the core part of a computer graphics system. Some core computer graphics books don't even mention them. Instead, fonts and text are applications of basic computer graphics techniques to solve a class of real-world problems. Font and text features in an operating system are normally implemented using basic graphics services like pixels, lines, curves, areas, and bitmaps. You can even implement your own font and text solution using these primitives.

The basic tool of word processing is fonts, which can be seen as templates representing characters in the language we choose to write in. Traditionally a font is defined as a complete set of type of one face and size, following its usage in the printing industry. A unit of type is a rectangular piece or block, usually of metal, having on its upper surface a letter or character in relief. Digital typology has expanded the meaning and features of a font greatly. In this section, we're going to cover the basic concepts and terminology related to fonts in the context of Windows graphics programming.

Character Set and Code Page

A character set in Windows is vaguely defined as a set of characters. A character set also has a name and a numeric identifier. For example, the default Windows character set is named ANSI_CHARSET, whose identifier is 0, and it contains characters in the 7-bit ANSI standard character set plus characters defined by Windows to represent Western languages. The DOS box uses OEM_CHARSET, whose identifier is 255, and it contains the same 7-bit ANSI character set plus extra characters defined by IBM in the old DOS days.

A character set and its single-byte identifier do not constitute a good design, especially when the Internet has opened worldwide electronic communication. A more precise concept is that of code page. A code page is an encoding scheme to represent characters in a character set using one or multiple bytes of information. So, formally speaking, a code page is a mapping from a bit sequence to a set of characters. Unlike a character set, a code page uses a two-byte numerical identifier that enables it to handle many more languages around the world.

[Table 14-1](#) lists character sets and their corresponding code pages supported by Windows operating systems. The first 14 character sets, from SHIFTJIS_CHARSET to EASTEUROPE_SET, have one-to-one correspondence with code pages. SHIFTJIS_CHARSET uses code page 932 for encoding. The JIS in it means Japanese Industry Standard. GB2312_CHARSET corresponds to code page 936, where GB is short for national standard in Chinese.

The last three character sets, ANSI_CHARSET, OEM_CHARSET, and MAC_CHARSET, have a one-to-many mapping to code pages, depending on the current system/process locale. They are mapped to different code pages depending on where your computer is located, or at least where you want your computer to believe you are located. If your default locale uses English, ANSI_CHARSET is mapped to code page 1252, OEM_CHARSET is mapped to code page 437, and MAC_CHARSET is mapped to code page 10000.

Table 14-1. Character Sets and Code Pages

Character Set Name	Character Set ID	Code Page	Usage
SHIFTJIS_CHARSET	128	932, Japanese	Japan
HANGUL_CHARSET	129	949, Korean	Korea
JOHAB_CHARSET	130	1361	
GB2312_CHARSET	134	936, Simplified Chinese	China, Singapore
CHINESEBIG5_CHARSET	136	950, Traditional Chinese	Taiwan, Hong Kong
GREEK_CHARSET	161	1253, Windows Greek	
TURKISH_CHARSET	162	1254, Windows Latin 5	Turkish
VIETNAMESE_CHARSET	163	1258, Windows Vietnamese	
HEBREW_CHARSET	177	1255, Windows Hebrew	
ARABIC_CHARSET	178	1256, Windows Arabic	
BALTIC_CHARSET	186	1257, Windows Baltic Rim	
RUSSIAN_CHARSET	204	1251, Windows Cyrillic	Slavic
THAI_CHARSET	222	874	
EASTEUROPE_CHARSET	238	1250, Windows Latin 2	Central Europe
ANSI_CHARSET	0	1252, Windows Latin 1	U.S., U.K., Canada, etc.
		1250, Windows Latin 2	Hungarian, Polish, etc.
		1256, Windows Arabic	Iraq, Egypt, Yemen, etc.
OEM_CHARSET	255	437, MS_DOS Latin 1	U.S., U.K., Canada, etc.
		852, MS_DOS Latin 2	Hungarian, Polish, etc.
		864, MS_DOS Arabic	Iraq, Egypt, Yemen, etc.
MAC_CHARSET	77	10000, Mac Roman	U.S., U.K., Canada, etc.
		10029, Mac Central Europe	Hungarian, Polish, etc.
		10007, Mac Cyrillic	Ukrainian, Russian, etc.

Most of the code pages contain 256 characters; only a single byte is needed to encode a character in these code pages. So they are called single-byte code pages or single-byte character sets. The first 128 characters in a single-byte character set are normally the same as in the 7-bit ANSI standard. The first 32 characters are nondisplayable control characters, which are followed by the space character, signs and symbols, digits, and upper-case and lower-case letters from the English alphabet. The second 128 characters vary widely from one code page to another. That is where letters from non-English alphabets, additional signs, border building characters, even the latest Euro sign are stored. [Figure 14-1](#) shows the encoding chart for code page 1252, the Window Latin 1 code page.

Figure 14-1. Windows Latin 1 code page (1252).

00	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
10	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
20	!	"	#	\$	%	_	'	()	*	+	,	-	.	/	
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
50	P	Q	R	S	T	U	V	W	X	Y	Z	[\]	^	
60	‘	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
70	p	q	r	s	t	u	v	w	x	y	z	{		}	~	□
80	€	□	,	f	„	…	†	‡	^	%	š	€	□	ž	□	
90	□	‘	’	“	”	•	—	—	™	š	»	œ	□	ž	ÿ	
A0	í	é	£	¤	¥	¡	§	“	©	ª	«	¬	-	®	—	
B0	°	±	²	³	¹	µ	¶	·	,	¹	º	»	¼	½	¾	¸
C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D0	Ð	Ñ	Ó	Ó	Ó	Ó	Ó	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	í	í	î	ï
F0	ð	ñ	ò	ó	ò	ó	ó	÷	ø	ù	ú	û	ü	ý	þ	ÿ

You can see that the second half of the code page contains Latin letters, money signs, quotation marks, trademark sign, copyright sign, etc. Several characters are not used, as marked by an empty rectangle. The first one, 0x80, was recently assigned to be the Euro sign.

As a comparison, [Figure 14-2](#) shows Windows Cyrillic code page, code page 1251. Note that the Euro sign is at a different position (0x88), because 0x80 was already taken.

Figure 14-2. Windows Cyrillic code page (code page 1251).

00	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□
10	□	□	□	□	□	□	□	□	□	□	□	□	□	□	□	—
20	!	"	#	\$	%	_	'	()	*	+	,	-	.	/	
30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
40	@	А	В	С	Д	Е	Ғ	Ғ	Ӣ	Ӣ	Ӣ	Ӣ	Ӣ	Ӣ	Ӣ	Ӣ
50	Р	Ҙ	ҙ	Қ	Ҝ	Ҫ	Ҫ	Ҫ	Ҫ	Ҫ	[\]	^	—	
60	‘	а	б	с	д	е	ғ	ғ	ӣ	ӣ	ӣ	ӣ	ӣ	ӣ	ӣ	ӣ
70	ҟ	ҟ	ҟ	ҟ	ҟ	ҟ	ҟ	ҟ	ҟ	ҟ	{		}	~	□	
80	Ҋ	Ҋ	,	Ҋ	Ҋ	„	„	†	†	€	%	Ҋ	Ҋ	Ҋ	Ҋ	Ҋ
90	҃	‘	’	“	”	•	—	—	□	Ҍ	Ҍ	Ҍ	Ҍ	Ҍ	Ҍ	Ҍ
A0	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ	Ҏ
B0	ҏ	±	I	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ	ҏ
C0	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ	Ґ
D0	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ	Ғ
E0	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ	Ҕ
F0	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ	ҕ

While single-byte character sets are enough for most of the languages in the world, three oriental languages have just too many characters to fit into single-byte encoding. Chinese uses thousands of characters, some of which were borrowed or adapted by Japanese and Korean a long time ago. These Chinese characters are called hanzi in Chinese, kanji in Japanese, and hanja in Korean. These large character sets are normally encoded using multi-bytes, so they are normally referred to as double-byte or multibyte character sets.

A multibyte character set (MBCS) uses a single byte to encode characters from the 7-bit ASCII character set and two bytes to encode characters in Chinese, Japanese, or Korean. A text string encoded in MBCS is always parsed from left to right. If the first byte (lead byte) is less than 128 (0x80), it's a single-byte character having the same meaning as the first half of the Windows Latin 1 code page (code page 1252). If the lead byte is 128 or above, only bytes within certain ranges are defined as double-byte characters. If the lead byte falls within the valid lead-byte range, the lead byte and second byte (tail byte) are checked to see if they are within the valid double-byte character set.

For code page 936, which is used in China and Singapore, the valid ranges for both the lead and tail bytes are in [0xA1..0xFE], allowing a maximum of 8836 double-byte symbols and characters. Code page 949, which is used in Korean, is a little bit more complicated. Its valid lead bytes are in [0x81..0xFE], and its valid tail bytes are in [0x41..0x5A], [0x61..0x7A], or [0x81..0xFE]. The number of allowed characters increases to 19,278. [Figure 14-3](#) shows a small portion of the traditional Chinese code page. Interestingly, the Chinese character part of code page 950 is sorted by the number of strokes in the characters. [Figure 14-3](#) shows the simplest Chinese characters having no more than four strokes.

Figure 14-3. Part of code page 950: simple Chinese characters.

A440	一	乙	丁	七	乃	九	了	二	人	儿	入	八	几	刀	刁	力
A450	匕	十	卜	又	三	下	丈	上	丫	丸	凡	久	么	也	乞	于
A460	亡	兀	刃	勺	千	叉	口	土	士	夕	大	女	子	子	乚	寸
A470	小	尤	尸	山	川	工	己	己	巳	巾	干	升	弋	弓	才	?
A4A0	?	丑	丐	不	中	丰	丹	之	尹	予	云	丶	互	五	亢	仁
A4B0	什	彳	仆	仇	仍	今	介	仄	元	允	内	六	兮	公	冗	凶
A4C0	分	切	刈	匚	勾	勿	化	匹	午	升	卅	卞	厄	友	及	反
A4D0	壬	天	夫	太	夭	孔	少	尤	尺	屯	巴	幻	廿	弌	引	心
A4E0	戈	戶	手	扎	支	文	斗	斤	方	日	曰	月	木	欠	止	歹
A4F0	毋	比	毛	氏	水	火	爪	父	爻	片	牙	牛	犬	王	丙	?

Different code pages, especially multiple-byte code pages, are quite hard to deal with in computer programs. For example, simple tasks, like moving to the next character, need to call a Windows API function CharNext, instead of just adding 1 to a pointer. Moving to the previous character is more complicated, as can be seen from the fact that the CharPrev function takes an extra parameter as the starting address of string. It can't even be completed in constant time. Converting characters between code pages is also very troublesome. The industrywide solution to these problems is the UNICODE standard.

Unicode is a double-byte worldwide character encoding developed, maintained, and promoted by the Unicode Consortium. Its members include Apple, Hewlett-Packard, IBM, Microsoft, Oracle, Sun, Xerox, etc. The Unicode standard provides the capability to encode most of the characters used for the written languages of the world. It uses a uniform 16-bit encoding, with no escapes or mode switching, which provides a maximum of 65,536 code characters.

A character in Unicode is a 16-bit value, from 0000 to FFFF in its hexadecimal format. Characters are grouped into logical areas. For example, area 01 is basic Latin, whose codes are from 0000 to 007F; area 29 is for general punctuation, whose codes are from 2000 to 206F. The largest area, area 54, the CJK unified ideographs area, contains 20,902 Chinese characters used in China, Japan, and Korea. The second largest area, area 55, contains 11,172 Hangul syllables used in Korea. [Figure 14-4](#) shows characters in the Unicode symbol area.

Figure 14-4. Unicode symbol area.

2600	☀	☁	☂	☃	⚡	★	☆	◀	▶	⌚	☊	ଓ	ଓ	ଓ	ଓ
2610	□	☑	☒	☒	□	□	□	□	□	□	□	☛	☚	☚	☚
2620	☠	࠵	࠷	࠸	࠹	࠺	࠻	࠼	࠽	࠾	࠷	࠷	࠷	࠷	࠷
2630	≡	≡	≡	≡	≡	≡	≡	≡	≡	≡	≡	࠷	࠷	࠷	࠷
2640	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷
2650	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷	࠷
2660	♠	♥	♦	♣	♠	♥	♦	♣	♣	♣	♣	♪	♪	♪	♪

Although Windows operating systems are designed to support multiple character sets and code pages, to really use individual character sets or code pages requires extra files which may not be installed by your default operating system installation. Extra language packs can be installed through the control panel Regional Settings applet, either from an operating-system CD or downloaded from the Microsoft web site.

An API function `EnumSystemCodePages` is provided to list all the code pages supported or installed on your system.

Glyph

Character sets and code pages only define the logical grouping and encoding of characters, not how the characters will look. A character itself is an abstract concept, instead of a concrete representation. Once a character is written on a piece of paper, a graphical shape called a glyph represents it. For example, the Windows Latin 1 code page puts the English capital letter A at index 0x41, but A can be written in different ways, as illustrated in [Figure 14-5](#).

Figure 14-5. Different glyphs for letter A.



Relationship between Glyph and Character

Characters and glyphs normally have a one-to-one correspondence in a font. One character is represented by a single glyph, and one glyph represents a single character. But this is not always the case. A character may be a combination of several glyphs, and a single glyph may be used in several characters. Sharing glyphs makes much more sense for Chinese or Korean Hangul characters, which can often be decomposed into several parts, although for high-quality fonts it's better to incorporate multiple versions of similar glyphs.

Glyphs for characters may also change according to the context in which characters are written. For example, special glyphs may be used when a character appears at the beginning or end of a sentence. Arabic languages use contextual glyph forms heavily. When Chinese text is written in vertical order, parentheses change orientation accordingly.

When certain combinations of characters are written next to each other, they may be converted to a single glyph.

Such a glyph is called a ligature.

Generally speaking, a character may be represented by one or more glyphs, which may be shared by multiple characters; special rules may map multiple characters to ligatures. [Figure 14-6](#) illustrates the relationship between characters and glyphs. The first row shows Latin character letter O with grave, acute, circumflex, tilde, and dieresis, followed by several Chinese characters having the same left-side radical. These show that one character may be mapped to multiple glyphs. The second row shows the AE, OE, fi, and fl ligatures used in Danish, Norwegian, French, and English. The third row illustrates how parentheses and brackets are converted to their vertical glyph when typesetting Chinese in its traditional vertical direction, which is still used for special purposes like wedding invitations. The last row shows four groups of glyphs for three Arabic characters. Each Arabic character can have four context-sensitive glyphs, for its isolated, final, initial, and medial forms.

[Figure 14-6. Relationship between characters and glyphs.](#)



Glyph Design Elements

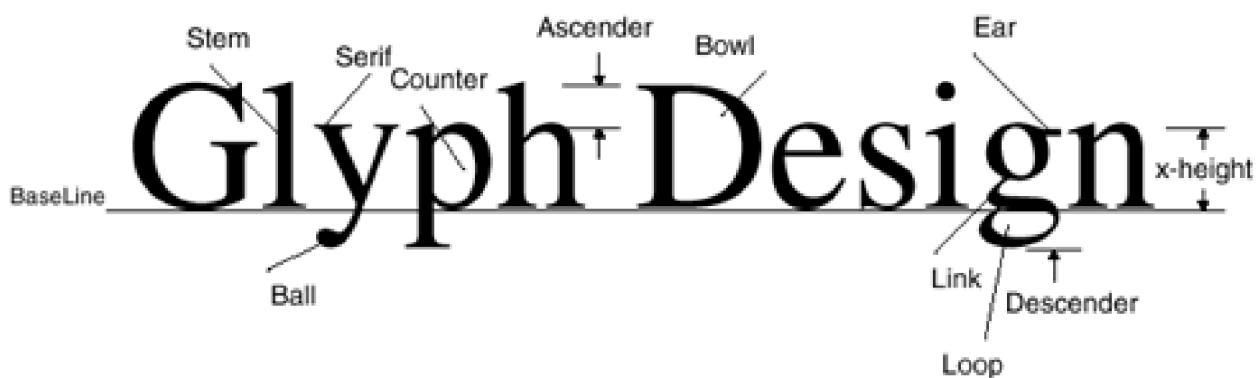
Glyphs of the same consistent design are usually grouped together. For Roman characters, glyph design elements include stroke thickness, stem design, use of serif, baseline alignment, shape of ovals and loops, amount of ascender and descender, etc.

Baseline is an imaginary line used to align glyphs vertically. For Roman characters, the bottom of most letters is aligned on the baseline, except for letters having descenders, such as f, g, j, and Q. The height of lower-case x is called x-height, which is normally the height of the body of all lower-case glyphs. Certain lower-case glyphs go higher than x-height; the extra amount is called ascender. Certain lower-case glyphs go below the baseline; the extra amount is called descender.

Glyphs may have serifs, which are small lines at the ends of the main strokes in a glyph. The circular shape at the end of a stroke in letters such as a, c, f, and y is called a ball, or a ball terminator. Counter is the name given to the fully or partially enclosed space, as in p, d, or e. Lowercase g has special design elements like ear, link, and loop. Bowl refers the basic body shape of letters like C, G, or D.

[Figure 14-7](#) illustrates several glyph design elements for glyphs with serifs.

Figure 14-7. Glyph design elements for Roman characters.



Glyphs for other languages may have similar design elements or their own unique design elements inherited from history.

Font

With character sets, code pages, and glyphs explained, we're ready to define a font. A font is a collection of glyphs having a consistent design and a mapping from characters in supported code pages to glyphs. A font may support one or more code pages; for each character in each supported code page, it maps the character to a group of glyphs which forms the graphical representation of the character.

Glyphs and character-to-glyph mapping are the basic components of a font. A font has much more information associated with it. Each font always has a full name—for example, Times New Roman Bold or Courier New Italic. Font names are normally copyrighted and protected by the copyright law. For example, Microsoft copyrights the font Wingdings; the font Courier New Italic is copyrighted by Monotype Corp.

Each font is normally stored in a physical font file, in the font subdirectory under your system directory. The control panel provides a font applet to list, examine, or install fonts on your system.

To list the fonts installed on your system, you need to walk through the registry. Here is a piece of sample code that enumerates all fonts installed on the system and feeds the information to a list view control.

```
void ListFonts(KListView * pList)
{
    const TCHAR Key_Fonts[] = _T("SOFTWARE\\Microsoft\\Windows NT"
        "\\CurrentVersion\\Fonts");

    HKEY hKey;

    if ( RegOpenKeyEx(HKEY_LOCAL_MACHINE, Key_Fonts, 0, KEY_READ,
        & hKey)==ERROR_SUCCESS )
    {
        for (int i=0; ; i++)
```

```
{  
    TCHAR szValueName[MAX_PATH];  
    BYTE szValueData[MAX_PATH];  
    DWORD nValueNameLen = MAX_PATH;  
    DWORD nValueDataLen = MAX_PATH;  
    DWORD dwType;  
  
    if ( RegEnumValue(hKey, i, szValueName, &nValueNameLen, NULL,  
        &dwType, szValueData, &nValueDataLen) !=  
        ERROR_SUCCESS )  
        break;  
  
    pList->AddItem(0, szValueName);  
    pList->AddItem(1, (const char *) szValueData);  
}  
RegCloseKey(hKey);  
}  
}
```

Typestyle and Font Family

A font's name determines the font family it belongs to and the typestyle it has. A font family is a set of fonts sharing the same set of characteristics and a common family name. For example, Times New Roman is a font family, which may contain four different fonts: Times New Roman, Times New Roman Italic, Times New Roman Bold, and Times New Roman Bold Italic.

The variation within a font family is called typestyle. Commonly seen typestyles include normal, bold, italic, condensed, underline, strikeout, proportional pitch, or fixed pitch. A typestyle may be simulated by adjusting the glyphs instead of providing a new font. For example, fonts designed by Knuth's METAFONT are controlled by a dozen or so parameters which can change the size or serif, thickness of stems and hairlines, etc. Underline and strikeout typestyles are normally simulated by GDI under Windows.

Font family provides a nice abstraction of fonts, but still applications may have too many fonts to choose from. GDI supports 8 flags to classify font families according to their basic glyph design, as shown in [Table 14-2](#).

A fixed-pitch font uses the same width for all the glyphs in the font, which is normally used in DOS windows, display program listing, or whenever vertical alignment is important. A variable-pitch font uses different widths for different glyphs; lower-case 'i' or 'l' is much thinner than 'm'. Text displayed using variable pitch is more pleasant to the human eye; that's why books, on-line helps, and web pages often use variable-pitch fonts. A variable-pitch font is also called a proportional font.

A Roman font is a font using variable stroke width and serif. Swiss fonts use variable stroke width but no serif. Roman and Swiss fonts are normally variable-pitch fonts. A modern font is a font with fixed stroke width, which is normally a fixed-width font. A script font is a font that simulates handwriting. All other fancy novel fonts are classified as decorative fonts. [Figure 14-8](#) illustrates a few Roman, Swiss, modern, script, and decorative fonts.

Figure 14-8. Classification of font families.

Roman	Roman	Roman	Roman
Swiss	Swiss	Swiss	Swiss
Modern	Modern	Modern	Modern
Script	Script	<i>Script</i>	Script
Decorative	Decorative	Decorative	DECORATIVE

Table 14-2. Font Pitch and Family Flags

Flags	Value	Usage
DEFAULT_PITCH	1	FONTS of all pitches
FIXED_PITCH	2	FONTS with the same glyph width
VARIABLE_PITCH	4	FONTS with variable glyph width
FF_DONTCARE	0 << 4	FONTS of any style
FF_ROMAN	1 << 4	FONTS with variable stroke width and serif
FF_SWISS	2 << 4	FONTS with variable stroke width, no serif
FF_MODERN	3 << 4	FONTS with fixed stroke width
FF_SCRIPT	4 << 4	FONTS designed to look like handwriting
FF_DECORATIVE	5 << 4	Fancy novel fonts

Applications will prefer to deal with font families instead of individual physical fonts, because there are fewer font families than fonts. GDI provides a function to enumerate all the font families currently available on the system: `EnumFontFamiliesEx`.

```
int EnumFontFamiliesEx(HDC hDC, LPLOGFONT lpLogFont,
    FONTENUMPROC lpEnumFontFamExProc, LPARAM lParam, DWORD dwFlags);
```

The first parameter is a handle to a device context. Some graphics devices may provide device fonts that can only be used on that particular device, such as laser printers and Postscript printers. The second parameter points to a `LOGFONT` structure, whose `lfCharset` and `lfFaceName` fields specify the character set and typeface name an application is interested in. If character set is `DEFAULT_CHARSET`, a font family supporting multiple character sets will be enumerated multiple times; if the character set is a specific character set, like `SYMBOL_CHARSET`, only font families providing symbol glyphs will be enumerated. The `lfPitchAndFamily` field in the `LOGFONT` structure must be zero. `lpEnumFontFamExProc` points to a global function which will be called for each font family enumerated—not very C++ friendly. But we have the `lParam` parameter which holds a caller-supplied data that will be passed back to the call-back function, so it can be used to bridge the C++ and Win32 worlds. The last parameter `dwFlags` must be 0.

Font-family enumeration is the key to the font list box or font selection dialog box provided by applications. It can be used to list all the font families supporting a particular character set or list all character sets supported by a typeface. [Listing 14-1](#) shows a wrapping class for this function. Its default implementation feeds enumeration results to a list view.

[Listing 14-1](#) Font-Family Enumeration

```
class KEnumFontFamily
{
    KListView * m_pList;

    int static CALLBACK EnumFontFamExProc(ENUMLOGFONTEX *lpelfe,
                                          NEWTEXTMETRICEX *lpntme, int FontType, LPARAM IParam)
    {
        if ( IParam )
            return ((CEnumFontFamily *) IParam)->EnumProc(lpelfe,
                lpntme, FontType);
        else
            return FALSE;
    }

public:
    LOGFONT    m_LogFont[MAX_LOGFONT];
    int        m_nLogFont;
    unsigned   m_nType;
    virtual int EnumProc(ENUMLOGFONTEX *lpelfe, NEWTEXTMETRICEX *lpntme,
                         int FontType)
    {
        if ( (FontType & m_nType)==0 )
            return TRUE;

        if ( m_nLogFont < MAX_LOGFONT )
            m_LogFont[m_nLogFont ++] = lpelfe->elflogFont;
        m_pList->AddItem(0, (const char *) lpelfe->elfFullName);
        m_pList->AddItem(1, (const char *) lpelfe->elfScript);
        m_pList->AddItem(2, (const char *) lpelfe->elfStyle);
        m_pList->AddItem(3, (const char *) lpelfe->elflogFont.lfFaceName);

        m_pList->AddItem(4, lpelfe->elflogFont.lfHeight);
        m_pList->AddItem(5, lpelfe->elflogFont.lfWidth);
        m_pList->AddItem(6, lpelfe->elflogFont.lfWeight);

        return TRUE;
    }

    void EnumFontFamilies(HDC hdc, KListView * pList,
                          BYTE charset = DEFAULT_CHARSET, TCHAR * FaceName = NULL,
                          unsigned type = RASTER_FONTTYPE | TRUETYPE_FONTTYPE |
                                         DEVICE_FONTTYPE)
    {
        m_pList = pList;
        m_nType = type;

        LOGFONT lf;
```

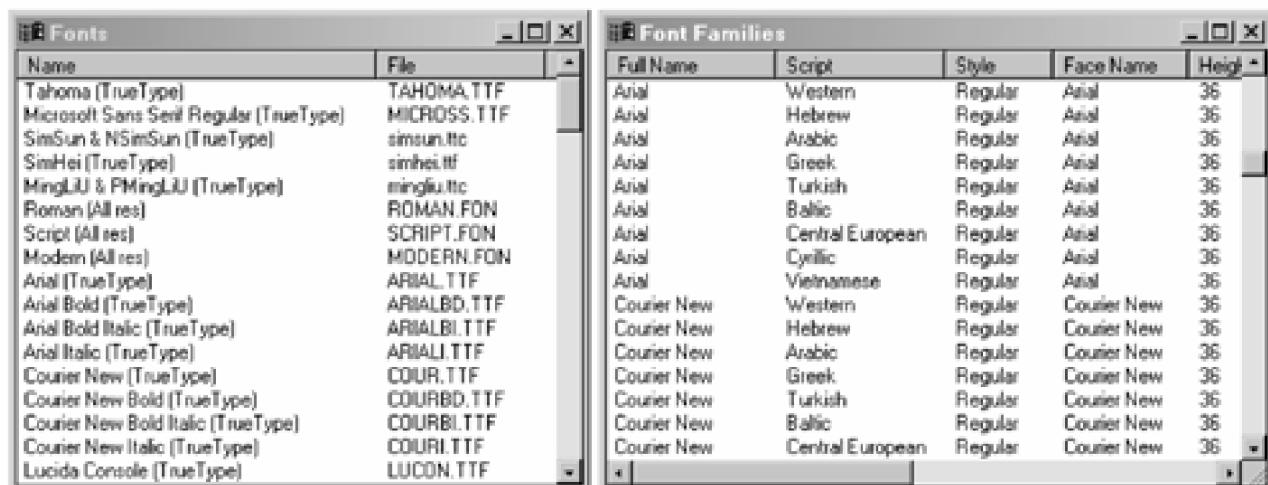
```
memset(&lf, 0, sizeof(lf));
lf.lfCharSet = charset;
lf.lfFaceName[0] = 0;
lf.lfPitchAndFamily = 0;

if (FaceName)
    _tcscpy(lf.lfFaceName, FaceName);

EnumFontFamiliesEx(hdc, &lf, (FONTENUMPROC)EnumFontFamExProc,
(LPARAM)this, 0);
}
```

[Figure 14-9](#) compares the results of font enumeration and font-family enumeration. Font enumeration through the registry lists all the physical fonts installed on the system. There are four fonts for the Arial family, four fonts for the Courier New family. Font-family enumeration lists a font family multiple times if multiple character sets are supported. For example, the Arial font family supports 9 character sets or scripts.

Figure 14-9. Font vs. font-family enumeration.



For a font family supporting double-byte character sets—that is, Chinese, Japanese, and Korean—EnumFontFamiliesEx returns two font families instead. For example, the Gulim font family supports HANGUL_CHARSET; font-family enumeration will show “Gulim” and “@Gulim” for it. Font families with a name starting with '@' add a special feature to rotate every double-byte glyph, to simulate the vertical writing system used in China, Japan, and Korea.

The font pitch and family flags shown in [Table 14-2](#) use a single byte to classify fonts and font families, which is certainly not precise enough. A more precise way to characterize fonts is the PANOSE structure, which uses 10 bytes to encode a font's main features like family type, serif style, weight, proportion, contrast, stroke variation, arm style, letterform, middle line, and x-height.

14.2 BITMAP FONTS

Glyphs in a font may be represented using different methods. The simplest way is using a bitmap to represent the pixels making up each glyph. Such fonts are called bitmap fonts. Another way is to use straight lines to represent the outline of each glyph, which gives vector fonts. Most of the fonts we use in Windows today are TrueType fonts or OpenType fonts. These fonts use a much more sophisticated method to represent the outline of glyphs and control the rendering of these outlines. We will discuss bitmap fonts in this section, Vector fonts in [Section 14.3](#), and TrueType fonts in [Section 14.4](#).

Bitmap fonts have a long history in computer display. In the old DOS days, BIOS ROM contained several bitmap fonts for different display resolutions. When an application issues a software interrupt to display a character in graphics mode, BIOS fetches the glyph data and displays it in a specified position. For the initial Windows operating systems before Windows 3.1, bitmap fonts were the only font type supported. Even today, bitmap fonts are still used as stock fonts which are heavily employed in user interface displays like menus, dialog boxes, and tool-tip messages, not to mention DOS boxes.

Even the latest Windows operating systems still use dozens of bitmap fonts. Different display resolutions use a different set of bitmap fonts to match the display resolution. For example, sserife.fon is MS Sans Serif font for 96-dpi display mode with 100% aspect ratio, while sseriff.fon is for 120-dpi display mode. When you switch display mode from small font (96-dpi) to large font (120-dpi), sseriff.fon gets enabled instead of sserife.fon. System font change affects the base units used in converting dialog box design-time coordinates to screen coordinates, so all your carefully crafted dialog boxes are messed up by the simple font change. Some bitmap fonts are so critical to system operation that they are marked as hidden files to avoid accidental deletion.

A bitmap font file usually has the .FON file extension. It uses the 16-bit NE executable file format initially used in 16-bit Windows. Within a FON file, a text string is embedded which describes the font characteristics. For example, the description for courf.fon is “FONTRES 100,120,120 : Courier 10,12,15 (8514/a res),” which contains font name, design aspect ratio (100), DPI (120x120), and point sizes supported (10, 12, 15).

For each point size supported by a bitmap font, there is one raster font resource, usually stored in a file with .FNT file extension. Multiple raster font resources can be added as a FONT type resource to the final bitmap font file. Platform SDK provides a FONTEDIT utility to modify an existing font resource file, with full source code.

Although old-fashioned, a bitmap font resource is still quite an interesting place to understand how fonts are designed and used. There are two versions of raster font resources, version 2.00 used for Windows 2.0 and version 3.00 originally designed for Windows 3.00. You may not believe that even Windows 2000 is using the version 2.00 raster font format. The fancy features provided by version 3.00 are well covered by TrueType fonts.

Each font resource starts with a fixed-size header, which contains version, size, copyright, resolution, character set, and font metrics information. For version 2.00 fonts, the Version field will be 0x200. For raster fonts, the LSB of Type is 1. Each font resource is designed for one normal resolution and a certain DPI resolution. Modern display monitors normally use square resolution—for example, 96 dpi by 96 dpi. A 10-point font for 96-dpi display will roughly be 13 (10*96/72) pixels in height. The bitmap font resource only supports one single-byte character set. It contains glyphs for all the characters within the range specified by FirstChar and LastChar. Each font resource defines a default char, which will be used to display characters not in the provided range. The BreakChar is the word break character.

```
typedef struct
{
```

```
WORD    Version; // 0x200 for version 2.0, 0x300 for version 3.00
DWORD   Size;      // Size of whole resource
CHAR    Copyright[60];
WORD    Type;      // Raster font if Type & 1 == 0
WORD    Points;    // Nominal point size
WORD    VertRes;   // Nominal vertical resolution
WORD    HorizRes;  // Nominal horizontal resolution
WORD    Ascent;
WORD    IntLeading;
WORD    ExtLeading;
BYTE   Italic;
BYTE   Underline;
BYTE   StrikeOut;
WORD   Weight;
BYTE   CharSet;
WORD   PixWidth;  // 0 for variable width
WORD   PixHeight;
BYTE   Family;    // Pitch and family
WORD   AvgWidth;  // Width of character 'x'
WORD   MaxWidth;  // Maximum width
BYTE   FirstChar; // First character defined in font
BYTE   LastChar;  // Last character defined in font
BYTE   DefaultChar; // Sub. for out-of-range chars.
BYTE   BreakChar; // Word break character
WORD   WidthBytes; // No. bytes/row of bitmap
DWORD  Device;    // Offset to device name string
DWORD  Face;      // Offset to face name string
DWORD  BitsPointer; // Loaded bitmap address
DWORD  BitsOffset; // Bitmap offset
BYTE   Reserved;  // 1 byte, not used
} FontHeader20;
```

The character table, or rather glyph table, comes after the font resource header. For the version 2.00 raster font, the character table entry for each character in the supported range contains two 16-bit integers: one for the glyph width and one for the offset to the glyph. Now you can see the serious design limitation of the version 2.00 font resource: Each font resource is limited to 64 KB in size because of the 16-bit offset. The character table contains (LastChar-FirstChar+2) entries. The extra entry is guaranteed to be blank.

```
typedef struct
{
    SHORT Glwidth;
    SHORT Gloffset;
} GLYPHINFO_20;
```

Version 2.00 supports only monochrome glyphs. Although version 3.00 is designed to support 16-color, 256-color, or even true color glyphs, no such fonts are actually found to exist in the real world. For monochrome glyphs, each pixel only needs one bit. But the order of these bits in these glyphs is nothing like the bitmap formats we've encountered. The first byte of the glyph is the first 8 pixels of the first scan line, the second byte is the first 8 pixels of the second scan line, until the first 8-pixel column of the glyph is completed. This is followed by the second 8-pixel

column, the third 8-pixel column, and so on, until the width of the glyph is fully covered. Such design was quite a common optimization technique to speed up character display.

Here is a routine to display a single glyph as a bitmap. The routine locates the GLYPHINFO table after the header, calculates the glyph index in the glyph table, and then converts the glyph to a monochrome DIB, which is then displayed using DIB function.

```
int CharOut(HDC hDC, int x, int y, int ch, FontHeader20 * pH,
            int sx=1, int sy=1)
{
    GLYPHINFO_20 * pGlyph = (GLYPHINFO_20 *) ((BYTE *)&pH->BitsOffset+5);

    if ( (ch<pH->FirstChar) || (ch>pH->LastChar) )
        ch = pH->DefaultChar;

    ch -= pH->FirstChar;

    int width = pGlyph[ch].Glwidth;
    int height = pH->PixHeight;
    struct { BITMAPINFOHEADER bmiHeader; RGBQUAD bmiColors[2]; } dib =
    {
        { sizeof(BITMAPINFOHEADER), width, -height, 1, 1, BI_RGB },
        { { 0xFF, 0xFF, 0xFF, 0 }, { 0, 0, 0, 0 } }
    };

    int bpl = ( width + 31 ) / 32 * 4;
    BYTE data[64/8*64]; // enough for 64x64
    const BYTE * pPixel = (const BYTE *) pH + pGlyph[ch].Gloffset;

    for (int i=0; i<(width+7)/8; i++)
        for (int j=0; j<height; j++)
            data[bpl * j + i] = * pPixel++;

    StretchDIBits(hDC, x, y, width * sx, height * sy, 0, 0, width, height,
                  data, (BITMAPINFO *) &dib, DIB_RGB_COLORS, SRCCOPY);

    return width * sx;
}
```

If we can convert the font resource into GDI supported bitmaps, we can display characters ourselves without GDI's text function. [Figure 14-10](#) shows all the glyphs in a raster font for 8-point and 10-point MS Serif font resource at 96 dpi.

[Figure 14-10. Glyphs in a bitmap font.](#)

D:\WINNT50\Fonts\SERIFE.FON

8 pts, 96x96 dpi, 0x13 pixel, avgw 5, maxw 11, charset 0

```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _  
' a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ |  
I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I  
i c € × ¥ | § “ ® ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘  
À Á Â Ã Ä Å Æ Ç È É Ë Ì Í Ð Ñ Ò Ó Ø × Ø Ù Ú Û Ý Þ Ø  
à á â ã ä å æ ç è é ë ì í ð ñ ò ó ø × ø ù ú û ý þ ø
```

10 pts, 96x96 dpi, 0x16 pixel, avgw 6, maxw 14, charset 0

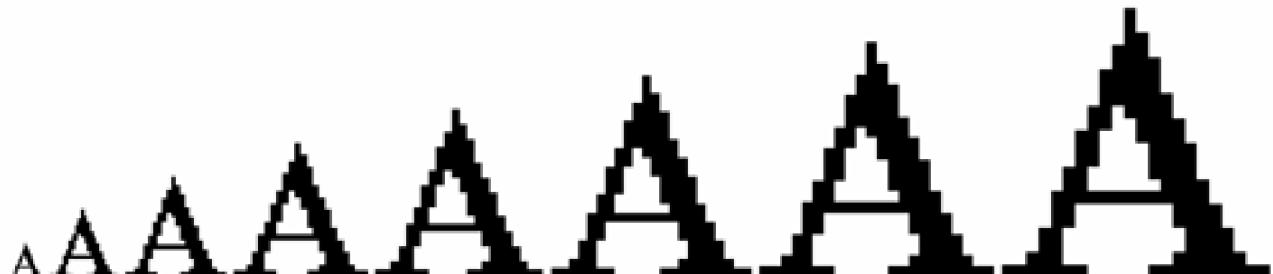
```
! " # $ % & ' ( ) * + , - . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ?  
@ A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [ \ ] ^ _  
' a b c d e f g h i j k l m n o p q r s t u v w x y z { | } ~ |  
I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I I  
i c € × ¥ | § “ ® ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘ ‘  
À Á Â Ã Ä Å Æ Ç È É Ë Ì Í Ð Ñ Ò Ó Ø × Ø Ù Ú Û Ý Þ Ø  
à á â ã ä å æ ç è é ë ì í ð ñ ò ó ø × ø ù ú û ý þ ø
```

Use the raster font as an example, we're seeing what a font really is. A raster font, as defined by the Windows version 2.00 font format, is a set of font resources designed at different point sizes; each font resource is a set of monochrome bitmap glyphs that are mapped one-to-one with a character in a single-byte character. The raster font supports a simple mapping from a character in the character set to glyph indexes in a glyph table through a range of supported characters. The glyphs for the characters can be easily converted to GDI-supported bitmap formats and displayed on a graphics device. Bitmap fonts also provide extra simple text metrics information.

Bitmap fonts are great for displaying small characters on the screen, both in terms of quality and performance, which is the main reason they still survive today. For different point sizes, bitmap fonts have to provide different sets of font resources. For example, a bitmap font used in Windows today normally provides 8-, 10-, 12-, 14-, 18-, and 24-point font resources. For other point sizes, or for a device with different resolution, glyphs need to be scaled to the required size. Bitmap scaling is always a problem, especially upscaling that requires new pixels to be generated.

[Figure 14-11](#) illustrates the result of scaling a glyph from a 24-point bitmap font, using a simple duplication method.

Figure 14-11. Scaling a raster font glyph.



[Figure 14-11](#) shows the result of integer ratio scaling in both directions, where each pixel in the glyph is duplicated the same number of times. The rough edge is clearly visible. If the scaling is a noninteger ratio, the character

displayed can have strokes with uneven thickness, as some pixels are scaled n times and others $n + 1$ times. Clearly, the scaling raster fonts does not provide good enough display/print quality; we have to find other ways to encode fonts for continuous and smooth scaling.

14.3 VECTOR FONTS

Raster fonts use bitmaps to represent glyphs, so naturally they don't scale well to big font sizes. Another simple way to represent a glyph is to store the path making up its strokes as a series of line segments. The line segments are then drawn using a pen to render a glyph. Fonts using line segments to represent glyphs are called vector fonts.

Windows' vector fonts use the same .FNT font resource format and the font header structure. For the vector fonts we are seeing today, the Version field of the FontHeader20 structure is 0x100, and Type field is 1. The main difference between a raster font and a vector font is the format of their glyph data.

For a vector font, each glyph is stored as a series of relative coordinates starting from (0, 0). If the glyph digitizing grid size is small enough, two signed bytes are used to store one point. A special marker 0x80 signals the starting of a new line segment. More precisely, the BNF syntax for a vector glyph is:

```
<vector_glyph> ::= <segment> { <segment> }
<segment>   ::= <ns_marker> { <rel_move> <rel_move> }
<ns_marker>  ::= 0x80
<rel_move>   ::= <signed_byte>
```

Knowing this, we can render vector font glyphs on our own using GDI's vector graphics functions.

```
int VectorCharOut(HDC hDC, int x, int y, int ch, const FontHeader20 * pH,
    int sx=1, int sy=1)
{
    typedef struct { short offset; short width; } VectorGlyph;

    const VectorGlyph * pGlyph = (const VectorGlyph *)
        ((BYTE *)&pH->BitsOffset + 4);

    if ( (ch<pH->FirstChar) || (ch>pH->LastChar) )
        ch = pH->DefaultChar;
    else
        ch -= pH->FirstChar;

    int width = pGlyph[ch].width;
    int length = pGlyph[ch+1].offset - pGlyph[ch].offset;

    signed char * pStroke = (signed char *) pH + pH->BitsOffset +
        pGlyph[ch].offset;

    int dx = 0;
    int dy = 0;
```

```
while ( length>0 )
{
    bool move = false;

    if ( pStroke[0]==-128 )
    {
        move = true; pStroke++; length--;
    }

    if ( (pStroke[0]==0) && (pStroke[1]==0) && (pStroke[2]==0) )
        break;

    dx += pStroke[0];
    dy += pStroke[1];

    if ( move )
        MoveToEx(hDC, x + dx * sx, y + dy * sy, NULL);
    else
        LineTo(hDC, x + dx * sx, y + dy * sy);

    pStroke += 2; length -= 2;
}

return width * sx;
}
```

Up to now, the Windows OS still uses three vector fonts: Roman, Script, and Modern. Vector fonts are normally smaller than raster fonts, because line segments can be easily scaled; only a single font resource is needed. [Figure 14-12](#) shows the first half of the glyphs from the vector script font.

Figure 14-12. Vector font glyphs.

D:\WINNT50\Fonts\SCRIPT.FON

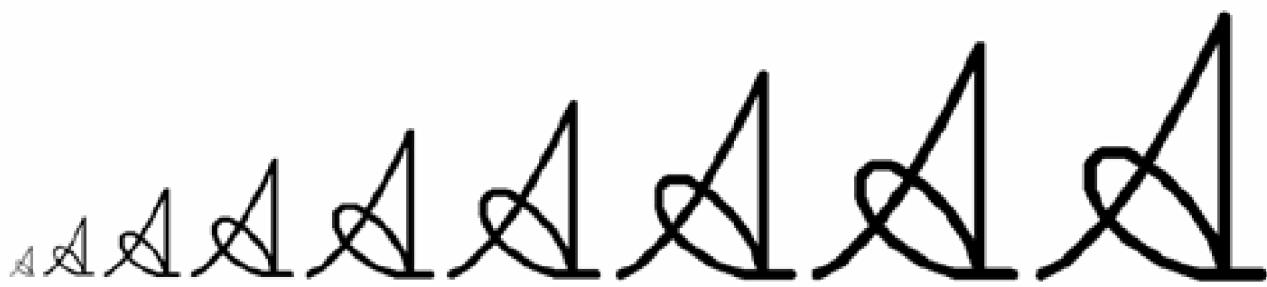
36 pts, 2x3 dpi, 0x37 pixel, avgw 17, maxw 33, charset 255

! " # \$ % & , () * + , - . /
0 1 2 3 4 5 6 7 8 9 : ; < = > ?
@ A B C D E F G H I J K L M N O
P Q R S T U V W X Y Z [\] ^ _
' a b c d e f g h i j k l m n o
p q r s t u v w x y z { | } ~ .

Compared with raster font glyphs, vector font glyphs scale quite well, although when the scaling ratio gets higher, the connection points between individual line segments become more and more visible. [Figure 14-14](#) (seen later) illustrates the scaling for the glyph for “A.”

[Figure 14-13](#) is displayed using a pen width proportional to the glyph size, unlike GDI, which uses a single-pixel pen to draw vector font glyphs. Although the use of vector fonts improve scaling for high point sizes, it is still not good enough to provide high-quality glyphs for high-resolution graphic devices. For high-quality text displays, TrueType fonts provide a solution.

Figure 14-13. Scaling a vector font glyph.



14.4 TRUETYPE FONTS

Up to Windows 3.0, the only available fonts in Windows were raster fonts and vector fonts. Raster fonts have jagged edges when scaled up, vector fonts have thin strokes, and neither of them provides good-quality text display at higher resolution, especially for printing. Adobe, armed with its deep technology buildup in the Postscript language and Postscript fonts, invaded the Windows world with an alien spaceship called Adobe Type Manager (ATM). ATM literally hacks into Windows GDI to bring a whole new set of continuously scalable fonts to all Windows applications. Magically, the jagged edges and thin strokes of Windows text display were replaced with smooth, professionally designed glyphs that look the same both on screens and printers. Microsoft was quick to realize the benefit of a better font technology for the Windows operating systems and brought Apple's TrueType font technology to Windows starting with Windows 3.1.

A TrueType font uses lines and curves to define the glyph outlines, which make it fully scalable to any large size without changing the shape of its glyphs. This is different from a vector font in two ways. First, a TrueType font uses curves that will still be smooth when scaled up, unlike a vector font whose intersection of line segments will be visible at a large size. Second, a vector font glyph only defines the paths of strokes, while a TrueType font defines the outline of its glyphs. Besides its much-improved glyph definition, a TrueType font also contains lots of other information that provides many advantages over previous Windows font technologies. Let's start with the basics of TrueType fonts.

TrueType Font File Format

A TrueType font is normally contained in a single TrueType font file, with .TTF file extension. Windows operating systems recently added support for OpenType fonts, which are PostScript fonts encoded in a format similar to TrueType fonts. OpenType fonts use the .OTF file extension. OpenType also allows combining multiple OpenType fonts in a single file to share data. Such fonts are referred to as a TrueType Collection, which uses .TTC file extension.

A TrueType font is encoded in Macintosh's outline font resource format, which has a unique tag name "sfnt." Windows does not use Macintosh's bitmap font resource format, which uses a different tag name "NFNT." A TrueType font starts with a small font directory, which is a guide to the dozens of tables following it. The font directory contains a version number of the font format, a number of tables, and one TableEntry structure for each table. A TableEntry structure holds the resource tag, checksum, offset, and size of each table. Here is a C definition of a TrueType font directory.

```
typedef struct
{
    char tag[4];
    ULONG checkSum;
    ULONG offset;
    ULONG length;
} TableEntry;
```

```
typedef struct
```

```
{
    Fixed     sfntversion; // 0x10000 for version 1.0
    USHORT    numTables;
    USHORT    searchRange;
    USHORT    entrySelector;
    USHORT    rangeShift;
    TableEntry entries[1]; // variable number of TableEntry
} TableDirectory;
```

As most Windows programmers may not be aware, TrueType fonts were originally defined by Apple, whose operating systems run on Motorola CPUs instead of Intel CPUs. All data in TrueType fonts use Big-Endian encoding, in which the most significant byte comes first. If a TrueType font starts with 00 01 00 00, 00 17, we know it's in outline font resource ("sfnt") version 1.0 format with 23 tables.

The last field of the TableDirectory structure is a variable-size array of Table Entry structures, one for each table in the font. Each table in a TrueType font stores a different logical piece of information—for example, glyph data, character-to-glyph map, kerning information, etc. Some tables are required, some are optional. [Table 14-3](#) lists commonly seen tables in a TrueType font.

Table 14-3. Common Tables in a TrueType Font

Tag	Name	Description
head	Font header	Global information about the font
cmap	Character-code-to-glyph mapping	Maps character codes to glyph indexes
glyf	Glyph data	Glyph outline definition and grid fitting instructions
maxp	Maximum profile	Font summary data for memory allocation
mmtx	Horizontal metrics	Glyph horizontal metrics
loca	Index to location table	Translates glyph index to glyph location
name	Naming table	Copyright notice, font name, family name, style name, etc.
hhea	Horizontal layout	Font horizontal layout: ascent, descent, line gap, max advance width, minimum left-side bearing, minimum right-side bearing, etc.
hmtx	Horizontal metrics	Advanced width, left-side bearing
kern	Kerning table	Array of kerning pairs
post	PostScript information	PostScript FontInfo directory entry and Postscript names of all the glyphs
PCLT	PCL 5 data	Font information for HP PCL 5 Printer Language: font number, pitch, xHeight, style, symbol set, etc.
OS/2	OS/2 and Windows specific metrics	Required metrics set for TrueType fonts.

Within the TableDirectory structure, all the TableEntry structures must be sorted according to their tag name. For example: "cmap" should appear before "head," which is before "glyf." But the actual table could be anywhere in the TrueType font file.

The Win32 API provides a function for applications to query the RAW TrueType font information:

```
DWORD GetFontData(HDC hDC, DWORD dwTable, DWORD dwOffset,
    LPVOID lpvBuffer, DWORD cbData);
```

You can use GetFontData to query for the TrueType font corresponding to the current logical font selected in a device context, so instead of passing a logical font handle, a device context handle is passed. You can query either for the whole TrueType file or for one table within the file. To query for the whole file, pass 0 as the dwTable parameter; otherwise, pass the DWORD form of the four-letter tag for the table. Parameter dwOffset is the starting offset of the table to query, or zero for the whole table, lpvBuffer is the buffer address, and cbData is the buffer size. If NULL and zero are passed as the last two parameters, GetFontData returns the size of the font file or table; otherwise data will be copied to the application-provided buffer.

The following routine queries for the whole TrueType font raw data:

```
TableDirectory * GetTrueTypeFont(HDC hDC, DWORD & nFontSize)
{
    // query font size
    nFontSize = GetFontData(hDC, 0, 0, NULL, 0);

    TableDirectory * pFont = (TableDirectory *) new BYTE[nFontSize];
    if ( pFont==NULL )
        return NULL;

    GetFontData(hDC, 0, 0, pFont, nFontSize);

    return pFont;
}
```

GetFontData is provided for applications to embed TrueType fonts in their documents to make sure they can be displayed on another machine which may not have the font. The idea is for the application to query for the font data, write it as part of the document, and install the font when the document is opened so that the document can be displayed in the same way as the original machine creating it. For example, the Windows NT/2000 spooler embeds TrueType fonts in spooler files when printing to a server, in order to make sure documents can be printed properly on another machine.

Once the raw data of a TrueType font has been retrieved, its header TableDirectory structure is very easy to parse. Only its version and number of tables need to be checked, and then individual tables can be examined. We will look at a few important and interesting tables.

Font Header (“head” Table)

The font header table contains global information about a TrueType font. Here is the head table structure.

```
typedef struct
{
```

```
Fixed Table;           // 0x00010000 for version 1.0.  
Fixed fontRevision;   // Set by font manufacturer.  
ULONG checkSumAdjustment;  
ULONG magicNumber;    // Set to 0x5F0F3CF5.  
USHORT flags;  
USHORT unitsPerEm;    // Valid range is from 16 to 16384  
longDT created;       // International date (8-byte field).  
longDT modified;      // International date (8-byte field).  
FWord xMin;           // For all glyph bounding boxes.  
FWord yMin;           // For all glyph bounding boxes.  
FWord xMax;           // For all glyph bounding boxes.  
FWord yMax;           // For all glyph bounding boxes.  
USHORT macStyle;  
USHORT lowestRecPPEM; // Smallest readable size in pixels.  
SHORT fontDirectionHint;  
SHORT indexToLocFormat; // 0 for short offsets, 1 for long.  
SHORT glyphDataFormat; // 0 for current format.  
} Table_head;
```

A font's history is recorded in three fields: a font version number, a font initial-creation time stamp, and a font last-modification time. Eight bytes are used to store a single time stamp using the number of seconds since 12:00 midnight Jan 1, 1904, so we never have to worry about Y2K problems, or even Y2M problems.

A font is designed on a reference grid called the em-square; glyphs in the fonts are represented using coordinates on the grid. So the size of the em-square determines how the font glyphs should be scaled and also reflects the quality of the font. Units per em-square and bounding box for all the glyphs are stored in the font header. The valid range for an em-square is 16 to 16,384, while the commonly seen values are 2048, 4096, and 8192. For example, for the Wingding font, units per em-square is 2048; the glyph bounding box is [0, -432, 2783, 1841].

Other information in the font header table includes smallest readable pixel size, font direction hint, glyph index to location table format, glyph data format, etc.

Maximum Profile (“maxp” Table)

A TrueType font is a very dynamic data structure, in that it contains a variable number of glyphs, each of them having a different number of control points, and even an unknown amount of glyph instructions. The purpose of the maximum profile table is to establish memory requirements for the font rasterizer so that the proper amount of memory can be allocated before handling the font. Because performance is so important to a font rasterizer, a dynamically growing data structure like MFC's CAarray that needs frequent data copying is not an option. Here is the map table structure.

```
typedef struct  
{  
    Fixed Version;           // 0x00010000 for version 1.0.  
    USHORT numGlyphs;        // Number of glyphs in the font.  
    USHORT maxPoints;        // Max points in noncomposite glyph.  
    USHORT maxContours;      // Max contours in noncomposite glyph.
```

```
USHORT maxCompositePoints; // Max points in a composite glyph.  
USHORT maxCompositeContours; // Max contours in a composite glyph.  
USHORT maxZones; // 1 if not use the twilight zone (Z0),  
                  // or 2 if so use Z0; 2 in most cases.  
USHORT maxTwilightPoints; // Maximum points used in Z0.  
USHORT maxStorage; // Number of storage area locations.  
USHORT maxFunctionDefs; // Number of FDEFs.  
USHORT maxInstructionDefs; // Number of IDEFs.  
USHORT maxStackElements; // Max stack depth.  
USHORT maxSizeOfInstructions; // Max byte count for glyph inst.  
USHORT maxComponentElements; // Max number top components referenced.  
USHORT maxComponentDepth; // Max levels of recursion.  
} Table_maxp;
```

The numGlyphs field stores the total number of glyphs in a font, which determines the size of the glyph index to location table and can be used to verify if a glyph index is valid. Each glyph in a TrueType font can be either a composite or a noncomposite glyph. A noncomposite glyph may have one or several contours, each defined by a number of control points. A composite glyph is defined as a composition of several other glyphs. The maxPoints, maxContours, maxCompositePoints, and maxCompositeContours fields capture the complexity of glyph definition.

Besides glyph definition, TrueType fonts use glyph instruction to hint to the font rasterizer how to adjust control points to make the rasterized glyph well balanced and nicer looking. Glyph instructions can also appear at the global font level in the font program table (“fpgm”) and control value program table (“prep”). TrueType glyph instructions are byte-instructions for a pseudo-machine, similar to Java’s virtual machine, which can be executed using a stack machine. The maxStackElements and maxSizeOfInstructions fields tell the stack machine the complexity of these instructions.

As an example, the Wingding font has 226 glyphs, 47 maximum contours for a glyph, 268 maximum points for a noncomposite glyph, 141 maximum points for a composite glyph, 14 maximum contours for a composite glyph, 492 levels of stack needed in the worst case, and longest instruction has 1119 bytes.

Character-to-Glyph Index Mapping (“cmap” Table)

The character-to-glyph index mapping table defines the mapping from character codes in different code pages to the glyph index, which is the key to accessing glyph information in a TrueType font. The cmap table can contain several subtables to support different platforms and different character encoding schemes.

Here is the structure of a “cmap” table.

```
typedef struct  
{  
    USHORT Platform; // platform ID  
    USHORT EncodingID; // encoding ID  
    ULONG TableOffset; // offset to encoding table  
} submap;
```

```
typedef struct  
{
```

```
USHORT TableVersion; // table version 0
USHORT NumSubTable; // number of encoding tables
submap TableHead[1]; // heads for encoding tables
} Table_cmap;
```

```
typedef struct
{
    USHORT format;      // format: 0, 2, 4, 6
    USHORT length;     // size
    USHORT version;    // version
    BYTE map[1];       // mapping data
} Table_Encode;
```

A “cmap” table (struct Table_cmap) starts with a version number, number of subtables, and table headers for all subtables. Each subtable (struct submap) has a platform ID, encoding ID, and an offset to the real subtable for the particular platform and encoding. Microsoft operating systems use platform ID 3, and the suggested encoding ID is 1 for UniCode. Other encoding IDs include 0 for symbol code page, 2 for Shift-JIS (Japanese Industrial Standard), 3 for Big5 (traditional Chinese), 4 for PRC (simplified Chinese), 5 for Wansung, and 6 for Johab (both for Korean).

The actual encoding table (Table_Encode) starts with format, length, and version fields, followed by the mapping data. Currently, there are four different formats of mapping table. Format 0 is simple byte-encoding table that can map 256 characters. Format 2 uses mixed 8/16-bit encoding for Japanese, Chinese, and Korean languages. Format 4 is the Microsoft standard segment mapping to delta values format. Format 6 is the trimmed table mapping.

A typical TrueType font used in Windows has two encoding tables, one a single-byte format 0 mapping table which maps ANSI characters to glyph indexes, another a format 4 mapping table for mapping UNICODE characters to glyph indexes.

Conceptually a mapping table is a simple data structure to map an integer to an integer, but the actual design of a format 4 mapping table is too complicated to describe in a few paragraphs.

When the character-to-glyph index mapping table is used to map a character code to glyph index, missing characters are mapped to glyph index 0, where a special glyph for a missing character is put.

The “cmap” table is normally hidden from an application unless you want to get its raw data using the GetFontData GDI function. Windows 2000 adds two new functions to give the application more friendly access to this information.

```
typedef struct
{
    WCHAR wcLow;
    USHORT cGlyphs;
}

typedef struct
{
    DWORD cbThis; // sizeof(GLYPHSET) + sizeof(WCRANGE) * (cRanges-1)
    DWORD flAccel;
    DWORD cGlyphsSupported;
```

```
DWORD cRanges;
WCRANGE ranges[1]; // ranges[cRanges]
} GLYPHSET;

DWORD GetFontUnicodeRanges(HDC hDC, LPGLYPHSET lpgs);
DWORD GetGlyphIndices(HDC hDC, LPCTSTR lptr, int c, LPWORD pgi,
    DWORD fl);
```

Normally a font only supplies glyphs for a subset of characters in the UNICODE character set. These characters can be grouped into ranges, as in a “cmap” mapping table. GetFontUnicodeRanges fills a GLYPHSET structure with the number of glyphs supported, number of UNICODE ranges supported, and the details about these ranges for the current font selected in a device context. The GLYPHSET structure is a variable-size structure, whose size depends on the number of UNICODE ranges supported. So, as for other Win32 API supporting variable-size structure, GetFontUnicodeRanges is normally called twice. The first call to it passes a NULL pointer as the last parameter; GDI returns the size of the space needed. Caller is supported to allocate the memory needed and call again to get the real data. In both cases, GetFontUnicodeRanges returns the size of data needed to store the full structure. MSDN document may still incorrectly state that GetFontUnicodeRanges returns a pointer to the GLYPHSET structure if the second parameter is NULL.

Here is a simple function for querying the GLYPHSET structure for the current font in a device context.

```
GLYPHSET *QueryUnicodeRanges(HDC hDC)
{
    // query for size
    DWORD size = GetFontUnicodeRanges(hDC, NULL);

    if (size==0) return NULL;
    GLYPHSET * pGlyphSet = (GLYPHSET *) new BYTE[size];

    // get real data
    pGlyphSet->cbThis = size;
    size = GetFontUnicodeRanges(hDC, pGlyphSet);

    return pGlyphSet;
}
```

If you try the GetFontUnicodeRanges function on some Windows TrueType fonts, you will find that it's common for these fonts to support over a thousand glyphs, which are grouped into hundreds of UNICODE ranges. For example, “Times New Roman” has 1143 glyphs in 145 ranges, the first range being 0x20..0x7F, the printable 7-bit ASCII code range.

GetFontUnicodeRanges only uses part of the information in a TrueType font “cmap” table, namely the domain of the mapping from UNICODE to glyph index. Get GlyphIndices actually uses the map to translate a character string to an array of glyph indexes. It accepts a device context handle, a string pointer, a string length, a pointer to a WORD array, and a flag. The WORD array is where the generated glyph indices will be stored. If the flag is GGI_MASK_NONEXISTING_GLYPHS, missing characters will be marked as 0xFFFF in glyph index. Glyph indexes generated by the function can be passed to other GDI functions like ExtTextOut.

Index to Location (“loca” Table)

The most significant information in a TrueType font is the glyph data in the “glyf” table. But given a glyph index, to find the corresponding glyph, you need the index to location table to translate glyph indexes to offsets within the glyph data table.

The index to location table stores $n + 1$ offsets into the glyph data table, where n is the number of glyphs stored in the maximum profile table. The extra offset at the end does not point to a new glyph; instead it points to the end of the last glyph. This design allows a TrueType font not to store the length of each glyph anywhere in the font. Instead, the font rasterizer can calculate glyph length using the difference of offsets between the next glyph and the current glyph.

Each index in the index to location table is stored as either an unsigned short or an unsigned long, depending on the `indexToLocFormat` field in the head table. Each glyph is required to be unsigned short aligned; when the short format index table is used, the index table actually stores WORD offset instead of BYTE offset. This allows the short form of the index to location table to support up to a 128-KB glyph data table.

Glyph Data (“glyf” Table)

The glyph data table is the core of a TrueType font, so usually it's the biggest table. With the glyph index to location being in a separate table, the glyph data table contains nothing else but a sequence of glyphs, each of them starting with a glyph header structure:

```
typedef struct
{
    WORD    numberOfContours; // contour number, negative if composite
    FWord   xMin;           // Minimum x for coordinate data.
    FWord   yMin;           // Minimum y for coordinate data.
    FWord   xMax;           // Maximum x for coordinate data.
    FWord   yMax;           // Maximum y for coordinate data.
} GlyphHeader;
```

For a simple noncomposite glyph, the `numberOfContours` field is the number of contours in the current glyph; for a composite glyph, the `numberOfContours` field is negative. In the latter case, the total number of contours needs to be calculated based on all the glyphs making up the composite glyph. The next four fields in the `GlyphHeader` structure store the bounding box of the glyph.

For a noncomposite glyph, the glyph description follows the `GlyphHeader` structure. A glyph description is made up of several pieces of information: endpoint indexes for all contours, glyph instructions, and a sequence of control points. Each control point has a flag, and an x, y coordinate. Conceptually the control points need the same information as is needed by the GDI PolyDraw function: an array of flags and an array of point coordinates. But in Truetype fonts, the control points are encoded using a sophisticated scheme. Here is the outline of a glyph description:

```
USHORT endPtsOfContours[n]; // n = number of contours
```

```
USHORT instructionlength;
BYTE instruction[i]; // i = instruction length
BYTE flags[]; // variable size
BYTE xCoordinates[]; // variable size
BYTE yCoordinates[]; // variable size
```

A glyph may contain one or several contours. For example, the letter “o” has two contours, one for the inside outline and another for the outside outline. For each contour, the endPtsOfContours array stores its endpoint index, from which the number of points in a contour can be calculated. For example, endPtsOfContours[0] is the number of points for the first contour; endPtsOfContours[1] - endPtsOfContours[0] is the number of points for the second contour.

Glyph instruction length and instruction array follow the endpoint array. But we will look at the control points first. The control points for a glyph are stored in three arrays: the flag array, x-coordinate array, and y-coordinate array. It's quite easy to find the starting point of the flag array, but there is no size field for the flag array and there is no direct reference to the other two arrays. You have to decode the flag array to locate and understand the x, y-coordinate arrays.

We mentioned that the em-square is limited to 16,384 units in size, so normally two bytes are needed for the x-coordinate, and two more bytes for the y-coordinate. To save space, which is why this form of encoding is used, relative coordinates are stored in a glyph description. For the first point, its coordinates are relative to (0,0); all the subsequent points store coordinate difference from the previous point. Some have relative difference small enough to be represented in a single byte, some may have zero difference, and some may have a difference that can't fit into a single byte. The flag array keeps per-coordinate encoding information, together with other information. Here is a summary of the meaning of each bit in a flag.

```
typedef enum
{
    G_ONCURVE = 0x01, // on curve, off curve
    G_REPEAT = 0x08, // next byte is flag repeat count

    G_XMASK = 0x12,
    G_XADDBYTE = 0x12, // X is positive byte
    G_XSUBBYTE = 0x02, // X is negative byte
    G_XSAME = 0x10, // X is same
    G_XADDINT = 0x00, // X is signed word

    G_YMASK = 0x24,
    G_YADDBYTE = 0x24, // Y is positive byte
    G_YSUBBYTE = 0x04, // Y is negative byte
    G_YSAME = 0x20, // Y is same
    G_YADDINT = 0x00, // Y is signed word
};
```

In [Chapter 8](#) on lines and curves, we mentioned that a segment of a cubic Bezier curve is defined by four control points: one on the curve defining the starting point, two off-curve control points, and another on-curve control point being the endpoint of the curve. A glyph outline in TrueType is defined using second-degree Bezier curves defined by one on-curve point, one off-curve point, and another on-curve point. Multiple continuous off-curve points are allowed, not to define cubic Bezier curves or other higher-order Bezier curves, but to reduce the number of control points. For example, for four points in the pattern on-off-off-on, an extra on-curve point is added implicitly to make it

on-off-on-off-on, thereby defining two second-degree Bezier curve segments.

If G_ONCURVE bit is on, a control point is on the curve; otherwise it's off the curve. If G_REPEAT bit is on, the next byte in the flag array is a repeat count, and the current flag should be repeated that many times. We have a kind of run-length encoding in the flag array. Other bits in a flag are divided to tell how the corresponding x, y-coordinates are encoded; they can mean whether the relative coordinate is the same as the last, a positive single-byte value, a negative single-byte value, or a signed two-byte value.

Decoding a glyph description is a two-pass process. First, walk through the flag array to find the end of it and the length of the x-coordinate array, from which we know the starting points for both x- and y-coordinate arrays. Second, go through each point in the glyph definition to convert it to a more manageable format. [Listing 14-2](#) shows a function to decode a TrueType glyph, which is a method of the KTrueType class.

Listing 14-2 KTrueType::DecodeGlyph: Decoding Noncomposite Glyph

```
int KTrueType::DecodeGlyph(int index, KCurve & curve, XFORM * xm) const
{
    const GlyphHeader * pHeader = GetGlyph(index);

    if ( pHeader==NULL )
        return 0;

    int nContour = (short) reverse(pHeader->numberOfContours);

    if ( nContour<0 ) // composite glyph
    {
        return DecodeCompositeGlyph(pHeader+1, curve); // after header
    }

    if ( nContour==0 )
        return 0;

    curve.SetBound(reverse((WORD)pHeader->xMin),
                  reverse((WORD)pHeader->yMin),
                  reverse((WORD)pHeader->xMax),
                  reverse((WORD)pHeader->yMax));

    const USHORT * pEndPoint = (const USHORT *) (pHeader+1);

    // total points: endpoint of last contour +1
    int nPoints = reverse(pEndPoint[nContour-1]) + 1;
    // instruction length
    int nInst  = reverse(pEndPoint[nContour]);

    // flag array: after instruction array
    const BYTE * pFlag = (const BYTE *) &pEndPoint[nContour] + 2 + nInst;
    const BYTE * pX   = pFlag;
    int xlen = 0;
```

```
// parse flag array once to find x-array location and size
for (int i=0; i<nPoints; i++, pX++)
{
    int unit = 0;

    switch ( pX[0] & G_XMASK )
    {
        case G_XADDBYTE:
        case G_XSUBBYTE:
            unit = 1;
            break;

        case G_XADDINT:
            unit = 2;
    }

    if ( pX[0] & G_REPEAT )
    {
        xlen += unit * (pX[1]+1);

        i += pX[1];
        pX++;
    }
    else
        xlen += unit;
}

const BYTE * pY = pX + xlen; // y-coordinate array is after x-array

int x = 0;
int y = 0;

i = 0;
BYTE flag = 0;
int rep = 0;
// walk three arrays at the same time
for (int j=0; j<nContour; j++) // one contour at a time
{
    int limit = reverse(pEndPoint[j]); // contour limit

    while ( i<=limit )
    {
        if ( rep==0 )
        {
            flag = * pFlag++;
            rep = 1;

            if ( flag & G_REPEAT )
```

```
    rep += * pFlag++;
}

int dx = 0, dy = 0;

switch ( flag & G_XMASK )
{
    case G_XADDBYTE: dx =  pX[0]; pX ++; break;
    case G_XSUBBYTE: dx = - pX[0]; pX ++; break;
    case G_XADDINT:  dx = (short )( (pX[0] << 8) + pX[1]);
                      pX+=2;
}

switch ( flag & G_YMASK )
{
    case G_YADDBYTE: dy =  pY[0]; pY ++; break;
    case G_YSUBBYTE: dy = - pY[0]; pY ++; break;
    case G_YADDINT:  dy = (short )( (pY[0] << 8) + pY[1]);
                      pY+=2;
}

x += dx;
y += dy;
assert(abs(x)<16384);
assert(abs(y)<16384);

if ( xm ) // apply transformation if given
    curve.Add((int) ( x * xm->eM11 + y * xm->eM21 + xm->eDx ),
               (int) ( x * xm->eM12 + y * xm->eM22 + xm->eDy ),
               (flag & G_ONCURVE) ? CCurve::FLAG_ON : 0);
else
    curve.Add(x, y, (flag & G_ONCURVE) ? CCurve::FLAG_ON : 0);

rep--;
i++;
}

curve.Close();
}

return curve.GetLength();
}
```

The KTrueType class handles the loading and decoding of a TrueType font; its complete source code is on the accompanying CD. DecodeGlyph handles decoding a single glyph, given a glyph index and a possible transformation matrix. The curve parameter, which is of class KCurve, is for gathering the TrueType glyph definition to a plain 32-bit point array and a simple flag array, which can then be easily displayed using GDI. A simple TrueType font editor can be built around it.

The code calls the GetGlyph method, which uses the index to location table, to find the GlyphHeader structure for that glyph. The number of contours for the glyph is retrieved. Note we have to reverse the byte order of the value because TrueType fonts use Big-Endian byte order. If it's negative, signaling a composite glyph, the DecodeCompositeGlyph method is called instead. The code then locates the endPtsOfContours array, finds the total number of points, and skips the instruction to find the starting point of the flag array.

Now it needs to find the starting point of the x-coordinate array and its length by walking through the flag array once. For each control point, the amount of space it takes in the x-coordinate array could be zero to two bytes, depending on whether its relative coordinate is zero, single byte, or double bytes.

From the x-coordinate array address and length, they-coordinate array address is known. The code then walks through all the contours one by one, decodes each control point within, converts relative coordinates to absolute coordinates, and then adds it to the curve object. A transformation is applied to each control point if given.

Recall that a TrueType font uses second-degree Bezier curves, with possible multiple off-curve points between two on-curve points. To simplify the curve-drawing algorithm, the KCurve::Add method adds an extra on-curve point between every two off-curve points.

```
void KCurve::Add(int x, int y, BYTE flag)
{
    // off off -> off on off
    if ( m_len && ( (flag & FLAG_ON)==0 ) &&
        ( (m_Flag[m_len-1] & FLAG_ON)==0 ) )
    {
        Append((m_Point[m_len-1].x+x)/2,
               (m_Point[m_len-1].y+y)/2,
               FLAG_ON | FLAG_EXTRA); // add a middle point
    }

    Append(x, y, flag);
}
```

Now that noncomposite glyph is taken care of, let's look at the composite glyph. A composite glyph is defined by a sequence of transformed glyphs. Each transformed glyph definition has three parts: a flag, a glyph index, and a transformation matrix. The flag field determines how the transformation matrix is encoded, again to save a few bytes, and whether the end of the sequence has been reached. A full 2D affine transformation needs six values. But if it's only a translation, only two values (dx , dy) are needed, which can be stored in either two bytes or two words. If x and y are scaled by the same value, one extra scale value is needed. The most general case still needs six values, but most of the time several bytes can be saved. The values for the transformation are stored in 2.14 signed fixed-point notation, except for dx and dy , which are stored as integers. Making a composite glyph is actually combining several glyphs together, each with a transformation matrix. For example, if a glyph in a font is the exact mirror image of another glyph, it can just be defined as a composite glyph that is generated after applying a reflection transformation of another image. [Listing 14-3](#) shows the code for decoding a composite glyph.

Listing 14-3 KTrueType::DecodeCompositeGlyph

```
int KTrueType::DecodeCompositeGlyph(const void * pGlyph,
```

```
CCurve & curve) const
{
    KDataStream str(pGlyph);

    unsigned flags;

    int len = 0;
    do
    {

        flags    = str.GetWord();

        unsigned glyphIndex = str.GetWord();

        signed short argument1;
        signed short argument2;

        if ( flags & ARG_1_AND_2_ARE_WORDS )
        {
            argument1 = str.GetWord(); // (SHORT or FWord) argument1;
            argument2 = str.GetWord(); // (SHORT or FWord) argument2;
        }
        else
        {
            argument1 = (signed char) str.GetByte();
            argument2 = (signed char) str.GetByte();
        }

        signed short xscale, yscale, scale01, scale10;

        xscale = 1;
        yscale = 1;
        scale01 = 0;
        scale10 = 0;

        if ( flags & WE_HAVE_A_SCALE )
        {
            xscale = str.GetWord();
            yscale = xscale;      // Format 2.14
        }
        else if ( flags & WE_HAVE_AN_X_AND_Y_SCALE )
        {
            xscale = str.GetWord();
            yscale = str.GetWord();
        }
        else if ( flags & WE_HAVE_A_TWO_BY_TWO )
        {
            xscale = str.GetWord();
            scale01 = str.GetWord();
```

```
scale10 = str.GetWord();
yscale = str.GetWord();
}
if ( flags & ARGs_ARE_XY_VALUES )
{
    XFORM xm;

    xm.eDx = (float) argument1;
    xm.eDy = (float) argument2;
    xm.eM11 = xscale / (float) 16384.0;
    xm.eM12 = scale01 / (float) 16384.0;
    xm.eM21 = scale10 / (float) 16384.0;
    xm.eM22 = yscale / (float) 16384.0;

    len += DecodeGlyph(glyphIndex, curve, & xm);
}
else
    assert(false);
}

while ( flags & MORE_COMPONENTS );

if ( flags & WE_HAVE_INSTRUCTIONS ) // skip instructions
{
    unsigned numInstr = str.GetWord();

    for (unsigned I=0; I<numInstr; I++)
        str.GetByte();
}

return len;
}
```

The DecodeCompositeGlyph method decodes the flag, glyph index, and transformation matrix for each glyph included, and calls the DecodeGlyph method to decode it. Note: the call to DecodeGlyph has a valid transformation-matrix parameter that needs to be applied. The method ends when the MORE_COMPONENTS flag bit is off. The complete code can be found on the CD.

The decoded TrueType font glyphs are ready to be drawn using GDI, except for a small problem. GDI only draws cubic Bezier curves, so the second-degree Bezier curve control points decoded from the glyph table need to be converted to cubic Bezier control points. After some struggling with the original mathematical definition of Bezier curves, here is a simple routine that draws a second-degree Bezier curve using GDI.

```
// draw a 2nd-degree Bezier curve segment
BOOL Bezier2(HDC hDC, int & x0, int & y0, int x1, int y1, int x2, int y2)
{
    // p0 p1 p2 -> p0 (p0+2p1)/3 (2p1+p2)/3, p2

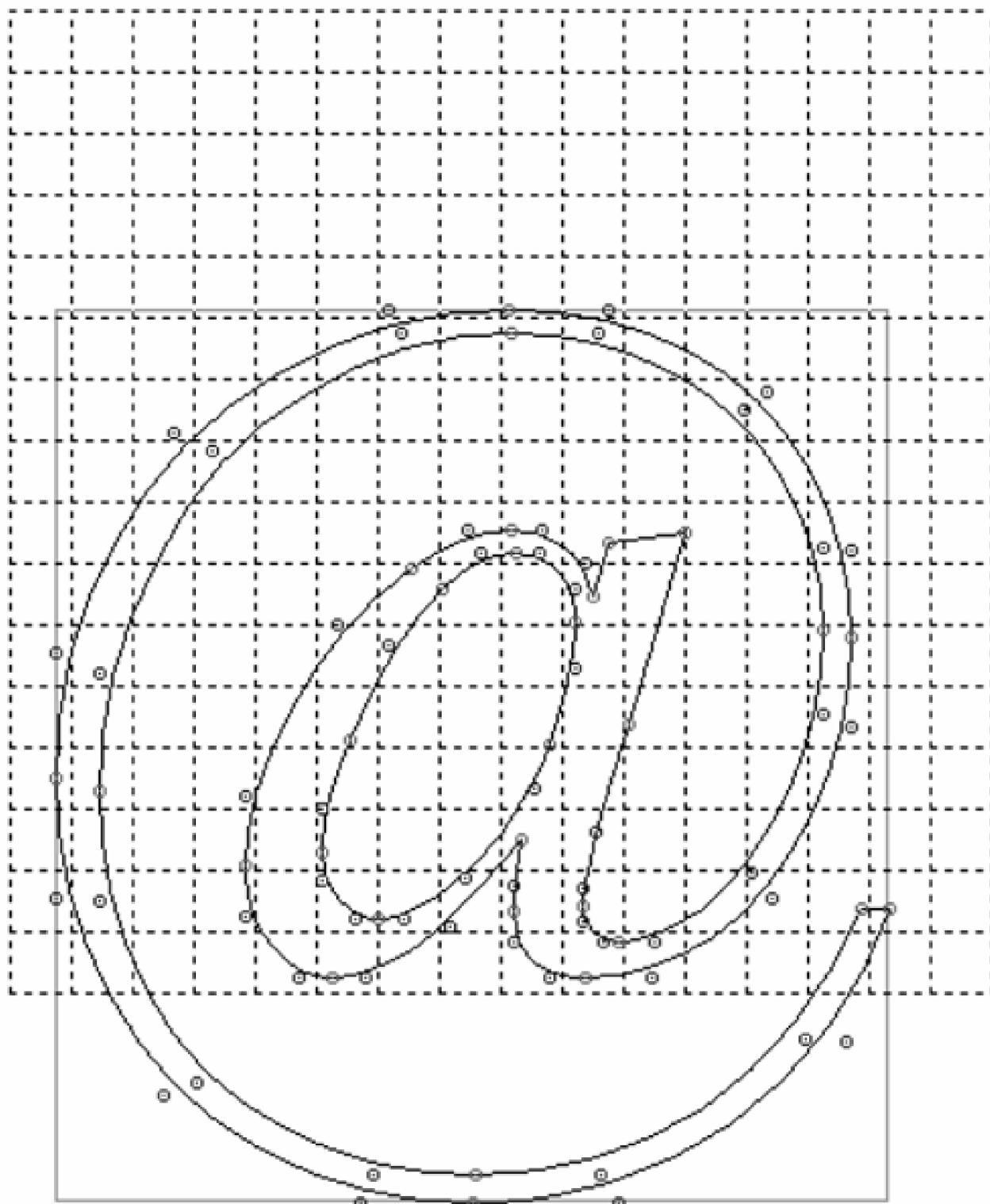
    POINT P[3] = { { (x0+2*x1)/3, (y0+2*y1)/3 },
```

```
{ (2*x1+x2)/3, (2*y1+y2)/3 },  
{ x2, y2 } };  
x0 = x2; y0 = y2;  
return PolyBezierTo(hDC, P, 3);  
}
```

For a second-degree Bezier curve defined by three control points (p_0, p_1, p_2) , the corresponding control points for a cubic Bezier curve are $(p_0, (p_0 + 2*p_1)/3, (2*p_1 + p_2)/3, p_2)$.

[Figure 14-14](#) shows the result of the actual drawing code implemented in the KCurve class. The dotted grid on the background is the em-square grid, divided into 16 portions along both axes. The solid rectangle is the bounding box for the glyph, in this case character “@”. The control points are marked with small circles. You can find alternating on-curve points and off-curve points. Most importantly, the curves represent the actual outline defined by the complicated glyph description.

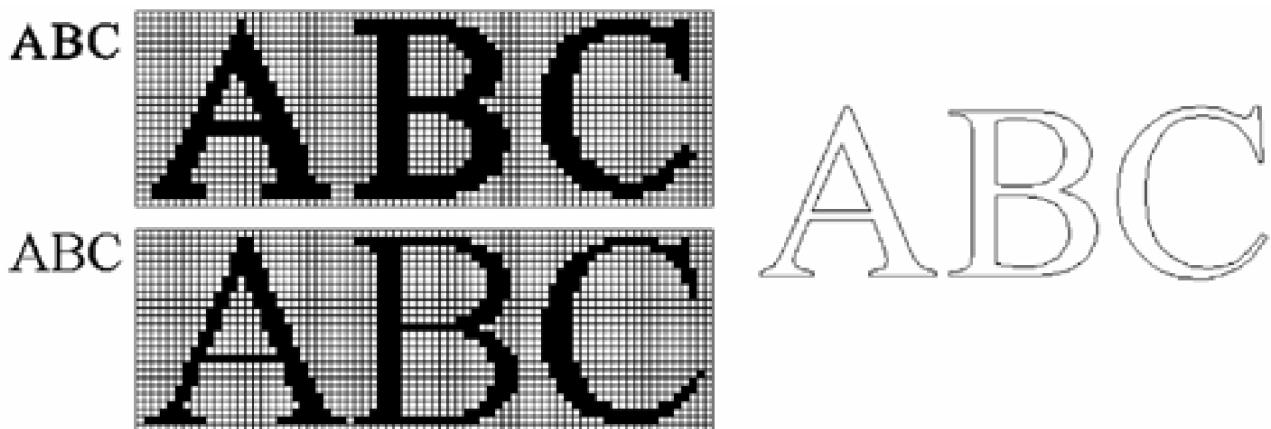
Figure 14-14. TrueType glyph description.



Glyph Instructions

[Listings 14-2](#) and [14-3](#) may give the impression that a TrueType font rasterizer can be easily implemented by scan converting glyph outlines—for example, using GDI's `Stroke AndFillPath` function to fill a path generated by drawing the glyph outlines. Such a simple-minded font rasterizer is not practically usable, unless it's used only on a high-resolution device like a printer. [Figure 14-15](#) illustrates this point.

Figure 14-15. Glyph rasterization.



[Figure 14-15](#) compares two implementations of TrueType glyph rasterization: a simple-minded rasterizer as shown in [Listings 14-2](#) and [14-3](#), and the actual TrueType font engine on Microsoft operating systems. The top-level part of the picture shows the result from the simple-minded rasterizer, while the bottom right part is what's implemented in the OS. The rasterization results are shown in both the original format and the zoomed-in version. The far right side of the picture shows the TrueType glyph outlines that both implementations try to approximate.

As can be seen from [Figure 14-15](#), the simple-minded rasterizer generates images with uneven stem weights, dropouts, loss of character features, loss of symmetry, etc. [Figure 14-15](#) shows the result of displaying 32-point characters on screen. When point sizes gets smaller, the situation can be even worse. Generally speaking, a simple-minded font rasterizer generates illegible images at small sizes and unpleasant results at larger sizes, and the results improve only as the point size increases.

When a glyph outline defined on a large em-square (typically 2048 units) is scaled to a much smaller grid (32 by 32, for example), inevitably precision is lost and error is introduced. For example, if two vertical lines are defined in the em-square units, the first line's bounding box is [14, 0, 25, 200], the second line's bounding box is [31, 0, 42, 200], and both lines have the same dimension, 11 by 200. Everything looks perfect until it's scaled down by, say, a factor of 10 with rounding; now the first line becomes [1, 0, 3, 20] and the second line [3, 0, 4, 20]. Note now that the first line's dimension is 2 by 20, while the second line's is 1 by 20, an uneven-stem-weight problem. You can find this problem in [Figure 14-15](#) in the bottom stroke of the letter 'B'; it's thicker than the top and middle strokes.

TrueType's remedy to this rasterization problem is to control the scaling of the glyph outline from em-square to rasterization grid so that the result can have a more pleasing look and stay closer to the original glyph design. The technique, which is called *grid fitting*, tries to achieve three goals:

- Eliminate the effect of chance relationships to the grid so as to generate uniform stem weights independent of the relative location on the grid.
- Control the key dimensions in and across glyphs.
- Preserve symmetries and other important glyph detail designs such as serifs.

The requirement for grid fitting is encoded in two places in a TrueType font: in a control value table and in per-glyph grid-fitting instructions.

A control value table (“cvt” table) is used to store an array of values that can be referenced by grid-fitting instructions. For example, for a font with serifs, the cap height, baseline, serif height, serif width, upper-case stem width, left-side bearing, and upper-case stroke may be among the values that need to be controlled. They can be put into the control value table in an order known to the font designer, and later referenced using their indexes. During font rasterization, values in the control value table are scaled according to the current point size. When the scaled values are referenced in the grid-fitting instructions, they ensure that the same values are applied independent of relative position to a grid. For example, if the horizontal line [14, 0, 25, 200] is specified as [14, 0, 14+CVT[stem_width], 0+CVT[cap_height]] using two values in the CVT table, the line's width and height are always the same regardless of its position on the grid.

For each glyph definition, a sequence of instructions, called *glyph instructions*, is attached to control the grid fitting for that glyph. Glyph instructions reference values in the control value table to make sure they are enforced by all the glyphs.

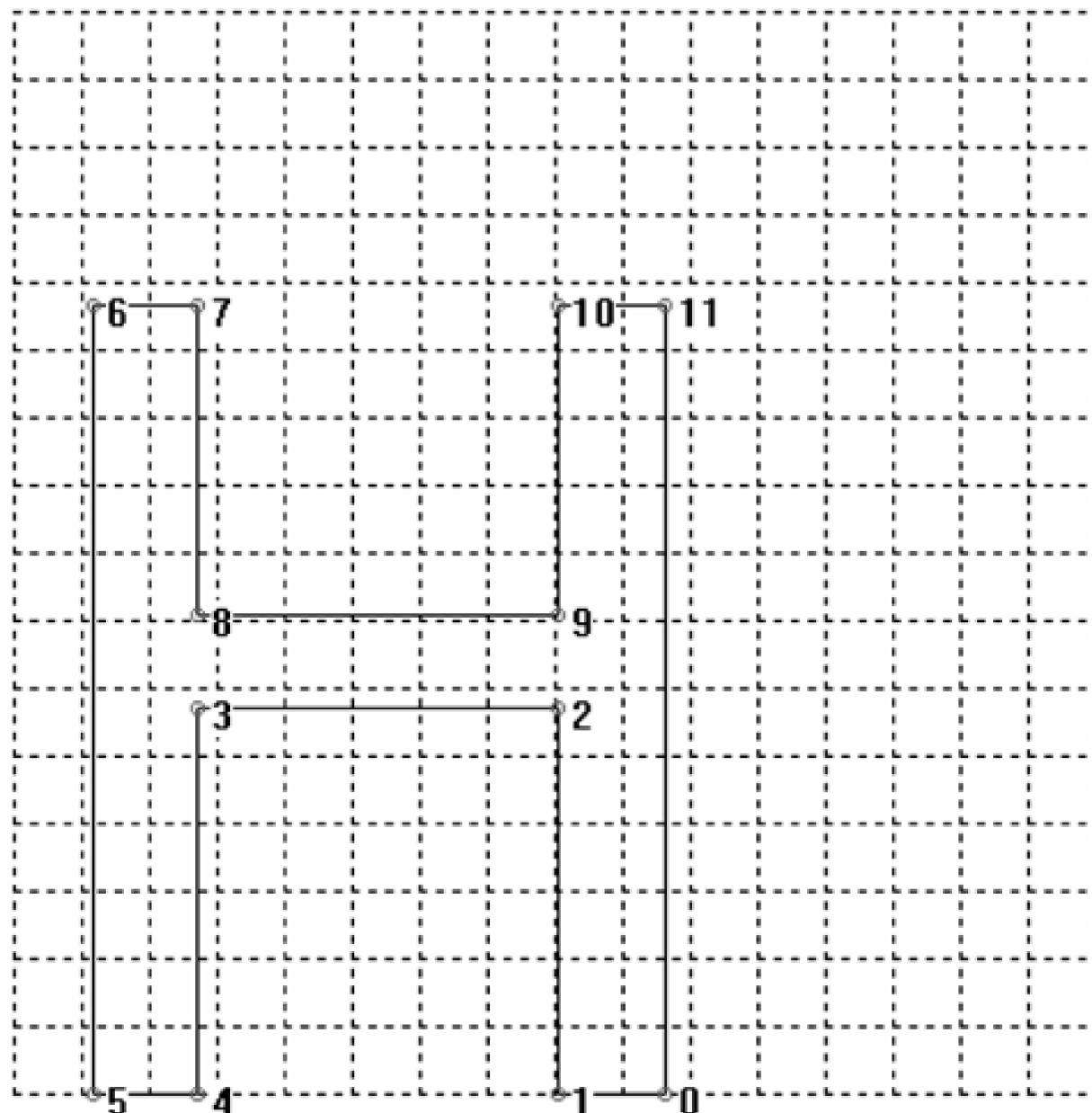
The glyph instructions are instructions for a stack-based pseudo-machine. A stack machine is widely used in interpretative implementation of computer languages, for the simplicity of its implementation. For example, Forth (a powerful simple language for embedded systems), RPL (a language used by HP calculators), and Java virtual machine are all stack machines.

A stack machine normally does not have any registers, as all computation happens on a stack (some stack machines use a separate control and data stack). For example, a push instruction pushes a value on the stack, a pop instruction removes the topmost value on the stack, a binary add instruction removes the two topmost values on the stack and pushes their sum on the stack.

The TrueType virtual machine is not a general-purpose stack machine; it's a special-purpose pseudo-machine designed for the sole purpose of grid-fitting glyph outlines. Besides referencing the control value table, it uses several *graphics state variables* such as reference point 0, reference point 1, projection vector, etc.

We will not try to explain the whole TrueType glyph instruction set; instead we will just explain its basic ideas through a simple example, letter “H” in the Tahoma font. [Figure 14-16](#) shows the details of its glyph outline.

Figure 14-16. Control points for letter “H” in Tahoma.



Letter "H" in Tahoma has a single contour with 12 control points, all on the curve, so there is no Bezier curve in this glyph. Besides the control points, it has 50 bytes of glyph instructions, which occupy more space than the coordinates. Here is the list of coordinates and glyph instructions.

Coordinates

0: 1232, 0
1: 1034, 0
2: 1034, 729
3: 349, 729
4: 349, 0
5: 151, 0
6: 151, 1489

7: 349, 1489
8: 349, 905
9: 1034, 905
10: 1034, 1489
11: 1232, 1489

Length of Instructions: 50

00: NPUSHB (28): 3 53 8 8 5 10 7 3
1 5 8 9 2 20 0 101
13 15 13 64 13 2 8 3
20 5 100 12
30: SRP0
31: MIRP[srp0,nmd,rd,2]
32: MIRP[srp0,md,rd,1]
33: SHP[rp2,zp1]
34: DELTAP1
35: SRP0
36: MIRP[srp0,nmd,rd,2]
37: MIRP[srp0,md,rd,1]
38: SHP[rp2,zp1]
39: SVTCA[y-axis]
40: MIAP[rd+ci]
41: ALIGNRP
42: MIAP[rd+ci]
43: ALIGNRP
44: SRP2
45: IP
46: MDAP[rd]
47: MIRP[nrp0,md,rd,1]
48: IUP[y]
49: IUP[x]

The 50 bytes of glyph instructions are divided into 21 instructions, most of them single-byte instructions except for the first. Each instruction has a mnemonic name, a set of flags enclosed in brackets, and some optional parameters. Let's look at the instructions one by one.

1. NPUSHB (push N bytes) instruction pushes a number of bytes on the stack. In this case, 28 bytes are taken from the instruction stream and put on the stack. The topmost element of the stack is 12.
2. SRP0 (set reference point 0) pops 12 from the stack and sets control point 12 as reference point 0. Control point 12 is the origin of the em-square.
3. MIRP[srp0, nmd, rd, 2] (move indirect relative point) pops 100 and 5 from the stack and moves point 5 so that its distance from reference point 0 is equal to CVT[100] (control value table value at index 100). This instruction anchors the leftmost point on the glyph to be a known distance from the origin in the x-direction. Extra flag srp0 sets reference point 0 to point 5, nmd does not keep distance greater than or equal to the minimum distance, rd rounds the distance and looks at the control value cut-in value, and 2 is a distance type for engine characteristic compensation.

4. MIRP[srp0,md,rd,1] (move indirect relative point) pops 20 and 3 from the stack and moves point 3 relative to point 5 according to CVT[20]. This ensures a fixed width for horizontal stem.
5. SHP[rp2, zp1] (shift point using reference point) pops 8 from the stack and shifts point 8 by the same amount that the reference point (point 3) has been shifted.
6. DELTAP1 (DELTA exception P1) pops 2, 13, 64, 13, and 15 from the stack and creates exceptions at point 64 with value 13, and at point 15 with value 13. It moves the specified points at the size and by the amount specified in the paired value (13 in this case). In this case, the point values seem to be invalid.
7. SRP0 (set reference point 0) pops 13 from the stack and sets point 13 as reference point 0. Point 13 is an automatically added point, whose distance from the em-square origin (point 12) is the advance width of the glyph.
8. MIRP[srp0, nmd, rd, 2] (move indirect relative point) pops 101 and 0 from the stack and moves point 0 relative to point 13 according to CVT[101]. It also sets reference point 0 to point 0.
9. MIRP[srp0, md, rd, 1] (move indirect relative point) pops 20 and 2 from the stack and moves point 2 relative to point 0 according to CVT[20].
10. SHP[rp2, zp1] (shift point using reference point) pops 9 from the stack and shifts point 8 by the same amount that the reference point has (point 2) been shifted.
11. SVTCA[y-axis] sets freedom and projection vectors to the y-axis. Now we've finished x-axis grid fitting and are moving to y-axis grid fitting.
12. MIAP[rd+ci] pops 8 and 5 from the stack and moves point 5 to the absolute coordinate position in CVT[8], which is 0.
13. ALIGNRP (align to reference point) pops 1 from the stack and aligns point 1 with reference point 0 (point 5).
14. MAIP[rd+ci] pops 3 and 7 from the stack and moves point 7 to the absolute coordinate position in CVT[3], which is 1489. This ensures the height of letter H to follow a uniquely defined value.
15. ALIGNRP (align to reference point) pops 10 from the stack and aligns point 10 with reference point 0 (point 7).
16. SRP2 (set reference point 2) pops 5 from the stack and sets point 5 as reference point 2.
17. IP (interpolate point) pops 8 from the stack and interpolates the position of point 8 to preserve the original relationship with reference points (point 5 and point 10).
18. MDAP[rd] (move direct absolute point) pops 8 from the stack, sets reference points 0 and 1 to point 8, and rounds point 8.
19. MIRP[nrop0, md, r1, 1] (move indirect relative point) pops 53 and 3 from the stack and moves point 3 relative to point 80 according to CVT[53].
20. IUP[y] interpolates untouched points through the outline in the y-direction.

21. UP[x] interpolates untouched points through the outline in the y-direction.

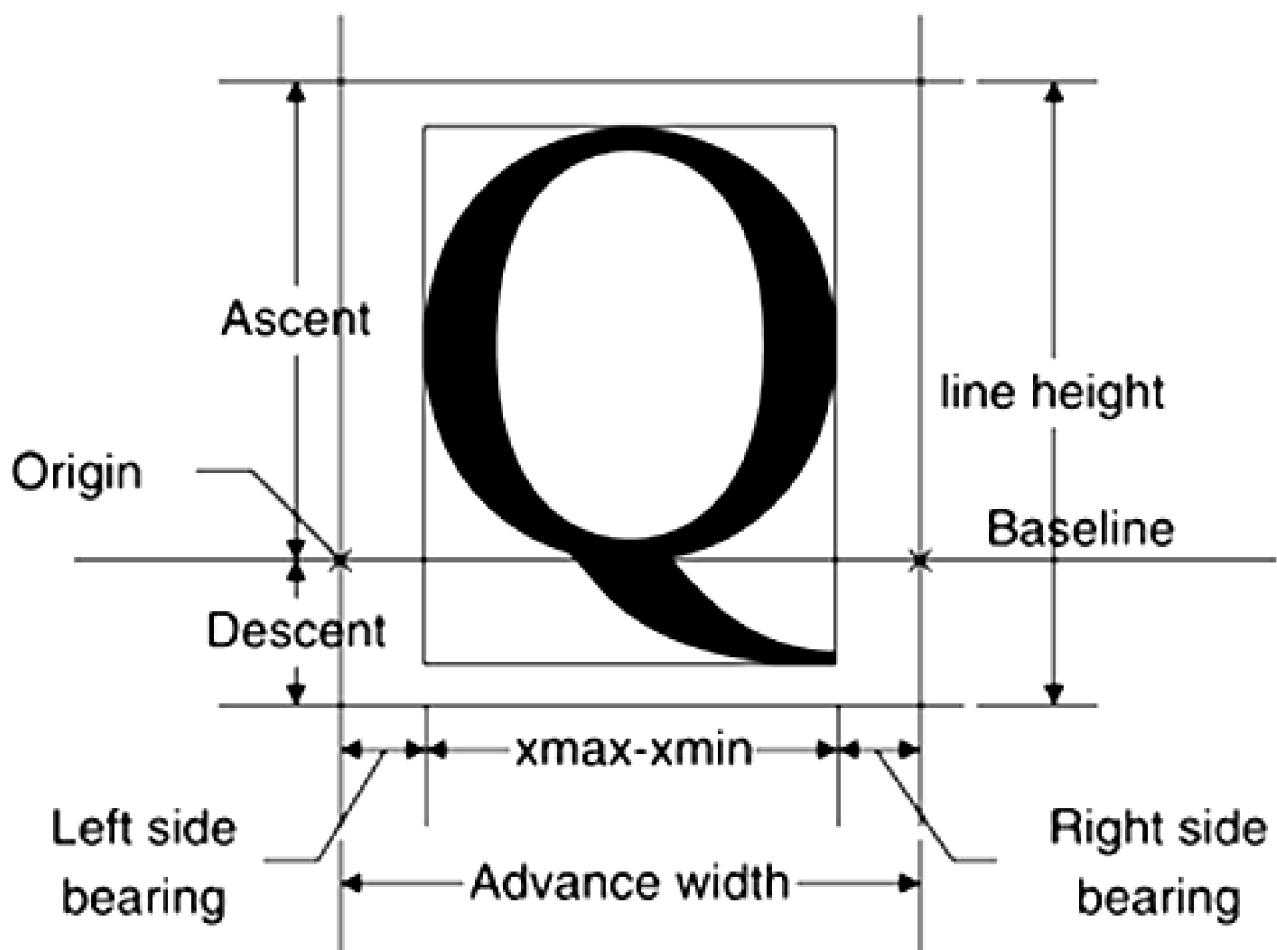
This is just an oversimplified description of a simple glyph's glyph instructions. The whole TrueType glyph instruction set and its semantics are much more complicated than what's shown here. There are over 100 different instructions, 20 graphics state variables, and several different data types. For complete information, refer to the *TrueType Reference Manual*, available from fonts.apple.com.

Horizontal Metrics (“hhea” and “htmx” Tables)

The glyph data table does not provide enough information to align glyphs horizontally together to form a line of text, or align lines of text vertically together to form a paragraph. Basic font metrics information for Roman fonts is encoded in two tables in a TrueType font: horizontal header table and horizontal metrics table.

Before we discuss these tables, let's look at a few glyph measurement terms, as illustrated in [Figure 14-17](#).

Figure 14-17. Glyph metrics terminology.



When a glyph is laid out in a line of text, it starts at a reference point, called the *origin*, which is marked by the first cross in [Figure 14-17](#). There is an imaginary horizontal line, called the *baseline*, which is the basis of aligning the glyph vertically. In a TrueType font, a glyph outline is defined on the em-square; the x-axis of the em-square is the baseline in aligning glyphs vertically.

For a font, *ascent* is the distance from top of the upper-case letters to the baseline. Ascent is an attribute of a font, instead of individual glyphs. It's used to determine the position of the baseline of a line of text from a starting position horizontally. Likewise, *descent* is a font attribute that is the distance from the baseline to the descenders (for glyphs like Q, q, or g). The sum of ascent and descent is the *line height* of a font, although when forming paragraphs, an extra *line gap* may be added.

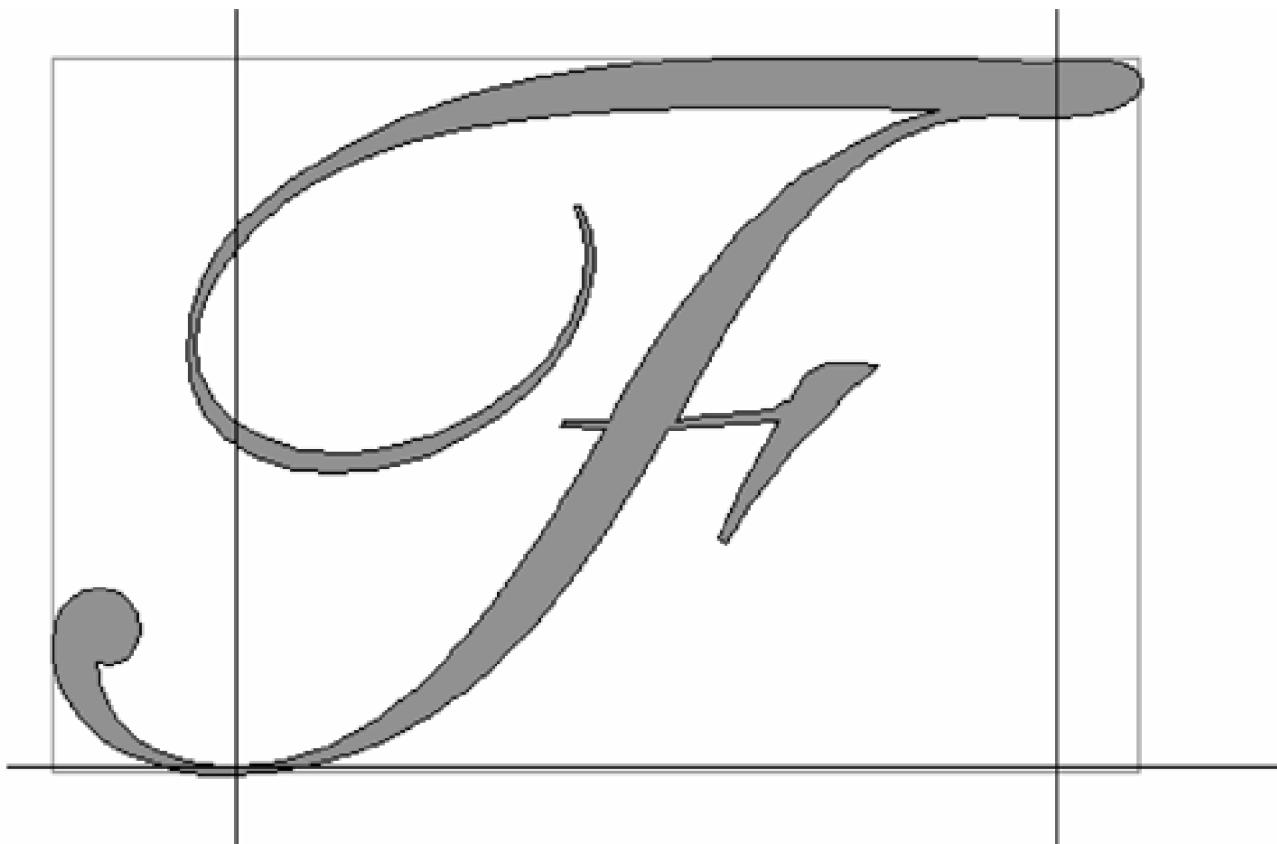
Each glyph has a bounding box, which is part of the glyph header in a TrueType font. A bonding box can be described as [xmin, ymin, xmax, ymax] which are the minimum and maximum vertical and horizontal coordinates of a glyph's control points.

Horizontally, there is usually some space between the origin and the xmin of a glyph, which is called *left-side bearing*. After a glyph is put on a line, the new origin of the next glyph has a distance away from its xmax position, which is called *right-side bearing*. Both left-side and right-side bearing are per-glyph attributes, just like a glyph's bounding box.

The sum of left-side bearing, width of the glyph ($xmax - xmin$), and right-side bearing is called *advance width*, which is the amount of horizontal movement for the origin after a glyph is put on the line. The next glyph starts from the new origin.

Left-side and right-side bearing are normally positive to add space between glyphs. But sometimes they can be negative to reduce space between glyphs. For example, in Times New Roman, lower-case "j" has negative left-side bearing, while lower-case "f" has negative right-side bearing. [Figure 14-18](#) shows the letter "F" in an italic font that has negative bearings on both sides.

Figure 14-18. Negative left-side and right-side bearings.



In a TrueType font, attributes like font ascent and descent are kept in a horizontal header table, while per-glyph information like left-side bearing and advance width are stored in a horizontal metrics table.

Here is the structure of a horizontal header table ('hhea' table).

```
typedef struct
{
    Fixed version;          // 0x00010000 for version 1.0.
    FWord Ascender;        // Typographic ascent.
    FWord Descender;       // Typographic descent.
    FWord LineGap;         // Typographic line gap.
    FWord advanceWidthMax; // Maximum advance width.
    FWord minLeftSideBearing; // Minimum left-side bearing.
    FWord minRightSideBearing; // Minimum right-side bearing.
    FWord xMaxExtent;      // Max(lsb + (xMax - xMin)).
    SHORT caretSlopeRise;  // for slope of the cursor.
    SHORT caretSlopeRun;   // 0 for vertical.
    SHORT reserved[5];     // set to 0.
    SHORT metricDataFormat; // 0 for current format.
    USHORT numberOfHMetrics; // hMetric entries in 'hmtx' table.
} Table_HoriHeader;
```

A horizontal header table ("hhea" table) stores a font's ascent, descent, extra line gap, maximum advance width, minimum left-side bearing, minimum right-side bearing, maximum extent (left-side extent + xmax - xmin), hints for caret display slope, and information about the horizontal metrics table.

A horizontal metrics table ("hmtx" table) stores per-glyph horizontal metrics information. For each glyph, there needs to be a way to get left-side bearing and advance width, from which the right-side bearing can be calculated using "advance width - left-side bearing - (xmax - xmin)." But for a monospace font, which has a constant advance width, storing multiple copies of the same advance width is considered a waste of space. So the horizontal metrics table is divided into two parts; the first part stores advance width and left-side bearing for each glyph, and the second part stores left-side bearing only while reusing the last advance width. The table should cover all the glyphs in the font; the number of glyphs having complete horizontal metrics is kept in the last field of the horizontal header table—that is, its numberOfHMetrics field. Here is the structure of the horizontal metrics table. Note that both parts are variable-size arrays.

```
typedef struct
{
    FWord advanceWidth;
    FWord lsb;
} longHorMetric;

typedef struct
{
    longHorMetric hMetrics[1];      // numberOfHMetrics;
    FWord      leftSideBearing[1]; // advanceWidth same as last
```

```
} Table_HoriMetrics;
```

Horizontal metrics information of a font is accessible through GDI using GetCharABCWidths, GetCharABCWidthsFloat, and GetCharABCWidthsI. In GDI terminology, A width is the left-side bearing, B width is xmax - xmin, and C width is the right-side bearing. We will cover these functions in the next chapter while talking about text formatting, because these functions are more tied together with GDI logical fonts than with physical TrueType fonts.

Kerning (“kern” Table)

Left-side and right-side bearing are used to fit glyphs together on a line to form a nice text string. But both of them are fixed values for a certain glyph. When two specific glyphs meet each other, their special shapes may need some adjustment to make them fit more nicely to each other. The process of adjusting text layout according to individual glyphs is called *kerning*. In other words, kerning is the process of adjusting text layout according to the context glyphs are in to make them appear to fit naturally together.

In a TrueType font, kerning is based on kerning pairs stored in a kerning table which the font designer puts into a font. Here are the structures of the TrueType kerning table.

```
typedef struct
{
    FWord  leftglyph;
    FWord  rightglyph;
    FWord  move;
} KerningPair;

typedef struct
{
    FWord      Version;
    FWord      nSubTables;
    FWord      SubTableVersion;
    FWord      Bytesinsubtable;
    FWord      Coveragebits;
    FWord      Numberpairs;
    FWord      SearchRange;
    FWord      EntrySelector;
    FWord      RangeShift;
    KerningPair KerningPair[1]; // variable size
} Table_Kerning;
```

A kerning table has a quite simple structure—a simple header and a plain array of KerningPair structures; each has two glyph indexes and an adjustment amount. Each kerning pair instructs text render to adjust distances between two specific glyphs when they are put next to each other in that order. For example, the first kerning pair of the Tahoma font contains 4, 180, -94, which means that if glyph 180 is put immediately after glyph 4, move the origin to the left by 94 em-square units to make them tighter. For a font with n glyphs, the maximum number of kerning pairs would be " $n \times n$," which will be huge for fonts with thousands of glyphs. Luckily, font designers put in kerning pairs only for a limited number of glyph pairs. For example, Tahoma has only 674 kerning pairs.

Kerning information of the font is accessible from an application using GDI function GetKerningPairs.

```
typedef struct
{
    WORD wFirst;
    WORD wSecond;
    int iKernAmount;
} KERNINGPAIR;
```

```
DWORD GetKerningPairs(HDC hDC, DWORD nNumPairs, LPKERNINGPAIR lpkrnpair);
```

To query the kerning pairs of the current logical font selected in a device context, call GetKerningPairs first with 0 as nNumPairs and NULL as lpkrnpair to query for the number of kerning pairs available, then allocate memory and call it again to retrieve the actual kerning-pair array. Note that the iKernAmount value in the KERNINGPAIR structure is in the device context's logical coordinate space, not the TrueType em-square units. Of course, you can get the raw kerning table using the Get FontData function.

OS/2 and Window Metrics (“OS/2” Table)

The OS/2 table is an important table, holding metrics information needed both for IBM's OS/2 family of operating systems and Microsoft's Windows family of operating systems. Perhaps its name suggests that it's used by OS/2 first. The graphics system needs a way to characterize different fonts installed on the system, so that when an application requests a font, a proper match can be found between the user's requirement and what's installed on the system. The OS/2 table provides lots of information for the graphics system to match user font requests.

The OS/2 table has the following structure.

```
typedef struct
{
    USHORT version;          // 0x0001
    SHORT xAvgCharWidth;    // weighted average width of a..z
    USHORT usWeightClass;   // FW_THIN .. FW_BLACK
    USHORT usWidthClass;    // ULTRA_CONDENSED .. ULTRA_EXPANDED
    SHORT fsType;           // embedding licensing rights
    SHORT ySubscriptXSize;
    SHORT ySubscriptYSize;
    SHORT ySubscriptXOffset;
    SHORT ySubscriptYOffset;
    SHORT ySuperscriptXSize;
    SHORT ySuperscriptYSize;
    SHORT ySuperscriptXOffset;
    SHORT ySuperscriptYOffset;
    SHORT yStrikeoutSize;   // strikeout stroke width in design units.
    SHORT yStrikeoutPosition;
    SHORT sFamilyClass;     // IBM font class
```

```
PANOSE panose;
ULONG ulUnicodeRange1; // Bits 0-31 Unicode Character Range
ULONG ulUnicodeRange2; // Bits 32-63
ULONG ulUnicodeRange3; // Bits 64-95
ULONG ulUnicodeRange4; // Bits 96-127
CHAR achVendID[4]; // vendor ID
USHORT fsSelection; // ITALIC .. REGULAR
USHORT usFirstCharIndex; // first UNICODE char
USHORT usLastCharIndex; // last UNICODE char
USHORT sTypoAscender; // typographic ascender
USHORT sTypoDescender; // typographic descender
USHORT sTypoLineGap; // typographic line gap
USHORT usWinAscent; // ascender metric for Windows
USHORT usWinDescent; // descender metric for Windows
ULONG ulCodePageRange1; // Bits 0-31
ULONG ulCodePageRange2; // Bits 32-63
} Table_OS2;
```

The OS/2 table has detailed information in a format quite closely matching GDI's font metrics structures—for example, LOGFONT, TEXTMETRICS, ENUM TEXTMETRIC, and OUTLINETEXTMETRICS. Because of the multiplatform nature of TrueType fonts, sometimes too much inconsistent information can be confusing. For example, the OS/2 table has two sets of ascenders and descenders which are not necessarily the same as similar attributes stored in the horizontal header table.

Other Tables

The most important tables in a TrueType font have been covered in detail. But a TrueType/OpenType font may have other tables for more advanced features, for use by other platforms or even printers.

The name table ("name") allows multilingual strings to be attached to a TrueType font. These strings can be font names, family names, style names, copyright notices, and so on.

The Postscript table ("post") contains additional information for Postscript printers, which includes the FontInfo dictionary entry and Postscript names of all the glyphs in the font. Having to name all the glyphs is quite interesting. If you get to read these names, you encounter things like integral, sheva, finalmem, etc.

The control value program table ("prep") contains some TrueType instructions that will be executed whenever font, point size, or transformation matrix changes, and before glyph outline is interpreted. The font program table ("fpgm") contains instructions that will be run when the font is first used.

The baseline table ("BASE") provides information used to align glyphs of different scripts and sizes in the same line of text.

A glyph definition table ("GDEF") contains information about glyph classification, attachment points for streamlining data access and bitmap caching, and ligature caret positioning data. The glyph positioning table ("GPOS") provides precise control over glyph placement for sophisticated text layout and rendering in each script and language system the font supports. Glyph substitution table ('GSUB') contains information for substituting glyphs to render the scripts and language system supported. It can be used to support ligature, contextual glyph substitution, positional glyph variants, etc. The justification table ("JSFT") provides additional control over glyph substitution and positioning in

justified text.

The vertical header table (“vhea”) and vertical metrics table (“vmtx”) contain metrics information for vertical fonts, which are mirror copies of the horizontal header table and the horizontal metrics table.

The digital signature table (“DSIG”) contains the digital signature for an OpenType font to offer some security features. For example, using the digital signature, the operating system can identify the source and integrity of font files before using them, embedded font files can identify the publisher, and the font developer can specify embedding restrictions.

TrueType Collections

With Microsoft's OpenType, multiple OpenType fonts can be packaged in a single font file called a TrueType collection (TTC). TrueType collections are useful for similar fonts that can share a large number of glyphs. For example, the Japanese character set is divided into a small number of kana glyphs and thousands of kanji glyphs. It makes perfect sense for a group of Japanese fonts to have unique designs for the kana glyphs while sharing the kanji glyphs.

Recall that a normal TrueType/OpenType font is made up of one table directory and multiple tables; a TrueType collection file is made up of one TTC header table, multiple table directories (one for each font), and multiple tables (which can be either shared or not shared).

A TTC header table is quite simple; it contains a tag (“ttcf”), version, directory count, and array of offsets to the TrueType table directories.

```
typedef struct
{
    ULONG TTCTag;      // TTC tag 'ttcf'
    ULONG Version;     // TTC version (initially 0x0001000)
    ULONG DirectoryCount; // Number of Table Directories
    DWORD Directory[1]; // offset to TableDirectory. Variable size
} TTC_Header;
```

While font collections save disk space and RAM space, they do break the GetFontData function. With GetFontData, an application is suppose to query TrueType data for a whole font, save it, ship it to another machine, and then later install it on another machine. With font collections, an application is not sure whether the data returned is complete data for a TrueType file, or part of a TrueType font collection. What's worse, some offsets are now relative to the invisible TrueType font collection header, instead of the current TableDirectory. For example, offsets in the TableDirectory structure to individual TrueTypes are relative to the starting of the physical file, which is different in a single font vs. a font collection.

A workaround is to use the TTC font tag to check for the size of the whole font collection. By comparing it with the size of one font, you can find the offset of the current font within a font collection, which can then be used to locate the right tables.

14.5 FONT INSTALLATION AND EMBEDDING

Fonts are shipped as font files. Before they can be used by user applications, fonts need to be installed on the operating system. GDI provides several functions to manage font installation and uninstallation, which can also be used to embed fonts in applications or documents. Here are these functions:

```
BOOL CreateScalableFontResource(DWORD fdwHidden, LPCTSTR lpszFontRes,
LPCTSTR lpszFontFile, LPCTSTR lpszCurrentPath);

int AddFontResource(LPCTSTR lpszFileName);
BOOL RemoveFontResource(LPCTSTR lpFileName);
int AddFontResourceEx(LPCTSTR lpszFileName, DWORD fl,
DESIGNVECTOR * pdv);
BOOL RemoveFontResourceEx(LPCTSTR lpszFileName, DWORD fl,
DESIGNVECTOR * pdv);

HANDLE AddFontMemResourceEx(LPVOID pbFont, DWORD cbFont,
DESIGNVECTOR * pdb, DWORD * pcFonts);
int RemoveFontMemResourceEx(HANDLE fh);
```

Font Resource File

The native font-type format for Windows is bitmap fonts and vector fonts; TrueType, OpenType, and Postscript fonts used to be aliens to the Windows operating system. For bitmap and vector fonts, multiple bitmap or vector-font resources, usually of the same typeface and different sizes, are linked as resources into a 16-bit DLL called the font resource file. Within a font resource file, font resources are attached as a binary resource of type FONT (RT_FONT).

GDI directly supports installation of fonts only in the old 16-bit font resource file format. To install a TrueType font, a *scalable font resource file* needs to be created. A scalable font resource file is in the same 16-bit DLL format, but it does not contain a copy of a TrueType font as its resource. Instead, a scalable font resource file stores the name of a TrueType font file so that GDI knows where to find the file. To create a scalable font resource file, call the GDI function CreateScalableFontResource with an integer flag, the name of the font resource file to be generated, an existing TrueType font file name, and the path to the files if they do not contain a complete path. The fdw Hidden flag informs GDI whether the font should be hidden from other processes on the system. CreateScalableFontResource writes a small font resource file to the disk. The suggested file extension for a TrueType font resource file is .fot, just to distinguish it from .fon for bitmap or vector fonts.

Install Public Fonts

Given a font resource file name, which can be for a bitmap, vector, or TrueType font, AddFontResource installs the corresponding font. Installing a font means adding a font resource file to the system font table, which makes it available to font enumeration, font mapping, logical font creation, and text drawing. A font added by AddFont Resource is available to all applications unless its font resource is created with the hidden flag, which hides it from

font enumeration. But a font installed by AddFont Resource is only available for the current session. After a reboot, the font is not automatically added back to the font table. To permanently install a font in a system, a font needs to be added to the registry.

RemoveFontResource does the opposite; it removes a font resource from the system font table. Current running applications need to be informed about changes in the system font table. It's the responsibility of an application making the change to inform all top-level windows, broadcasting a WM_FONTCHANGE message. For an application using a current list of installed fonts, it needs to handle the WM_FONT CHANGE message to update its font listing.

Install Private or Multiple Master OpenType Fonts

AddFontResourceEx and RemoveFontResourceEx are new functions added for Windows 2000. The second parameter to AddFontResourceEx controls the privacy of the font. If FP_PRIVATE bit is on, the font will not be available to other processes, nor will it be available after the current process terminates; if the FP_NOT_ENUM flag is on, no process can enumerate the font. If either of these flags is on, there is no need to broadcast WM_FONTCHANGE to inform other applications about a font they can't use. RemoveFontResourcEx uses the same parameter as AddFontResourcEx to remove a font installed by AddFontResourcEx.

The last parameter is a pointer to a DESIGNVECTOR structure, used only for Multiple Master OpenType font. A Multiple Master OpenType font is built using Postscript's Type 1 font technology. Multiple Master OpenType fonts have one or more font characteristics, or axes, that can be set to values within a certain range, to make fine adjustments of the appearance of a font. For example, a Multiple Master OpenType font can have a font weight axis in the range of thin (300) to heavy (900). DESIGNVECTOR is a variable-size data structure that specifies the number of axes and a value for each axis.

Install Fonts from Memory Image

To install a TrueType font using AddFontResource or AddFontResourcEx, you need two physical files on the disk—one for the TrueType font file and another for the font resource file. This makes applications using private fonts harder to program and hide completely from other applications. AddFontMemResourceEx, provided by Windows 2000, tries to solve these two problems by allowing installation of fonts from memory image. Its first two parameters specify the location and size of a memory block containing one or several font resources, and the third parameter is a DESIGN VECTOR pointer for Multiple Master OpenType font. AddFontMemResourceEx installs the fonts from their memory image, returning a handle and the number of fonts installed.

Fonts installed by AddFontMemResourceEx are always private to the calling application, which can remove them using RemoveFontMemResourceEx using the handle returned. Or if the application fails to do that, fonts added will be removed when the process terminates.

The memory block passed to AddFontMemResourceEx is in raw font resource format, not the 16-bit DLL form of font resource file format. Compared with AddFontResource and AddFontResourceEx, AddFontMemResourceEx is much easier to use so that a single application can install and use some fonts all by itself.

Font Embedding

One of the major problems with document portability among different machines is fonts. With the proper fonts installed on a system, you may fine-tune the formatting of a document. But when a document is shipped to another

machine without the same set of fonts installed, it may look dramatically different. This may happen to applications using special fonts, documents generated by a word processor, web pages, or even print spooler files when printing to a remote server.

Font embedding is a technology of bundling a document with the special fonts it uses so that these embedded fonts can be installed on another machine to give the document the same look.

Embedding of fonts must conform to fonts license agreements. There are six levels of font embeddability defined for a TrueType/OpenType font, as indicated by the `fsType` flag in a font's OS/2 and Windows metrics table ("OS/2").

- Installable embedding (0x0000): the font may be embedded in documents and permanently installed on remote systems. Most fonts that come with Microsoft OS allow installable embedding.
- Editable embedding (0x0008): the font may be embedded in documents, but can only be installed *temporarily* on the remote system. For example, a font may be embedded in a Word document, which can be viewed and edited on a remote machine. But when WinWord is closing, the font must be removed.
- Print & Preview embedding (0x0004), also known as read-only embedding: the font may be embedded in documents, but must only be installed temporarily on the remote system. Documents can only be opened read-only. A read-only embedded font should be encrypted in the document. On the remote machine, it should be decrypted to a hidden file without a .ttf file extension, installed as a hidden font, used only to view or print the document, and deleted when the viewing application quits.
- No subsetting (0x0100): font may not be subset when embedding.
- Embed bitmaps only (0x0200): only bitmaps contained in the font can be embedded. If the font only contains glyph outlines, it can't be embedded.
- Restricted license (0x0002): no embedding is allowed; the font may not be embedded in a document.

Note that font embeddability only mentions embedding fonts in a document, not an application. According to MSDN, a font cannot be embedded in an application, nor can an application be distributed with documents with embedded fonts.

To check whether a TrueType/OpenType font can be licensed, and whether it's read-only, use `GetOutlineTextMetrics`. `GetOutlineTextMetrics` returns an OUTLINE TEXTMETRICS structure, whose contents is quite close to the OS/2 and Windows metrics table ("OS/2" table) within a TrueType font file. Its `otmfstype` field has the same value as the `fsType` field described above.

[Listing 14-4](#) shows two routines for TrueType/OpenType font installation and un installation. `InstallFont` accepts an in-memory image of a TrueType/OpenType font, creates a font file and a font resource file, and then installs the font. `RemoveFont` removes a font from the system's font list and deletes the related .ttf and .fot files. Both functions accept an option parameter, which controls whether the font should be public, hidden, private, not enumerateable, or installed directly from memory image. The option parameter determines the GDI function to call for installation and un installation.

Listing 14-4 Font Installation and Uninstallation

```
#define FR_HIDDEN 0x01  
#define FR_MEM    0x02
```

```
BOOL RemoveFont(const TCHAR * fontname, int option, HANDLE hFont)
{
    if ( option & FR_MEM )
    {
        return RemoveFontMemResourceEx(hFont);
    }
    TCHAR ttffile[MAX_PATH];
    TCHAR fotfile[MAX_PATH];

    GetCurrentDirectory(MAX_PATH-1, ttffile);
    _tcscpy(fotfile, ttffile);
    wsprintf(ttffile + _tcslen(ttffile), "\\%s.ttf", fontname);
    wsprintf(fotfile + _tcslen(fotfile), "\\%s.fot", fontname);

    BOOL rslt;

    switch ( option )
    {
        case 0:
        case FR_HIDDEN:
            rslt = RemoveFontResource(fotfile);
            break;

        case FR_PRIVATE:
        case FR_NOT_ENUM:
        case FR_PRIVATE | FR_NOT_ENUM:
            rslt = RemoveFontResourceEx(fotfile, option, NULL);
            break;

        default:
            assert(false);
            rslt = FALSE;
    }

    if ( ! DeleteFile(fotfile) )
        rslt = FALSE;

    if ( ! DeleteFile(ttffile) )
        rslt = FALSE;

    return rslt;
}

HANDLE InstallFont(void * fontdata, unsigned fontsize,
                  const TCHAR * fontname, int option)
{
    if ( option & FR_MEM )
    {
```

```
DWORD num;
return AddFontMemResourceEx(fontdata, fontsize, NULL, & num);
}

TCHAR ttffile[MAX_PATH];
TCHAR fotfile[MAX_PATH];

GetCurrentDirectory(MAX_PATH-1, ttffile);
_tcscpy(fotfile, ttffile);

wsprintf(ttffile + _tcslen(ttffile), "\\%s.ttf", fontname);
wsprintf(fotfile + _tcslen(fotfile), "\\%s.fot", fontname);

HANDLE hFile = CreateFile(ttffile, GENERIC_WRITE, 0, NULL,
    CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL |
    FILE_FLAG_SEQUENTIAL_SCAN, 0);

if ( hFile==INVALID_HANDLE_VALUE )
    return NULL;

DWORD dwWritten;
WriteFile(hFile, fontdata, fontsize, & dwWritten, NULL);
FlushFileBuffers(hFile);
CloseHandle(hFile);

if ( ! CreateScalableFontResource(option & FR_HIDDEN, fotfile,
    ttffile, NULL) )
    return NULL;

switch ( option )
{
    case 0:
    case FR_HIDDEN:
        return (HANDLE) AddFontResource(fotfile);

    case FR_PRIVATE:
    case FR_NOT_ENUM:
    case FR_PRIVATE | FR_NOT_ENUM:
        return (HANDLE) AddFontResourceEx(fotfile, option, NULL);

    default:
        assert(false);
        return NULL;
}
}
```

Based on the functions in [Listing 14-4](#), we created a simple demo program, Font Embed. FontEmbed is a simple dialog-box-based application, as illustrated in [Figure 14-19](#).

Figure 14-19. Font-embedding demo.



FontEmbed's dialog box has three main functional buttons. Button "Generate" generates a "document" that embeds user-selected TrueType/OpenType fonts, with a simple encryption. Button 'Load' loads the document generated and installs the fonts embedded in it, with the installation option controlled by the radio buttons on the right. Button "Unload" removes all the font resources installed. The space on the right shows display results generated with the embedded fonts.

[Figure 14-19](#) is generated with three free TrueType fonts from the HP Font Smart Homage Page at www.fontsmart.com: Euro Sign, Ozzie Black, and Ozzie Black Italic. When these fonts are not installed, a default symbol font will be used to display the first line, and Arial will be used to display the next two lines. After the fonts are properly installed, the dialog box will show what's in [Figure 14-19](#). But after the fonts are uninstalled, the dialog box returns to its original display.

If you don't have these fonts installed on your system, download them; if you already have them, search for some new shareware or freeware fonts on the Internet. Run the FontEmbed program; play with different installation options to verify whether a font is available to the current application and other applications after it is installed.

System Font Lists

On Windows NT/2000, fonts permanently installed on the system are listed under the registry key:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts`

When the system boots up, some component in the system is responsible for installing these fonts to the runtime system font table, so that they can be used. These fonts are physical fonts which are shared by all processes in the

system.

The graphics engine actually keeps three tables of fonts internally in kernel mode address space—one for public fonts, one for private fonts, and one for device fonts which are normally provided by high-end printers like Postscript printers. Permanent fonts provided by the operating system are normally available to all applications, so they definitely belong to the public font table. If an OpenType/TrueType font resource file is created by a hidden flag, or if FR_HIDDEN flag is used in CreateFontResourceEx, or if CreateFontMemResourceEx is used, the font will be put into the private font table. If the FR_NOT_ENUM flag is used without FR_HIDDEN, a font is still put into the public font table. The system font list contains the full path to the font file. In the case of memory font resource, a pseudo file name like “MEMORY-1” will be used.

The GDI debugger extension provides three commands: “pubft,” “pvtft,” and “devft,” to display the contents of system font tables. They can be used in the Fosterer host program developed in [Chapter 3](#).

[< BACK](#) [NEXT >](#)

14.6 SUMMARY

This chapter covers the foundation of text display in Windows graphics programming, fonts. The chapter starts with a discussion about the basic concepts around fonts: characters, character set, glyph, code page, and character-to-glyph mapping. It then examines the details of three basic font types used by Windows: bitmap fonts, vector fonts, and TrueType fonts. We look at how glyphs are represented in different types of fonts, and how they can be displayed using a simple rasterizer. We finish the chapter with a discussion of font installation, uninstallation, and font embedding in documents.

Based on the solid understanding of fonts built in this chapter, we move to the exciting practical application side of fonts—that is text display—in [Chapter 15](#).

Future Reading

An excellent reference on digital typography is Donald E. Knuth's book, *Digital Typography*, published in 1999. This book contains 34 articles written by the author on the subject of digital typography, popularly called “desktop publishing.” Compared with the best font technology we’re using today on the Windows platform, TrueType fonts, Knuth’s METAFONT is much more versatile and flexible. A glyph in METAFONT is described by computer programs that are controlled by parameters. So you can easily adjust these parameters to change the weight, stem width, serif style, serif size, randomness, etc. Recall that a TrueType glyph is described by second-degree Bezier curves with the assistance of glyph instructions whose sole purpose is to make the glyph image nicer in small point sizes. Literally, a metafont is a schematic description of how to draw a family of fonts, not simply the drawings themselves.

With Adobe's ATM (Adobe Type Manager) and the new OpenType technology, more and more Postscript fonts are being used in the Windows environment. Every time you use Acrobat Reader to view a .pdf document, you're enjoying Postscript fonts. The official reference on Postscript is *Postscript Language Reference*, third edition, written by Adobe Systems Incorporated; Chapter 5 is devoted to Postscript fonts. *Postscript Language Reference* can also be downloaded from www.adobe.com.

TrueType is an Apple technology. To read the original *TrueType Reference Manual* and lots of technical articles on the TrueType technology, visit fonts.apple.com. You can find detailed information on TrueType font files, glyph instruction set, how to instruct fonts, the font engine, etc.

Microsoft also provides a very informative typography page at www.microsoft.com/typography. You can find specifications like the OpenType specification, the TrueType font file specification, and the TrueType Open Specification. Parts of them are available on the MSDN Library under Specifications\Applications. Microsoft also provides quite a few useful TrueType font tools.

- The font property extension allows the displaying of extended properties of TrueType fonts. It provides features, embedding, hinting/font smoothing, names, license, character set, and links information.
- The font property editor is a tool to allow font designers to add links, descriptions, and license information to their fonts.
- TTFDump is a command-line tool that dumps the contents of TrueType font files.
- Flint is a 32-bit TrueType font-testing tool.

Also provided is the OpenType embedding SDK. The bitmap font format is documented in the Knowledge Base article Q65/23.

Hewlett-Packard provides a font manager and a set of TrueType fonts in a package called FontSmart. FontSmart's home page is at www.fontsmart.com.

Sample Programs

There are two sample programs for this chapter, as listed in [Table 14-4](#).

Table 14-4. Sample Programs for Chapter 14

Directory	Description
\Samples\Chapt_14\Font	Demonstrates character set, code page, glyph, font family, font enumeration, bitmap font, vector font, and TrueType font.
\Samples\Chapt_14\FontEmbed	Demonstrates font installation, uninstallation, and font embedding in documents.

[< BACK](#) [NEXT >](#)

Chapter 15. Text

Fonts, as we discussed in the last chapter, serve as a basic building block for text drawing, which is the topic of this chapter. In this chapter, we are going to discuss logical fonts, text drawing functions, simple text formatting, high-quality and high-precision text formatting, and text drawing effects.

[< BACK](#) [NEXT >](#)

15.1 LOGICAL FONTS

[Chapter 14](#) covers substantial details about the three major font technologies used in Windows programming—that is, bitmap fonts, vector fonts, and TrueType/OpenType fonts. Even with in-depth knowledge of physical fonts, working directly with them is hard, time consuming, and not a good way to spend an application programmer's time.

To make text drawing easier for Windows applications, physical fonts are not handled directly by application programs, not even the graphics engine. An application program normally deals only with logical fonts through a set of API built around logical fonts. Physical fonts are handled only by font drivers, which are at the same level as graphics device drivers. The Windows NT/2000 graphics engine implements three font drivers for three types of fonts supported by Microsoft directly. ATM fonts are supported by a separate ATM font driver (`atmfd.dll`). The graphics engine communicates with the font drivers to provide support for logical fonts.

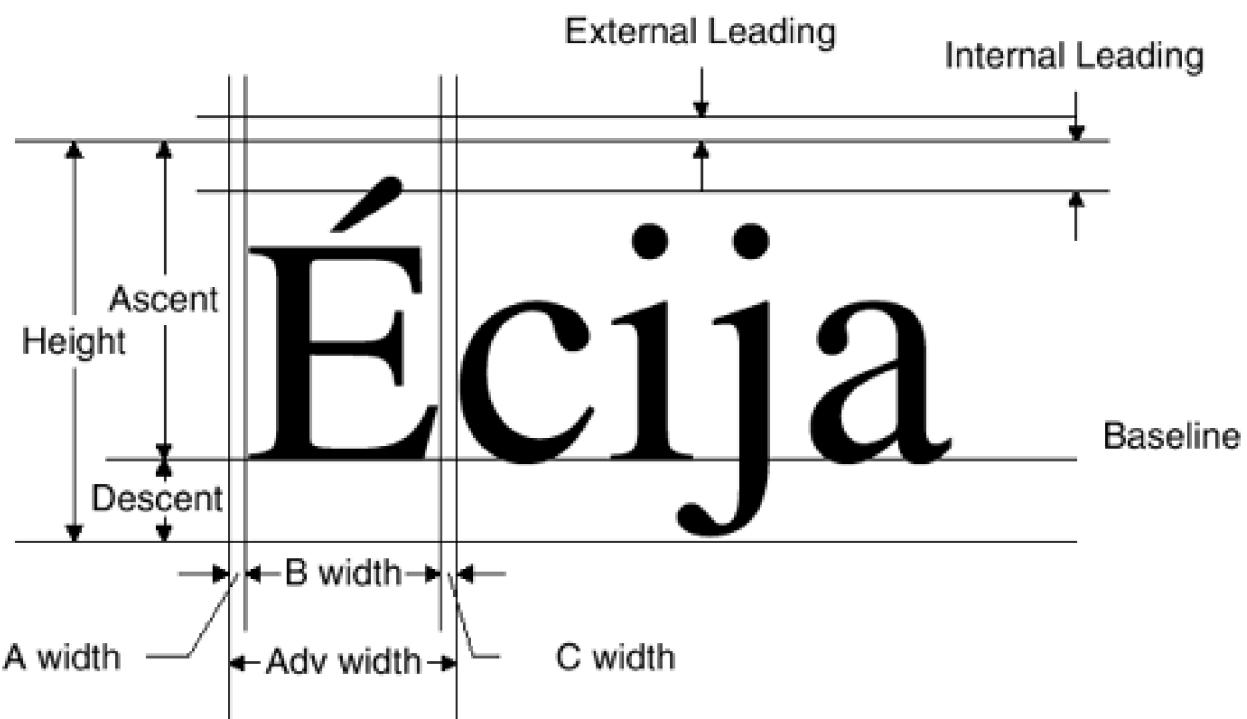
Compared with physical fonts, logical fonts provide substantial benefits:

- Logical fonts provide device independence. A logical font is generated from a description of a user's requirements for a font. The graphics engine is responsible for matching font requirements with physical fonts currently installed on the system. The system will be able to find a close match even if certain fonts are not installed; it may also choose different fonts for different graphics devices to meet an application's requirement.
- Logical fonts support code pages. To fetch a glyph in a TrueType font for a character in a code page, you have to search through character-to-glyph index mapping. Logical fonts hide glyph indexes away from applications, using code pages and character points.
- Logical fonts provide fixed-size instantiation of fonts. Glyph descriptions in a font are generic templates for generating glyphs of different sizes and rotation angles. A bitmap font normally contains different font resources for different point sizes. Vector fonts and TrueType/OpenType fonts are fully scalable and transformable. When a logical font is selected into a device context, an instantiation of a font for a particular point size and rotation angle is created. This design allows the graphics engine and font drivers to cache scaled and rasterized versions of glyphs to enhance system performance.
- Logical fonts provide simulation of features. Certain features we commonly associate with fonts, such as underlining and striking out, are not implemented by the physical fonts; instead, they are simulated by GDI. GDI is also able to simulate italic or bold fonts when the exact physical fonts are not available.

Windows Typography Terminology

Before looking at the details of logical fonts, let's iron out a few Windows typography terms. Note that Windows GDI may use terms in a slightly different way than similar terms in TrueType font specifications, or traditional typography. [Figure 15-1](#) illustrates the terms we will commonly encounter in GDI text formatting.

Figure 15-1. GDI typography terminology.



An imaginary line for aligning glyphs vertically is called the *baseline*. Normally the lowest point of most upper-case letters is almost exactly on the baseline. Characters are defined in character cells of the same height. The distance from the top of the character cell in a font to the baseline is called the *ascent*. There may be some white space between the ascent line and the highest glyph. So the ascent used in GDI is a little bit different from the typography ascent used in TrueType fonts. The distance between the baseline and the bottom of the character cell is called the *descent*. Again, there may be space between the bottom of the lowest glyph and the descent line. The sum of the ascent and the descent is called the *height* of a font.

The top part of the space between the ascent line and the baseline is usually for accent marks and other diacritical characters. The height of this space is called *internal leading*. When lines of text form a paragraph, extra space is added between the descent line of the previous line and the ascent line of the next line. The amount of this extra space is called *external leading*.

Text size is measured in points. In traditional printing, a point is 0.01389 inch, or 1/71.99424 inch. In desktop publishing, a point is rounded to a perfect 1/72 inch, or an inch is exact 72 points. The difference between them is 1/12,500, too small to be noticeable for practical purposes.

When referring to font or text, point size corresponds to the amount of ascent + descent - internal leading, or height - internal leading. Note that point size does not include internal leading or external leading. For example, for a 10-point text paragraph its ascent + descent - internal leading is 10 points, which is 13.3 pixels on a 96-dpi screen display, 83.3 pixels on a 600-dpi printer. A 10-point text paragraph normally has a 12-point or 13-point line distance, or 6 lines to 5.54 lines per inch.

Horizontal terminology used by GDI is almost the same as in TrueType fonts. The distance between one character and the next is called the *advance width*. The advance width is divided into three portions. The left part is normally white space before the leftmost part of a glyph. It's called the *A-width*, or left-side bearing in TrueType. The middle part is the width of the actual character cell, named the *B-width*. The right part is normally white space after the rightmost part of a glyph. It's called *C-width*, or right-side bearing in TrueType. The advance width of a character is A-width + B-width + C-width. Both A-width and C-width could be negative to align glyphs tighter, especially for italic fonts.

Stock Fonts

A logical font is a GDI object, which describes requirements for a particular instantiation of a physical font. Like other GDI objects, logical font objects are managed by GDI as black boxes. User applications can only deal with them through logical font handles, which are of type HFONT.

The system provides seven stock GDI logical fonts, which are used by the operating system to display its user interface and are also shared by applications. Stock logical font handles can be retrieved using GetStockObject(DEFAULT_GUI_FONT) or GetStockObject(SYSTEM_FONT), etc. Stock logical fonts are mostly bitmap fonts used for the fast display of a window's title bar, menu, dialog box, etc. [Figure 15-2](#) shows the seven stock fonts, displayed on a 96-dpi monitor. For each stock font, it shows how to get its handle using GetStockObject, and its related LOGFONT structure, which will be described below.

Figure 15-2. Stock fonts at 96 dpi.

DialogBaseUnits: baseunitX=8, baseunitY=16

GetDeviceCaps(LOGPIXELSX)=96, GetDeviceCaps(LOGPIXELSY)=96

```
GetStockObject(DEFAULT_GUI_FONT)
{11, 0, 0, 0, 400, 0, 0, 0, 0, 0, MS Shell Dlg}

GetStockObject(OEM_FIXED_FONT)
{12, 8, 0, 0, 400, 0, 0, 0, 255, 1, 2, 2, 49, Terminal}

GetStockObject(ANSI_FIXED_FONT)
{12, 9, 0, 0, 400, 0, 0, 0, 0, 2, 2, 1, Courier}

GetStockObject(ANSI_VAR_FONT)
{12, 9, 0, 0, 400, 0, 0, 0, 2, 2, 2, MS Sans Serif}

GetStockObject(SYSTEM_FONT)
{16, 7, 0, 0, 700, 0, 0, 0, 1, 2, 2, 34, System}

GetStockObject(DEVICE_DEFAULT_FONT)
{16, 7, 0, 0, 700, 0, 0, 0, 1, 2, 2, 34, System}

GetStockObject(SYSTEM_FIXED_FONT)
{15, 8, 0, 0, 400, 0, 0, 0, 1, 2, 2, 49, Fixedsys}
```

You can find stock fonts being used by a window's title bar, menu display, and text display in various controls. The pixel size of stock fonts changes when screen logical resolution changes. For example, a normal screen's logical resolution is set at 96 dpi, the so-called "small font" mode. But you can use the control panel to switch it to the "large font" mode, which is at 120 dpi. When the screen switches from the small font mode to the large font mode, all stock fonts need to be remapped to physical fonts at a larger size, which normally requires rebooting of the system. After that, all title bars, menu bars, controls, and dialog boxes get enlarged to accommodate the change in font size.

To design a user interface which looks perfect in both small font and large font mode is a real challenge. You can call GetSystemMetrics to query various system metrics, which include current title bar and menu bar size. For example, GetSystemMetrics(SM_CYMENU) returns the height of a single-line menu bar. Dialog boxes are designed

using device-independent *dialog template units*. When creating a dialog box, coordinates in dialog template units need to be translated to screen pixels, according to the current *dialog base units*, using the formulae:

```
pixelX = (templateunitX * baseunitX) / 4;  
pixelY = (templateunitY * baseunitY) / 8;
```

Dialog base units are the average width and height of characters in a stock font, which is used to display controls in a dialog box. They can be queried using GetDialogBaseUnits. At 96 dpi, baseunitX is 8 and baseunitY is 16, so each dialog template unit is translated to two screen pixels. At 120 dpi, baseunitX is 10 and baseunitY is 20, so each dialog template unit is translated to 2.5 screen pixels. The end result is that, when switching from the small font mode to the large font mode, your dialog box gets 12.5% bigger. While this may sound like a good deal, for you to get high-resolution text for free, not all user interface elements can handle the enlargement at the same pace. If you have bitmaps and icons in a dialog box, or have a modalless dialog box embedded in nondialog box windows, bitmaps and icons may look shrunken in the enlarged dialog box, text may get clipped, and dialog-box based windows may misalign with nondialog-box based windows.

Creating Logical Fonts

Stock fonts are normally only for simple user-interface-related screen display. For anything else, you need to create your own logical fonts. GDI provides three functions for creating custom logical fonts.

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

```
typedef struct tagENUMLOGFONTEX {  
    LOGFONT elfLogFont;  
    TCHAR elfFullName[LF_FULLFACESIZE];  
    TCHAR elfStyle[LF_FACESIZE];  
    TCHAR elfScript[LF_FACESIZE];  
} ENUMLOGFONTEX, *LPENUMLOGFONTEX;
```

```
typedef struct tagENUMLOGFONTEXDV {
```

```
ENUMLOGFONTEXelfEnumLogfontEx;
DESIGNVECTORelfDesignVector;
} ENUMLOGFONTEXDV,*PENUMLOGFONTEXDV

HFONT CreateFont(int nHeight, int nWidth, int nEscapement,
    int nOrientation, int fnWeight, DWORD fdwItalic,
    DWORD fdwUnderline, DWORD fdwStrikeOut, DWORD fdwCharSet,
    DWORD fdwOutputPrecision, DWORD fdwClipPrecision,
    DWORD fdwQuality, DWORD fdwPitchAndFamily, LPCTSTR lpszFace);
HFONT CreateFontIndirect(CONST LOGFONT * lplf);
HFONT CreateFontIndirectEx(const ENUMLOGFONTEXDV * penumIfex);
```

The input parameters to these three functions serve as a description for a user's requirements for a logical font. CreateFont uses 14 parameters to describe a logical font, the record number of parameters for a GDI function. CreateFont accepts a pointer to LOGFONT structure, which packs the same 14 parameters into a single entity. CreateFontIndirectEx is a new function provided only in Windows 2000. It accepts a pointer to an ENUMLOGFONTEXDV structure. The ENUMLOGFONTEXDV structure adds a DESIGNVECTOR field to the ENUMLOGFONTEX structure, which adds a font's unique name, style name, and script name to LOGFONT structure. So the requirements for a logical font are basically described by a LOGFONT structure, as required by CreateFontIndirect; CreateFont just uses an unfolded version of the LOGFONT structure, and CreateFontIndirectEx just uses an extended version of it.

LOGFONT and related structures are really important to one's understanding of fonts in GDI. So there is a need to go through some important fields within them.

- **IfHeight.** Specifies the desired font height in logical device units. If it's 0, a default font height around 12 points will be used. If the value is positive, it represents the required font height—that is, the ascent plus the descent of a font. If the value is negative, it represents the required point-size height—ascent + descent - internal leading.
- **IfWidth.** Specifies the desired font width in logical device units. If it's zero, the aspect ratio of the graphics device is matched with the physical font aspect ratio to find the closest match. Normally, display or print devices have square resolution—for example, 120 by 120 dpi, or 600 by 600 dpi. For these square resolutions, 0 width will favor fonts with a 1:1 aspect ratio. It's possible to change IfWidth to get 'non-square' fonts.
- **IfEscapement.** Specifies the angle, in tenths of a degree counterclockwise, of the baseline of the text and the x-axis of a device. For example, an escapement of 900 means all text goes bottom-up with the baseline parallel to the y-axis.
- **IfOrientation.** Specifies the angle, in tenths of a degree counterclockwise, of each character's baseline and the x-axis of a device. Note that the orientation controls the angle for each character on a line of text, while escapement controls the baseline of a line of text. When the device is in advanced graphics mode (GM_ADVANCED), which is only available on Windows NT/2000, escapement and orientation can be independent. When running in compatible graphics mode (GM_COMPATIBLE), which is the only mode on Windows 95/98, the IfEscapement field determines IfOrientation, and they should be set to the same value.
- **IfWeight.** Specifies the weight, or boldness, of the font using values in the range 0 through 1000. FW_FONTCARE (0) lets GDI choose a font of any weight, FW_NORMAL (400) is medium, and FW_HEAVY (900) is normally the boldest.

- **IfItalic.** Specifies that the italic font is preferred if set to TRUE.
- **IfUnderline.** Specifies that underlining should be simulated in a text drawing if set to TRUE.
- **IfStrikeOut.** Specifies that striking out (a line in the middle of text) should be simulated in text drawing if set to TRUE.
- **IfCharSet.** Specifies the character set needed for the font, which also determines the code page in which text strings will be passed to GDI text drawing functions. Character sets and code pages are explained in [Section 14.1](#). A special value for IfCharSet is DEFAULT_CHARSET. On Windows 95/98 other fields will determine the matching font; on Windows NT/2000, the default character set for the current system locale will be used. For example, when the system locale is English, ANSI_CHARSET will be used. If you're using a special language, IfCharSet is very important in matching the right font, because there may be only a few physical fonts which provide the glyphs for the character set you want.
- **IfOutPrecision.** Specifies preferences in matching logical fonts with physical fonts.
OUT_DEFAULT_PRECIS specifies the default behavior. OUT_DEVICE_PRECIS favors device fonts. OUT_RASTER_PRECIS favors raster bitmap fonts. OUT_OUTLINE_PRECIS (Windows NT/2000 only) favors TrueType fonts and other outline fonts. OUT_TT_PRECIS favors TrueType fonts. OUT_TT_ONLY_PRECIS allows only TrueType fonts.
- **IfClipPrecision.** Specifies how to clip characters that are partially clipped. There are several flags defined, but only CLIP_DEFAULT_PRECIS seems to be actually used with regard to clipping, specifying the default clipping behavior. If IfClipPrecision is CLIP_EMBEDDED, a read-only embedded font can be used. If it is CLIP_LH_ANGLES, it tells the device font to rotate the glyph based on whether the logical coordinate system is left-handed or right-handed; otherwise device fonts always rotate counterclockwise.
- **IfQuality.** Specifies the glyph output quality. DEFAULT_QUALITY tells GDI that the appearance of the character does not matter. DRAFT_QUALITY tells GDI that glyph quality is less important than font size, allowing GDI to scale bitmap fonts to the right size with possible distortion. PROOF_QUALITY tells GDI that glyph quality is more important than font size, thereby disabling the scaling of bitmap fonts. For TrueType, DRAFT_QUALITY and PROOF_QUALITY flags are not important because glyph outline can be freely scaled. ANTIALIASED_QUALITY allows GDI to antialias the text display if the font supports it and the font size is not too small or too big. NONANTIALIASED_QUALITY disables antialiasing.
- **IfPitchAndFamily.** Specifies font pitch and family in a single field. The lower 2 bits can be DEFAULT_PITCH for default behavior, FIXED_PITCH for fixed-width fonts, or VARIABLE_PITCH for variable-width fonts. Bits 4–7 specify the font family using FF_DECORATIVE, FF_DONT CARE, FF_MODERN, FF_ROMAN, FF_SCRIPT, and FF_SWISS. Font family styles are explained in [Section 14.1](#).
- **IfFaceName.** Specifies the font typeface name. Typeface names for currently installed fonts can be enumerated using EnumFontFamilies.
- **elfFullName.** Specifies a font's unique name, which includes the company name, typeface name, styles, etc.
- **elfStyle.** Specifies the font style—for example, Bold Italic.
- **elfScript.** Specifies the language script name required—for example, Cyrillic.

- **elfDesignVector.** Specifies the axes for a Multiple Master OpenType font.

CreateFont, CreateFontIndirect, or CreateFontIndirectEx creates a GDI logical font object and returns its handle to its caller. Given a GDI object handle, you can use GetObject to return the LOGFONT or ENUMLOGFONTEX structure defining the logical font. Like other GDI objects, logical font objects should be deleted when not needed using DeleteObject.

To create a logical font, the first parameter to figure out is the height of the font in logical coordinates. If the point size of the required font is known, a reference device context is needed to translate the point size to logical coordinate space size. The following function translates the point size to logical coordinate space size.

```
// convert point size to logical coordinate space size
int PointSizetoLogical(HDC hDC, int points, int divisor=1)
{
    POINT P[2] = // POINTs in device space whose distance is the height
    {
        { 0, 0 },
        { 0, ::GetDeviceCaps(hDC, LOGPIXELSY) * points / 72 / divisor }
    };

    DPtoLP(hDC, P, 2); // map device coordinate to logical size
    return abs(P[1].y - P[0].y);
}
```

The PointSizetoLogical function takes a reference device context handle, a point size, and an optional divisor for the point size to make calculation more accurate. It first converts the point size to height in pixels based on the device's vertical resolution, then converts it to height in logical coordinate space. To calculate height for a 12-point font, just call PointSizetoLogical(hDC, 12); for a 12.25-point font, call PointSizetoLogical(hDC, 1225, 100). On a high-resolution device like a 1200-dpi printer, each pixel is 0.06 point, so fractional point size can make a difference in text formatting.

Setting up the 14 fields in LOGFONT or 14 parameters to CreateFont is a tedious and error-prone job. [Listing 15-1](#) shows a simple KLogFont class which encapsulates the LOGFONT structure for logical fonts.

Listing 15-1 KLogFont Class for LOGFONT Structure Encapsulation

```
class KLogFont
{
public:
    LOGFONT m_If;

    KLogFont(int height, const TCHAR * typeface=NULL)
    {
        m_If.lfHeight      = height;
        m_If.lfWidth       = 0;
        m_If.lfEscapement  = 0;
        m_If.lfOrientation = 0;
```

```
m_If.IfWeight      = FW_NORMAL;
m_If.IfItalic     = FALSE;
m_If.IfUnderline   = FALSE;
m_If.IfStrikeOut  = FALSE;
m_If.IfCharSet    = ANSI_CHARSET;
m_If.IfOutPrecision = OUT_TT_PRECIS;
m_If.IfClipPrecision = CLIP_DEFAULT_PRECIS;
m_If.IfQuality    = DEFAULT_QUALITY;
m_If.IfPitchAndFamily = DEFAULT_PITCH | FF_DONTCARE;

if ( typeface )
    _tcscopy(m_If.IfFaceName, typeface, LF_FACESIZE-1);
else
    m_If.IfFaceName[0] = 0;
}

HFONT CreateFont(void)
{
    return ::CreateFontIndirect(& m_If);
}

int GetObject(HFONT hFont)
{
    return ::GetObject(hFont, sizeof(m_If), & m_If);
}
};
```

The KLogFont class reduces the 14 parameters to 2 values, the rest being filled by meaningful default values, which can be changed by accessing a public member variable `m_If`. Here is a simple case, creating a 36-point italic logical font for "Times New Roman."

```
KLogFont lf(- PointSizetoLogical(hDC, 36), "Times New Roman");
lf.m_If.IfItalic = TRUE;
HFONT hFont = lf.CreateFont();
```

Logical to Physical Font Mapping

A new logical font created with `CreateFont`, `CreateFontIndirect`, or `CreateFontIn directEx` is not associated with any physical font because it's not associated with any device context yet. Once a logical font is selected into a device context, GDI needs to find the right physical font for the logical font through a process called *font mapping*.

Font mapping matches a logical font's requirements with all the fonts available for a graphics device. Besides fonts permanently installed on the system, embedded fonts and device fonts may also be available for font mapping. [Chapter 14](#) discusses how to embed fonts in a document, install them when a document is opened, and enumerate fonts currently installed.

Device fonts are fonts provided by a graphics device driver and implemented by a graphics device in hardware. For

example, a PostScript printer normally supports dozens of PostScript fonts. So a PostScript printer will report them as device context to tell GDI to use them for text output. Usually, a user application only queries a device context's metrics information for text formatting. When actual text drawing commands are sent to it, a printer driver can generate commands to use these device fonts, instead of having to download TrueType fonts to a PostScript printer.

For Windows's traditional bitmap and vector fonts, their font header structure is very similar to the TEXTMETRIC structure used in GDI. TEXTMETRIC contains almost all the information a LOGFONT has, such as height, average width, weight, character set, pitch and family, italic, underline, strike-out, etc. So it's quite easy for GDI to match LOGFONT with font header structure. Considered from another angle, the way we are creating logical fonts using LOGFONT was really designed to work with bitmap and vector fonts.

TrueType and OpenType fonts have much more precise information about the characteristics of a physical font. A TrueType/OpenType font's font metrics information is put into its OS/2 and Window metrics table, which is similar to GDI's OUTLINETEXTMETRIC structure.

The most important factor in matching logical fonts with physical fonts is the character set. While most fonts support the ANSI character set, character sets corresponding to other languages may be supported by only a few fonts installed on the system. For example, very few fonts support the symbol character set. When an application asks for a particular character set, GDI has to try its best to find a font supporting it; otherwise characters may be displayed using totally wrong glyphs. Bitmap and vector fonts support only a single character set; a single TrueType font can support several to a dozen character sets. Each TrueType/OpenType font has 64-bit flags that specify the code pages supported by the font.

The output precision field is also very important. It restricts font-mapping candidates to certain types of fonts. For example, OUT_OUTLINE_PRECIS favors outline fonts. Monospace fonts or variable-width fonts differ greatly in appearance, so pitch is an important factor in mapping fonts.

Typeface name plays a very important role. When a physical font with the exact typeface name is found, the GDI font mapper can take a shortcut if several other important factors like character set, height, italicness, and weight have perfect matches. The system registry maintains a list of font typeface name substitutions, which will be applied to the typeface name required by user. For example, the list tells the font mapper to replace "Helv" with "MS Sans Serif," "MS Shell Dlg" with "Microsoft Sans Serif," or "Times" with "Times New Roman."

Other major factors in font mapping include font family, font height, width, and aspect ratio for bitmap fonts. Font weight, underline, strike-out, font height, width, and aspect ratio for outline fonts are less important.

PANOSE Typeface Matching

Font mapping using information provided in LOGFONT is rather ad hoc, as you can see. But that's the only information provided by CreateFont, CreateFontDirect, and the font header in bitmap and vector fonts. The situation could be really bad if a document is transferred to a computer that does not have the same set of fonts installed. If the document stores its font information in a LOGFONT structure with a face name like "Antique Olive Compact," what can GDI do to map it to the right physical font?

For TrueType/OpenType fonts, although OUTLINETEXTMETRIC contains a copy of the simple TEXTMETRIC structure used by bitmap and vector fonts, the most important font-classification mechanism is the PANOSE structure.

PANOSE is a typeface-matching system that tries to classify and match type fonts according to their appearance. What's used in TrueType fonts now is the PANOSE 1.0 structure, which uses 10 single-byte values to characterize

the appearance of fonts.

```
typedef struct tagPANOSE
{
    BYTE bFamilyType;
    BYTE bSerifStyle;
    BYTE bWeight;
    BYTE bProportion;
    BYTE bContrast;
    BYTE bStrokeVariation;
    BYTE bArmStyle;
    BYTE bLetterform;
    BYTE bMidline;
    BYTE bXHeight;
} PANOSE, * LPPANOSE;
```

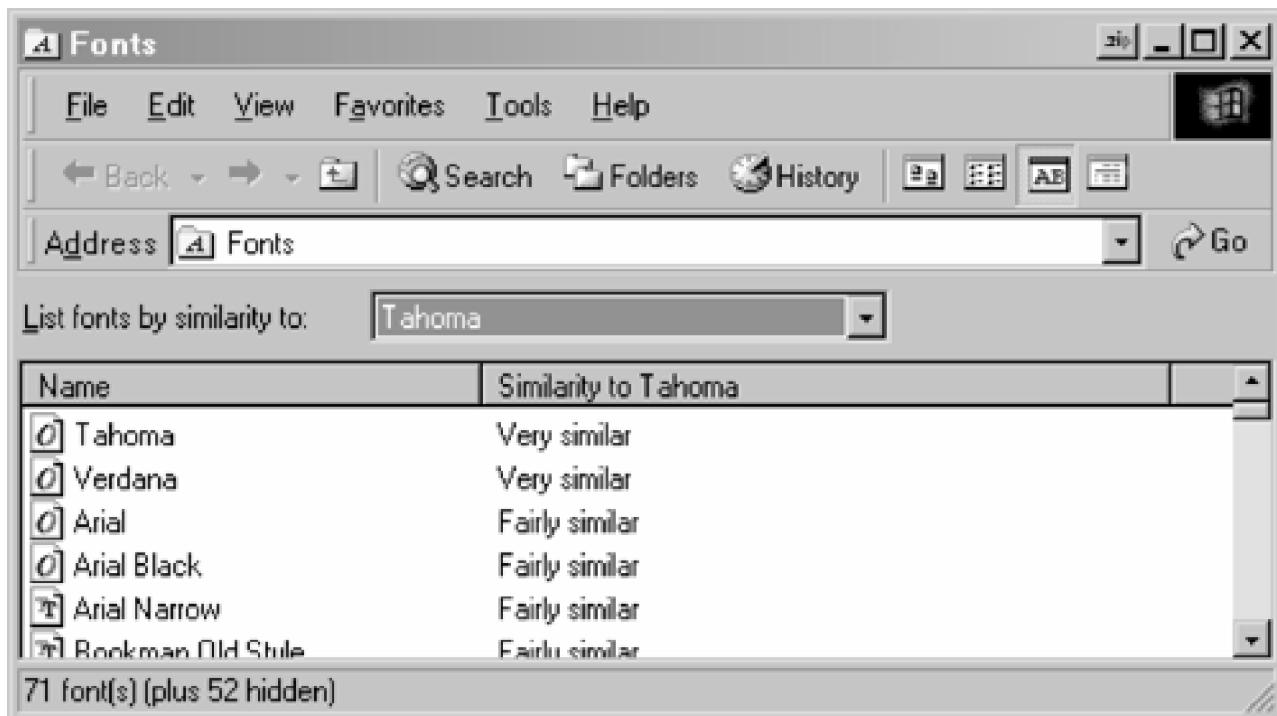
In the above PANOSE structure, a font is characterized by its family type, serif style, weight, proportion, contrast, stroke variation, arm style, letter form, middle line, and x-height. Compared with the LOGFONT structure, which has only two fields dealing with font appearance, lfWeight and lfPitchAndFamily, the PANOSE structure lays the foundation for more accurate font mapping. For example, PANOSE defines 14 different serif styles—cove, square, triangle, flared, rounded, etc.

PANOSE is a very compact and efficient way for classifying and matching fonts in a system. Every TrueType/OpenType font has a PANOSE structure. So two fonts can be matched with each other to find their “distance” in 10-dimensional font-appearance space. PANOSE 2.0 goes even further to characterize fonts, using 36 values which have more numerical sense. For details on the PANOSE Typeface Matching System, visit www.fonts.com/hp/panose/index.htm.

Although PANOSE typeface matching is much better than font matching based on LOGFONT and TEXTMETRIC structures, there is no direct GDI function to support it. The three functions, CreateFont, CreateFontIndirect, and CreateFont In directEx, do not include the PANOSE structure in defining a logical font.

Actually, the PANOSE typeface matching algorithm is implemented using a COM interface, IPANOSEMMapper, which is implemented by panmap.dll as one of many unknown DLLs on the system. It's used by the control panel font applet when asked to list fonts by similarity. [Figure 15-3](#) shows PANOSE typeface matching in action.

Figure 15-3. PANOSE typeface matching in the control panel.



[Figure 15-3](#) shows the display results when you turn on “List Fonts By Similarity” under the View menu. If you select “Tahoma” as the font to compare with, “Verdana” will be reported to be very similar to it, and “Arial” fairly similar, but “Courier New” is said to be not similar, and some other fonts are described as having no PANOSE information.

[Listing 15-2](#) shows a simple wrapper class for the IPANOSEMapper interface.

Listing 15-2 KFontMapper Class for IPANOSEMapper Interface

```
class KFontMapper
{
    IPANOSEMapper * m_pMapper;
    const PANOSE   * m_pFontList;
    int           m_nFontNo;

public:
    KFontMapper(void)
    {
        m_pMapper = NULL;
        m_pFontList = NULL;
        m_nFontNo = 0;

        CoInitialize(NULL);
        CoCreateInstance(CLSID_PANOSEMapper, NULL,
                        CLSCTX_INPROC_SERVER, IID_IPANOSEMapper,
                        (void **) & m_pMapper);
    }

    void SetFontList(const PANOSE * pFontList, int nFontNo)
    {
```

```
m_pFontList = pFontList;
m_nFontNo = nFontNo;
}

int PickFonts(const PANOSE * pTarget, unsigned short * pOrder,
              unsigned short * pScore, int nResult)
{
    m_pMapper->vPANRelaxThreshold();

    int rsIt = m_pMapper->unPANPickFonts(
        pOrder,          // order, best-to-worst
        pScore,          // match result
        (BYTE *) pTarget, // PANOSE number mapping to
        nResult,         // fonts to return
        (BYTE *) m_pFontList, // PANOSE array
        m_nFontNo,       // fonts to compare
        sizeof(PANOSE),
        pTarget->bFamilyType);

    m_pMapper->bPANRestoreThreshold();
    return rsIt;
}

~KFontMapper()
{
    if ( m_pMapper )
        m_pMapper->Release();
    CoUninitialize();
}
};
```

Besides constructor and destructor, KFontMapper class has two member functions. SetFontList sets an array of PANOSE structures corresponding to the available fonts. PickPonts takes a PANOSE number and tries to find a number of good matches for it. The result is returned in two arrays—one for the order of preferences and one for the distance between the source PANOSE structure and those picked.

To use the KFontMapper class, there are still two problems to solve. The first is to find the PANOSE number for the font that needs to be matched. The second is to generate a database of PANOSE numbers for all the available fonts on the system.

One way to solve the problem is keep the PANOSE number together with the LOGFONT structure in the document when a font choice is made. When a logical font is created and selected into a device context, GDI maps the logical font to a physical font that is currently installed on the system. GDI provides a function GetOutlineTextMetrics, which returns an OUTLINETEXTMETRIC structure for the physical font. Its otmPanoseNumber field is a PANOSE number.

The PANOSE number is saved by both the RTF (rich text format) file format and EMF (enhanced metafile) file format. The RTF format is used by rich edit control, the Windows help file source file, applications like Wordpad, or maybe even Microsoft Word. On MSDN, there is a knowledge-base article on a Word 97 defect. Although the RTF

format used by Word 97 stores PANOSE numbers with fonts, Word 97 ignores them when substituting for missing fonts.

If you browse through GDI's header file wingdi.h, searching for PANOSE, you will find PANOSE is used in the EXTLOGFONT structure. EXTLOGFONT is an extension of the LOGFONT structure with the full typeface name, style name, vendor identifier, PANOSE number, etc. So it captures both the logical font information and the physical font information. The strange thing is that not a single documented GDI function accepts or returns the EXTLOGFONT structure. Its only documented usage is in the EMREXTCREATEFONTINDIRECTW structure, the EMF record for logical font creation calls.

Generating a PANOSE number database for all available fonts may sound simple. [Chapter 14](#) discussed how to use EnumerateFontFamiliesEx or its variations to enumerate font families available on the system. For each font family, EnumerateFontFamiliesEx calls a user-supplied callback routine with a NEWTEXTMETRICEX structure that does have a PANOSE number field among other interesting stuff. But the problem is that these functions do not enumerate physical fonts, they enumerate font families, possibly for each character set they support. For example, for font family "Arial" there are four different fonts: "Arial," "Arial Bold," "Arial Bold Italic," and "Arial Italic." But EnumerateFontFamiliesEx only counts one of them, "Arial," 9 times for each of the scripts it supports (Western, Hebrew, Arabic, Greek, Turkish, Baltic, Central European, Cyrillic, and Vietnamese).

Clearly, "Arial" and "Arial Bold Italic" have quite a different appearance. But EnumerateFontFamiliesEx shows only one member of a family and hides the rest. If you're relying on it to generate a PANOSE database, it's only an incomplete database. Such a database is already stored in the Windows registry under:

`HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Shared Tools\Panose`

The list of currently installed physical fonts is stored in the Windows registry under

`SOFTWARE\Microsoft\Windows NT\CurrentVersion\Fonts`

By enumerating the physical font list, typeface names for fonts can be retrieved and filtered to get only TrueType/OpenType fonts. From a typeface name a logical font can be created, selected into a device context, and mapped to a physical font. If the matched physical font is a TrueType/OpenType font, its PANOSE number can be obtained from the GetOutlineTextMetrics function. [Listing 15-3](#) shows a routine to convert a typeface name to a PANOSE number and the name of the matched typeface. The fontenumeration part is covered in [Chapter 14](#).

Listing 15-3 From Typeface Name to PANOSE Number

```
// 'Arial Bold Italic' -> PANOSE
bool GetPANOSE(HDC hDC, const TCHAR * fullname, PANOSE * panose,
    TCHAR facename[])
{
    TCHAR name[MAX_PATH];

    // remove space before
    while (fullname[0]==' ')
        fullname ++;
```

```
_tcscpy(name, fullname);

// remove space after
for (int i=_tcslen(name)-1; (i>=0) && (name[i]== ' '); i--)
    name[i] = 0;

LOGFONT lf;
memset(&lf, 0, sizeof(lf));
lf.lfHeight = 100;
lf.lfCharSet = DEFAULT_CHARSET;
lf.lfWeight = FW_REGULAR;

if ( strstr(name, "Italic") )
    lf.lfItalic = TRUE;

if ( strstr(name, "Bold") )
    lf.lfWeight = FW_BOLD;

_tcscpy(lf.lfFaceName, name);

HFONT hFont = CreateFontIndirect(& lf);

if ( hFont==NULL )
    return false;

HGDIOBJ hOld = SelectObject(hDC, hFont);

OUTLINETEXTMETRIC otm[3]; // big enough

if ( GetOutlineTextMetrics(hDC, sizeof(otm), otm) )
{
    _tcscpy(facename, (const TCHAR *) otm +
        (int) otm[0].otmpFaceName);

    memcpy(panose, ((const BYTE *) & otm[0].otmPanoseNumber),
        sizeof(PANOSE));
}

else
    facename[0] = 0;

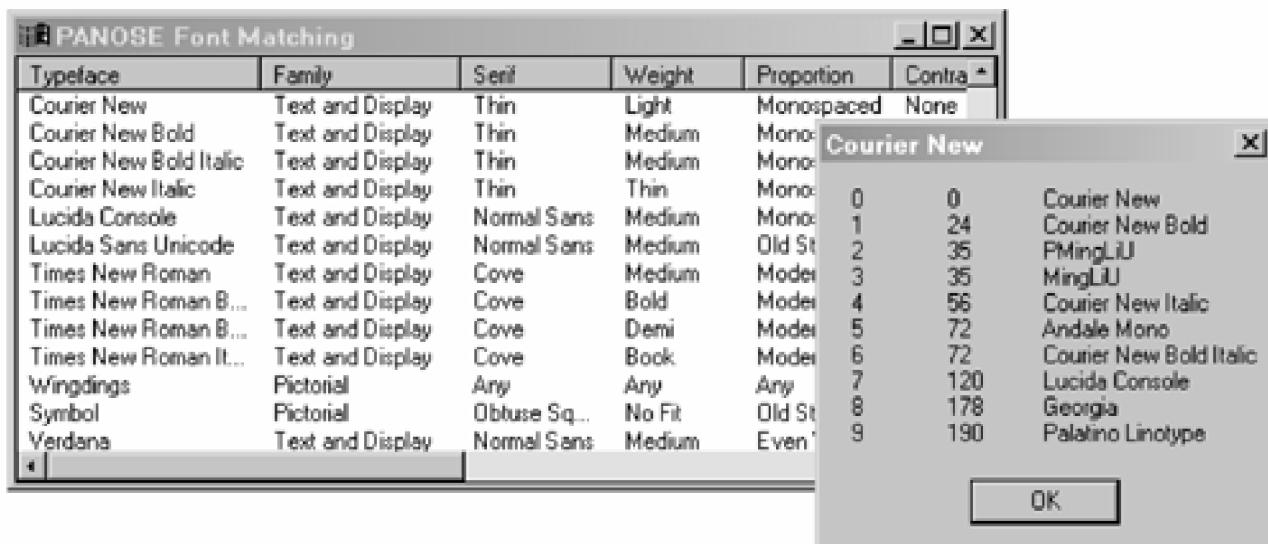
SelectObject(hDC, hOld);
DeleteObject(hFont);

return facename[0] != 0;
}
```

[Figure 15-4](#) shows the PANOSE Font Matching MDI window from this chapter's demo program “Font”. The main window of the MDI child window is a list box with all currently available fonts and their PANOSE numbers. Once you right-mouse-click on a typeface name, that font's PANOSE number is matched with the PANOSE numbers for all the

fonts. The message box on the right shows the font-matching result for "Courier New." Note that the font "MingLiU" does not have the proper PANOSE number, which generates some noise in matching. It should have been removed from the PANOSE-number array. The result shows "Courier New" is close to "Andale Mono," "Lucida Console," "Georgia," and "Palatino Linotype."

Figure 15-4. PANOSE font-matching demonstration.



If an application wants to improve its handling of missing fonts, there are several things to do. During document generation, the PANOSE number and other information regarding the matched physical font on the original machine should be recorded together with the LOGFONT structure. The EXTLOGFONT structure used by EMF is a good reference. When a document is transmitted to another machine, it's possible that certain special fonts may be missing. The application should check if the physical font matched by GDI is the same or close enough to the physical font on the original machine. This can be done using the exact matching of the fonts' full names or their PANOSE numbers. If it's decided that the original font is missing and GDI's default font mapping is not acceptable, the application should perform a PANOSE-based font mapping on its own. It can build its own PANOSE-number database for all available fonts, use the KFontMapper class or other PANOSE matching algorithm to find the best match, and then recreate logical fonts using their best matches on the local machine.

15.2 QUERYING LOGICAL FONT

After a logical font object is created and selected into a device context, GDI maps the logical font to a physical font currently available on the system. When this happens, a logical object is associated with a realized font. A realized font is not a physical font, it's just a particular instance of a physical font for a particular size, aspect ratio, rotation angle, feature simulation, etc. For example, for the "Times New Roman" physical font, during a particular time, it may have several instantiations, one for a 12-point logical font on a 96-dpi screen display, another for a 24-point logical font on a 300-dpi printer device rotated by 30 degrees.

Instantiation of a physical font is a complex, time-consuming, and memory-intensive task. [Chapter 14](#) explains the complexity of TrueType fonts. Raw data in TrueType is hard to parse and search in its original form. To find the exact glyph for a character in a particular character set, the character needs to be translated to UNICODE and then mapped to glyph index by searching a TrueType mapping table. To generate glyphs for a particular size, the glyph outline needs to be scaled to the right size, grid-fitted using glyph fitting instructions, and then converted to bitmap. A simple naïve implementation, in which each drawing of a character starts a mapping from character code to glyph index, from glyph outline to rendered glyph bitmap, will cause a huge system performance problem. In reality, the graphics engine maintains a complicated data structure between logical font objects and the raw physical font files.

Under Windows NT/2000, each physical font currently being used has a corresponding data structure (PFF) kept in kernel address space, which maintains a linked list of font instantiations (FONTOBJ). Each FONTOBJ keeps a cache of rendered glyphs so that they can be reused. Each logical font object has a pointer to an undocumented GDI object, a PFE object, which maintains links to the corresponding PFF objects. The complicated data structure makes sure font instantiation can be efficiently cached and reused on the system level while still allowing GDI logical fonts to be created, used, and deleted freely. For details on the font-related internal graphics engine data structure, refer to [Chapter 3](#).

Once a logical font is selected into a device context, several functions can be used to query the details of the physical font mapped and the metrics information about the current instance of the physical font.

```
int GetTextFace(HDC hDC, int nCount, LPSTR lpFaceName);
DWORD GetFontLanguageInfo(HDC hDC);
int GetTextCharSet(HDC hDC);
int GetTextCharSetInfo(HDC hDC, LPFONTSIGNATURE lpSig, DWORD dwFlags);

BOOL GetTextMetrics(HDC HDC, LPTEXTMETRIC lptm);
UINT GetOutlineTextMetrics(HDC hDC, UINT cbData,
    LPOUTLINETEXTMETRIC lpOTM);
```

GetTextFace returns the typeface name of the physical font that is mapped to the current logical font in the device context, which may be different from the typeface name used in creating the logical font.

GetFontLanguageInfo returns glyph-related information for the current font selected in a device context, such as whether double-byte glyphs are provided, whether glyphs have diacritics, whether kerning table is provided, etc. Information provided by this function is useful for advanced text output; for example, using GetCharacterPlacement or ExtTextOut to deal with glyphs directly. GetFontLanguageInfo returns a combination of several flags, of which

FLI_GLYPHS (0x40000) and GCP_USER KERNING (0x0008) are normally seen flags. FLI_GLYPHS indicates the font has extra glyphs not normally accessible through characters in various code pages. GCP_USERKERNING indicates the kerning table is available for the font.

When the Arabic or the Hebrew language pack is installed, for several Windows TrueType fonts, GetFontLanguageInfo returns GCP_REORDER, GCP_GLYPH SHARE, GCP_LIGATE, GCP_DIACRITIC, and GCP_KASHIDA flags. For example, “Arial,” “Lucida Sans Unicode,” “Tahoma,” and “Andalus” support GCP_REORDER, but not “Times New Roman.”

GetTextCharset returns the character-set identifier for the font that is currently selected into a device context. It can be used to verify whether the physical font supports the character set required in logical font. For example, if HANGUL_CHARSET is used in creating the logical font, but your system does not have any Korean font installed, GDI may map the change character set to the default system character set, and GetTextCharSet may return ANSI_CHARSET.

GetTextCharsetInfo returns a FONTSIGNATURE structure that provides information on a TrueType/OpenType font's UNICODE subrange and code page supported. The first four DWORDs in a FONTSIGNATURE form a 128-bit Unicode subset bitfield (USB); the remaining two DWORDs form a 64-bit code-page bitfield (CPB). For example, bit 9 of USB indicates whether Cyrillic glyphs are supported, bit 16 of CPB indicates whether code page 874 (Thai) is supported. For “Tahoma” font, 12 bits in CPB are turned on, so it provides glyphs for 12 different code pages. The FONT SIGNATURE structure is generated from a TrueType font's “OS/2” table.

GetTextMetrics and GetOutlineTextMetrics return important metrics information about the current instantiation of a physical font. GetTextMetrics returns a TEXT METRIC structure that was originally designed for bitmap and vector fonts but nevertheless still works for TrueType/OpenType fonts. For TrueType/OpenType fonts, GetOutlineTextMetrics returns an OUTLINETEXTMETRIC structure, an expanded version of TEXTMETRIC, which has more information.

Metrics for Bitmap and Vector Fonts

The TEXTMETRIC structure is evolved from the font header in a Windows bitmap or vector font, although it works for TrueType/OpenType fonts too. To get a TEXTMETRIC structure for the current font selected in a device context, call GetTextMetrics with a device context handle and a pointer to a TEXTMETRIC structure.

The TEXTMETRIC structure is as follows:

```
typedef struct tagTEXTMETRIC
{
    LONG      tmHeight;
    LONG      tmAscent;
    LONG      tmDescent;
    LONG      tmInternalLeading;
    LONG      tmExternalLeading;
    LONG      tmAveCharWidth;
    LONG      tmMaxCharWidth;
    LONG      tmWeight;
    LONG      tmOverhang;
    LONG      tmDigitizedAspectX;
```

```
LONG    tmDigitizedAspectY;
BCHAR   tmFirstChar;
BCHAR   tmLastChar;
BCHAR   tmDefaultChar;
BCHAR   tmBreakChar;
BYTE    tmItalic;
BYTE    tmUnderlined;
BYTE    tmStruckOut;
BYTE    tmPitchAndFamily;
BYTE    tmCharSet;
} TEXTMETRIC;
```

The tmAscent field specifies the ascent (height of character above baseline), tm Descent specifies the descent (height of character below baseline), and tmHeight is the total character height, which is the sum of tmAscent and tmDescent. Refer back to [Figure 15-1](#) for an illustration.

The tmInternalLeading field specifies the amount of whitespace below the ascent line that is normally used for accent marks and diacritical characters, and tmExternalLeading specifies the suggested extra space between lines.

The difference between the tmHeight and the tmInternalLeading corresponds to font point size, which is a common measure of character size. For example, a 36-point font for a 96-dpi display device context in MM_TEXT mode would set LOGFONT's lf Height field to be -48 (36*96/72). When a logical font is created from such a LOGFONT structure and selected into the display device context, the TEXTMETRIC structure returned by GetTextMetrics will have a tmHeight equal 55 and an internal leading equal 7. The difference between them is exactly 48, the absolute value of LOGFONT's lfHeight field. For bitmap fonts, if scaling is disabled using the PROOF_QUALITY flag in LOGFONT's lfQuality field, GDI may not be able to find the exact font resource for the size required. For example, for a 10-point font using "Terminal" as a typeface, LOGFONT's lfHeight is -13, but TEXTMETRIC's tmHeight may only be 12, and internal leading is 0. For bitmap fonts, it's normal for a smaller font size to be used when the exact match can't be found.

The sum of tmHeight and tmExternalLeading is the suggested line space between a line and the next line. Given both values, the amount of space for n lines of text can be calculated using $tmHeight \cdot n + tmExternalLeading \cdot (n - 1)$, because no external leading space is needed before the first line and after the last line. For example, a 36-point font for a 96-dpi display device context in MM_TEXT mode would have a tmHeight of 55 and a tmExternalLeading of 2, so the line space is 57 units, or 42.75 points. Note that, for the same point size, different fonts may have different tmHeight and tmExternalLeading, or their sum.

The tmAveCharWidth is the average width of characters in the font. Microsoft documentation states that the average width is usually the width of lower-case letter "x." For a TrueType/OpenType font, the average character width is more precisely defined as a weighted sum of lower-case letters "a" through "z" in the Latin alphabet plus the space character. For a monospace font, all characters have the same width, so the average character width can be used to calculate the length of a character string. The average character width can also be used to select the closest font to replace a missing font. The tmMaxCharWidth specifies the maximum width for characters in a font.

The tmWeight field specifies the required font weight, which has the same value as the lfWeight field in a LOGFONT structure. Note that each font resource for a bitmap or vector font, or a physical font for TrueType fonts, has a fixed weight. A single TrueType typeface normally has four physical font files, for regular, italic, bold, and bold italic. The first two normally have weight 400 (FW_NORMAL); the remaining two normally have weight 700 (FW_BOLD). GDI tries to find the best match between the required font weight and the available font weight. When the exact match can't be found, GDI uses a simple technique to simulate the required font weight.

The tmOverhang field specifies the extra horizontal space needed for GDI to synthesize font features. If the required weight is larger than the available font's weight, GDI overstrikes characters to make them appear to have a higher weight; if italic is required but a physical italic font is not available, GDI shears glyphs to make them look italic. In either case, extra *overhang* space is needed at the end of a string. The tm Overhang field allows the application to precisely calculate the horizontal space needed by a string of characters to make sure it will not get out of boundary, or get clipped. In reality, bitmap and vector fonts are found to provide nonzero overhang values, while TrueType/OpenType fonts always return a zero overhang. For example, italic "Wing dings" at 72 points, which only has a regular physical font, returns zero tmOverhang.

The tmFirstChar field specifies the value of the first character which the font has a glyph for; the tmLastChar field specifies the last. Note the type for these two fields is BCHAR, which is defined as WCHAR for UNICODE and BYTE otherwise. For bitmap and vector fonts, glyphs are provided for all characters within the range of tmFirstChar and tmLastChar. For TrueType/OpenType fonts, the single-byte version of tmFirstChar and tmLastChar fields can't represent the true range of characters having glyphs in the font, while the UNICODE version does not mean all the characters between tmFirstChar and tmLastChar have glyphs.

The tmDefaultChar field specifies the default character to use if a certain character does not have a glyph in the font. The normal glyph for a default character is a rectangular box. For a TrueType font, it's normally the first glyph in the glyph description table with glyph index 0. The tmBreakChar field specifies the character that will be used to define word breaks for text justification.

The next three fields repeat fields by similar names in LOGFONT; tmItalic is non-zero if italic is required, tmUnderline is nonzero if underlining is required, and tmStruckOut is nonzero if striking out is required. Note that they represent what's required in the logical font object, not what's in the physical font. GDI is responsible for synthesizing features if not present in the physical font.

The tmPitchAndFamily field specifies the pitch, technology, and family of a physical font. The high nibble (4 bits) of this field is the same as a LOGFONT's lfPitchAndFamily's high-order nibble. The lower nibble has four independent bits, which makes it different from lfPitchAndFamily's lower nibble.

- TMPF_FIXED_PITCH (0x01) is set if the font is a variable-pitch font. Note that the logic is unbelievably reversed here. Why don't they change the name to TMPF_PROP_PITCH?
- TMPF_VECTOR (0x02) is set if the font is an outline font, which only means it's not a bitmap font.
- TMPF_TRUETYPE (0x04) is set if the font is a TrueType/OpenType font.
- TMPF_DEVICE (0x08) is set if the font is a device.

The last field, tmCharSet, specifies the character set of the logical font, which will be the same as LOGFONT's lfCharSet field if it's not DEFAULT_CHARSET, and the same value is returned by GetTextCharSet. A TrueType/OpenType font normally supports multiple character sets, which can be queried by GetTextCharSetInfo.

Metrics for TrueType/OpenType Fonts

A TrueType/OpenType font has much more metrics information than what can be captured in the TEXTMETRIC structure, for which the OUTLINETEXTMETRIC structure is designed. The qualifier "outline" here may be a bit confusing, because the structure does not apply to vector fonts, only TrueType/OpenType fonts, or any device fonts for which the font driver can provide an OUTLINTETEXTMETRIC structure.

Here is the OUTLINETEXTMETRIC structure:

```
typedef struct _OUTLINETEXTMETRIC {  
    UINT      otmSize;  
    TEXTMETRIC otmTextMetrics;  
    BYTE      otmFiller;  
    PANOSE    otmPanoseNumber;  
    UINT      otmfsSelection;  
    UINT      otmfsType;  
    int       otmsCharSlopeRise;  
    int       otmsCharSlopeRun;  
    int       otmItalicAngle;  
    UINT      otmEMSSquare;  
    int       otmAscent;  
    int       otmDescent;  
    UINT      otmLineGap;  
    UINT      otmsCapEmHeight;  
    UINT      otmsXHeight;  
    RECT     otmrcFontBox;  
    int       otmMacAscent;  
    int       otmMacDescent;  
    UINT      otmMacLineGap;  
    UINT      otmusMinimumPPEM;  
    POINT    otmptSubscriptSize;  
    POINT    otmptSubscriptOffset;  
    POINT    otmptSuperscriptSize;  
    POINT    otmptSuperscriptOffset;  
    UINT      otmsStrikeoutSize;  
    int       otmsStrikeoutPosition;  
    int       otmsUnderscoreSize;  
    int       otmsUnderscorePosition;  
    PSTR     otmpFamilyName;  
    PSTR     otmpFaceName;  
    PSTR     otmpStyleName;  
    PSTR     otmpFullName;  
} OUTLINETEXTMETRIC;
```

After a logical font handle is selected into a device context, assuming that a TrueType/OpenType physical font is mapped to it, you can call GetOutlineTextMetrics to ask GDI to fill an OUTLINETEXTMETRIC structure.

Although the OUTLINETEXTMETRIC structure does not look so complicated, except that it does have many fields, it's a very deceiving data structure. Microsoft documentation does not explain it clearly. First, OUTLINETEXTMETRIC is a variable-size structure, for its last four fields are offsets to strings, which are normally attached to the data block after its last public field. Now you can see the second tricky thing about it: The last four fields are not pointers as the declaration indicates, but offsets relative to the starting of the data block. The first field is named otmSize, so you would have guessed that you have to set it to the size of the structure before calling GetOutlineTextMetrics. Actually, it does not need to be set to the data block size, because the second parameter to

GetOutlineTextMetrics is a data block size.

Because of the variable-size nature of OUTLINETEXTMETRIC, the proper way to use GetOutlineTextMetrics is to call it twice, first to get the actual size, and next to query the real data after the required amount of memory is allocated. On the CD, there is a wrapping class, KOutlineTextMetric, for querying the OUTLINETEXTMETRIC structure. Here is its class declaration:

```
class KOutlineTextMetric
{
public:
    OUTLINETEXTMETRIC * m_pOtm;
    KOutlineTextMetric(HDC hDC);
    ~KOutlineTextMetric();
};
```

The constructor of the KOutlineTextMetric class gets an OUTLINETEXTMETRIC into a memory block that is allocated from the heap. The destructor deletes it when an instance of the class is out of scope.

If you're reading the code, you may have noticed a strange assert on the offset of the otmFiller field. This is for the trickiest part of this structure: OUTLINETEXTMETRIC must be DWORD aligned. This restriction is not documented and not enforced by Windows header files. It took me several days to figure out why OUTLINE TEXTMETRIC returned by GetOutlineTextMetrics does not match its declaration. Finally, I figured it out by looking at the binary dump of the data.

The second field of OUTLINETEXTMETRIC is a TEXTMETRIC structure, which is $4n + 1$ bytes in size. The original designer of the data structure added a single-byte filler (otmFiller) to make sure the PANOSE structure following it would be on the WORD boundary. Note that this structure was first designed in the Windows 3.1 Win16 API, when TrueType fonts were first introduced to Windows. It seems that when Microsoft's GDI source code is compiled, structure member alignment is set at 4 bytes. The embedded TEXTMETRIC structure is now padded to be a multiple of DWORDs (the largest of its members), which adds three bytes before the otmFiller field. Now otmFiller is on the DWORD boundary; the PANOSE structure follows without any padding, because all its members are BYTES. But the PANOSE structure is 10 bytes long, and the next DWORD size otmfsSelection field needs to start a DWORD boundary, so two extra bytes are added.

Normally, when a special structure alignment is needed, the Windows header file should enforce it by adding "#pragma pack" to overwrite what's set in the user application's project setting. This makes sure that the same exact Win32 API is presented to the application regardless of the application's project setting. But for OUTLINE TEXT METRIC, wingdi.h fails to do so. The header file at the time of writing uses DWORD packing when macro_MAC is defined. If your project is using 1- or 2-byte structure alignment, be sure to switch alignment to 4 bytes if you want to use OUTLINETEXTMETRIC using:

```
#pragma pack(push, 4)
#include <windows.h>
#pragma pack(pop)
```

The first field otmSize specifies the size of the complete structure, including embedded strings. This is followed by a TEXTMETRIC structure, described above, which is exactly the same as what's returned by GetTextMetrics. The third field otmFiller was meant to align the PANOSE structure following it to a WORD boundary. But it seems that the

GDI source code is compiled with a 4-byte or 8-byte structure alignment, which adds three hidden padding bytes after otmTextMetrics and before otmFiller. The otmPanoseNumber field stores the PANOSE number for the physical font. After that there are two hidden padding bytes to make sure the next field is on DWORD boundary.

Field otmfsSelection specifies the nature of font patterns, using a combination of 6 bits. Bit 0 (0x01) is on for italic font, bit 1 (0x02) is for underscore, bit 2 (0x04) for negative, bit 3 (0x08) for outline, bit 4 (0x10) for strikeout, bit 5 (0x20) for bold, and bit 6 (0x40) for regular. A TrueType font has an attribute by a similar name, which is defined to tell the original design of a physical font. For example, if bit 6 and bit 5 are both on, it means the physical font is actually a bold font—that is, bold is not a simulated feature. GDI seems to use this field in a different way; otmfsSelection represents features of the logical font, instead of the physical font. So you can't use otmfsSelection to tell if a physical font is a bold font. To check the features of the physical font, the PANOSE number is a reliable source; for example, its bLetterform field can tell whether a physical font is italic (oblique).

Field otmfsType specifies font-embedding licensing rights for the font. Refer to [Section 14.5](#) for a detailed explanation.

The next three fields are for text caret display. For upright fonts, a text caret is a thin vertical line that indicates the position to enter the next character. For italic fonts, a text caret in a more professional word processor should follow the italic angle. Field otmtitalicAngle specifies the italic angle of a font, in tenths of a degree counterclockwise from the y-axis. For upright fonts, otmtitalicAngle is 0; for italic fonts, otmtitalic Angle is normally a negative number. The ratio between otmsCharSlopeRise and otms Char SlopeRun defines the slope of the caret. For upright fonts, otmsCharSlope Rise is 1 and otmsCharSlopeRun is 0, so the caret is a vertical line. For example, the italic “Times New Roman” font has otmtitalicAngle being -164, otmCharSlopeRise being 24, and otmCharSlopeRun being 7. Tan(16.4 degrees) is 0.2943, which is very close to 7/24 (0.2917). Note that these three features are for the physical fonts. GDI simulation of an italic font does not change them. For example, try italic “Tahoma,” which does not have an italic physical font; Wordpad will display an upright caret, but Word is smart enough to display a slanted caret.

The otmEMSquare field specifies the number of units defining the dimension of the em-square for the physical font. Em-square is the reference grid on which glyphs are designed. All points in a glyph description are represented using integer coordinates on the em-square, so a larger em-square normally means better glyph quality. This field is normally used by an application to determine how to set up logical coordinate space to get high-precision font metrics information. Refer to [Chapter 14](#) for details about TrueType glyph description.

The otmAscent field specifies the typographic ascent for the font, otmDescent is the typographic descent, and otmLineGap is the typographic line spacing. GDI has its own way of defining ascent, descent, internal leading, and external leading, which may match typographic measures in some cases or may be different in others. Another difference is that otmDescent is normally a negative number, because descent is below the baseline, while Windows always uses absolute distance value. There is another set of line metrics named otmMacAscent, otmMacDescent, and otmMacLineGap, which is Macintosh's font vertical metrics.

Fields otmCapEmHeight and otmXHeight are said to be unsupported in Microsoft documentation; otmCapEmHeight seems to be the height of a capital letter without descender, and otmXHeight seems to be the height of lower-case “x.”

Field otmrcFontBox specifies the bounding box for all the glyphs in the font, relative to character origin. Value otmrcFontBox.left is normally a negative value, corresponding to smallest A-width; otmrcFontBox.bottom is normally a negative value corresponding to the largest descender.

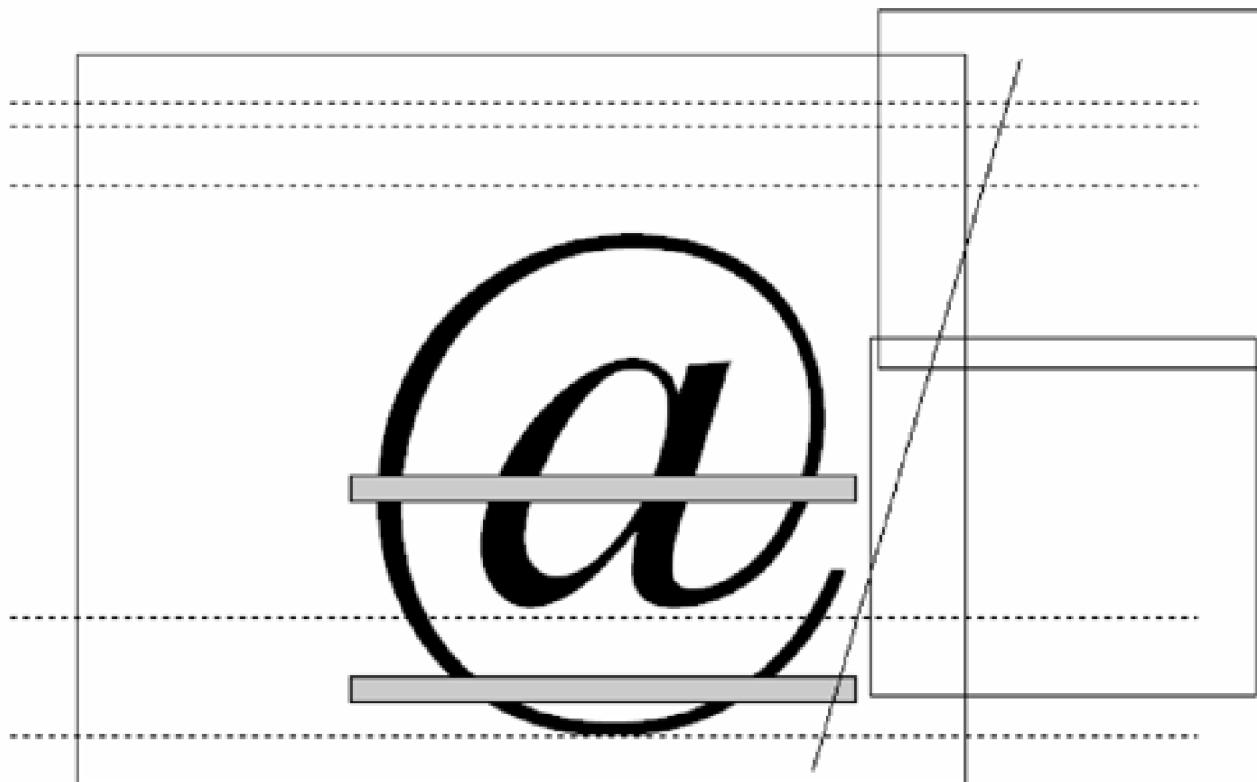
Field otmsMinimumPPEM specifies the minimum pixel size to which the em-square can be scaled down according to the original design of the font. This is an indication of how well the glyph-fitting instructions are designed to make sure nice bitmaps can still be generated at small pixel size. A normal value is 9 pixels or 12 pixels.

Fields `otmptSubscriptSize` and `otmptSubscriptOffset` specify the size and position of subscripts for this font, measured from character origin. Likewise, fields `otmptSuperscriptSize` and `otmptSuperscriptOffset` specify the size and position of superscripts for this font.

Fields `otmsStrikeoutSize` and `otmsStrikeoutPosition` specify the width of the strike-out stroke and the horizontal position relative to the baseline. Likewise, fields `otmsUnderscoreSize` and `otmsUnderscorePosition` specify the width of the underscore and the horizontal position relative to the baseline.

[Figure 15-5](#) illustrates several new measures provided by the `OUTLINETEXTMETRIC` structure—namely, bounding box, subscription, superscription, underline, strikeout, and character slope. The five dotted lines are the external leading line, ascent line, internal leading line, baseline, and descent line. The big box is the bounding box of the font, which seems to be too big. The two gray bars are for strikeout and underscore simulation. The two boxes on the right represent origins and sizes for superscript and subscript characters. One slanted line represents the slope of character which the character caret should follow.

Figure 15-5. Font metrics information in OUTLINETEXTMETRIC.



The last four fields of `OUTLINETEXTMETRIC` structures are offsets to the start of the structure for the physical font's family name, face name, style name, and full name. An example will make it clear. For the "Times New Roman" italic bold font, there is a corresponding physical font, so the last four fields for its `OUTLINETEXTMETRIC` structure are offsets to "Times New Roman," "Times New Roman Bold Italic," "Bold Italic," and "Monotype Times New Roman Bold Italic Version 2–76 (Microsoft)."

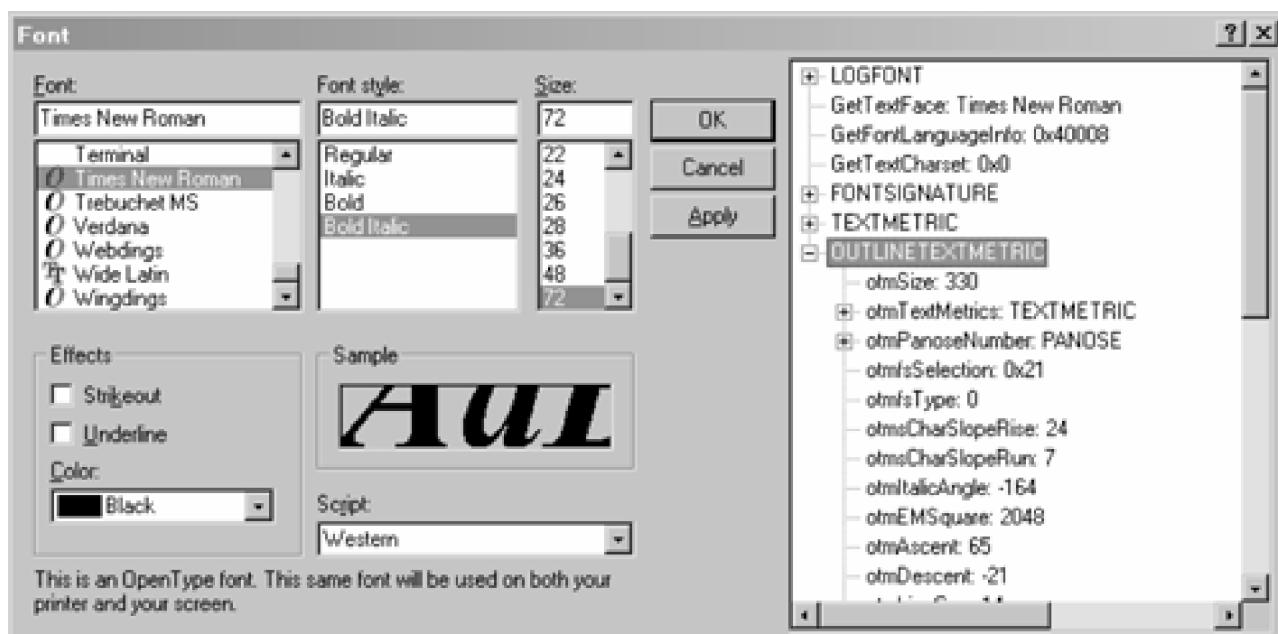
Explore LOGFONT and Font Metrics

To help understand the relationship between logical font as specified in a LOGFONT structure and the corresponding physical font, as can be queried using several GDI functions described above, the “Font” program for this chapter customizes the standard font dialog box to display all related information.

The Win32 API provides a ChooseFont function to display a standard font dialog box, allowing the user to select a logical font. The ChooseFont functions allow the application to provide a hook function which will be called to override message handling in the common dialog box for fonts. By providing a callback function, information regarding the current LOGFONT structure and metrics about the corresponding instantiation of a physical font can be displayed. In the code that comes on the CD, the font dialog box is expanded on the right to display a TreeView window for all related information.

[Figure 15-6](#) illustrates the customized font dialog box. On the left-hand side, which is the original font dialog box, a 72-point bold italic “Times New Roman” font has been chosen. The Treeview child window on the right showing LOGFONT results from GetTextFace, GetFontLanguageInfo, GetTextCharset, GetTextCharsetInfo, GetTextMetrics, and GetOutlineTextMetrics. The OUTLINETEXTMETRIC structure is expanded in [Figure 15-6](#), which shows an embedded TEXTMETRIC and PANOSE, and the first few fields within it. You can change the selections on the left and hit the “Apply” button to synchronize the selection with the result displayed in the TreeView window.

Figure 15-6. Customized font dialog box for displaying font metrics.



Font Metrics Accuracy

When creating a logical font, the height and width fields or parameters determine the size of the font. When a physical font is mapped to a logical font, its metrics information in the em-square is scaled to the size of the logical font and returned in either the TEXTMETRIC or OUTLINETEXTMETRIC structure. When a logical font is selected into a device context, its height and width are interpreted as logical coordinates in that device context, so the metrics information in the TEXTMETRIC or OUTLINETEXTMETRIC structure is all in that device context's logical coordinate space.

Metrics information is used to break a long string of text into multiple lines to form a paragraph, and to break lines of

text into pages. If document printing is supported, it's essential that line breaks and page breaks displayed on the screen match exactly with line breaks and page breaks when the document is printed on different printers. It's also critical to keep line breaks and page breaks constant when the screen display is zoomed to different scales. This is the basic requirement of WYSIWYG (what-you-see-is-what-you-get).

Suppose you're writing a simple text-processing software, which uses the same font to draw lines of text. The problem now is that, given the point size and page height, the number of lines per page needs to be calculated. Here is an implementation of such a function:

```
int LinesPerPage(HDC hDC, int nPointSize, int nPageHeight)
{
    KLogFont lf(-PointSizetoLogical(hDC, nPointSize), "Times New Roman");

    HFONT hFont = lf.CreateFont();
    HGDIOBJ hOld = SelectObject(hDC, hFont);

    TEXTMETRIC tm;
    GetTextMetrics(hDC, &tm);

    int linespace = tm.tmHeight + tm.tmExternalLeading;
    SelectObject(hDC, hOld);
    DeleteObject(hFont);

    POINT P[2] = { 0, 0, 0, nPageHeight }; // device coordinate
    DPtoLP(hDC, P, 2); // logical coordinate
    nPageHeight = abs(P[1].y-P[0].y);

    return (nPageHeight + tm.tmExternalLeading) / linespace;
}
```

The LinesPerPage function accepts a device context handle, a font point size, and a page height in device coordinate space (pixels). It converts point size to logical coordinate space, creates a logical font, selects it into the device context, and then queries for the font's vertical metrics. The distance between two lines of text can be calculated using height plus external leading. After converting the page height (assuming margins are removed) into logical coordinates, the number of lines per page can be calculated using (page height + external leading) / line space. External leading is added to the page height because it's the extra space between lines, so for N lines, only $(N - 1)$ external leading is needed.

The question now is, how accurate is the computation? [Table 15-1](#) shows some data, assuming the page height is 10 inches (11-inch letter-size paper with half-inch top and bottom margins).

According to [Table 15-1](#), by changing the screen dpi or mapping mode, or by switching from display device context to printer device context, based on font metrics returned in the TEXTMETRIC structure, there are three answers to the simple question of how many lines can fit into a 10-inch vertical space: 59, 60, and 61. Note that the function LinesPerPage truncates a fractional result to an integer, because an incomplete line can't be displayed. Even if rounding were used, there would still be three results: 60, 61, and 62. It's interesting to note that the same standard mapping mode (MM_LOENGLISH or MM_TWIPS) can generate different results on the same computer by merely switching from small font mode (96-dpi) to large font mode (120-dpi).

Table 15-1. Page-Break Computation

Device	Mapping Mode	Logical Page Height	External Leading	Logical Line Space	Lines per Page
96-dpi screen	MM_TEXT	960	1		16 60.0625 (60)
	MM_LOENGLISH	1050	1		17 61.8235 (61)
	MM_TWIPS	15118	9	245	61.74286 (61)
120-dpi screen	MM_TEXT	1200	1	20	60.05 (60)
	MM_LOENGLISH	1312	1	22	59.6818 (59)
	MM_TWIPS	18897	11	310	60.9935 (60)
360-dpi printer	MM_TEXT	3600	2	59	61.0508 (61)
600-dpi printer	MM_TEXT	6000	4	98	61.2653 (61)

The fundamental problem with this anti-WYSIWYG behavior is that errors are introduced when scaling metrics information in a physical font to logical coordinates used in a device context, especially when logical coordinates are represented using integers in the Windows GDI.

Recall that a font's point size is a measure of its ascent + descent - internal leading. For a TrueType physical font, ascent + descent - internal leading is the same as its em-square size. This simple formula makes scaling coordinates in the glyph description to the required point size really simple. For a logical font object created by Create Font Indirect, if the height is given as font height (which matches ascent + descent - internal leading), and the width is given as font width, point (x, y) in the original glyph description is scaled to:

$$(x * \text{width}/\text{emsquaresize}, y * \text{height}/\text{emsquaresize})$$

If the width in LOGFONT is zero, vertical and horizontal axes will be scaled to the same scale. Besides points in the glyph description, size information like ascent, descent, or strikeout size is scaled in the same way. Rounding is used to convert fractional results to integers.

For example, the "Times New Roman" font has an em-square size of 2048, the ascent being 1825, descent 443, and external leading 87. For a 10-point font on a 96-dpi screen, LOGFONT's height will be -13; GetTextMetrics will set tmAscent to 12, tm Descent to 3, and tmExternalLeading to 1. The accurate values here are $11 + 1197/2048$ ($1825*13/2048$), $2 + 1663/2048$ ($443*13/2048$), and $1131/2048$ ($87*13/2048$). When they are rounded to the closest integer values, the errors are 0.4155, 0.188, and 0.4478. When errors accumulate over a large number of lines, it could be the difference between fitting 59 lines into a page or 60 or even 61 lines into a page. Although error values are always less than 1, when compared with the small number they are coming from, the relative error is quite high. In this example, the integer value of the line space (tmAscent + tmDescent + tmExternalLeading) is 6.5% off its accurate value.

Logical Coordinate Space Resolution vs. Accuracy

Given a device context, there are two possible ways to get more accurate font metrics information. One way would be to set up a high-resolution logical coordinate space, because text metrics information in TEXTMETRIC and OUTLINETEXTMETRIC structures is in logical coordinate space. For example, if you change the mapping mode of a 96-dpi screen device context to MM_ANISOTROPIC, the window extent to (100, 100), and the viewport extent to (1,

1), now its logical coordinate space is 9600 dpi. That is, moving 9600 units in logical coordinate space should be one logical inch on the screen. The 9600-dpi logical coordinate space should be much more precise than the 600-dpi printer device context commonly seen today. Surprisingly, the reality is not what you may expect. Increasing the resolution of logical coordinate space does not improve text metrics accuracy. It seems that the graphics engine is making a huge mistake by scaling metrics information to device coordinates first and then to the logical coordinate system. Although it's possible that the results for the first-step scaling are stored in some fixed-point format, when scaled to a high-resolution logical coordinate space, not much improvement in accuracy is observed.

Here is a little routine to test the relationship between logical coordinate space resolution vs. text metrics accuracy.

```
void Test_LC(void)
{
    HDC hDC = GetDC(NULL);
    SetMapMode(hDC, MM_ANISOTROPIC);
    SetViewportExtEx(hDC, 1, 1, NULL);

    TCHAR mess[MAX_PATH];
    mess[0] = 0;

    for (int i=1; i<=64; i*=2)
    {
        SetWindowExtEx(hDC, i, i, NULL);

        KLogFont lf(-PointSizetoLogical(hDC, 24), "Times New Roman");
        HFONT hFont = lf.CreateFont();
        SelectObject(hDC, hFont);

        TEXTMETRIC tm;
        GetTextMetrics(hDC, &tm);
        wsprintf(mess + _tcslen(mess), "%d:1 lfHeight=%d, tmHeight=%d\n",
                 i, lf.m_lf.lfHeight, tm.tmHeight);

        SelectObject(hDC, GetStockObject(ANSI_VAR_FONT));
        DeleteObject(hFont);
    }
    ReleaseDC(NULL, hDC);
    MessageBox(NULL, mess, "LCS vs. TEXTMETRIC", MB_OK);
}
```

Each iteration sets up a higher dpi logical coordinate system than the previous one, creates the same 24-point font, selects it into the device context, and queries for the height of the font. On a 96-dpi screen display, lfHeight changes from -32, -64, up to -2048, and tmHeight changes from 36, 72, up to 2304. The value of tmHeight just gets doubled every time. How do you know it's not the right result? Try to create a $24 \times 64 = 1536$ -point font in the customized font dialog box shown in [Figure 15-6](#). lfHeight is the same -2048, but tmHeight is now 2268, not 2304.

Increasing the resolution of the logical coordinate system does not improve text metrics accuracy, at least on the current implementation of GDI.

Font Point Size vs. Accuracy

The second way to increase logical font size, so as to improve font metrics accuracy, is plain simple: increase the font's point size.

Here is a similar routine to test the relationship between font size and font metrics accuracy.

```
void Test_Point(void)
{
    HDC hDC = GetDC(NULL);
    TCHAR mess[MAX_PATH*2];
    mess[0] = 0;

    for (int I=1; I<=64; I*=2)
    {
        KLogFont lf(-PointSizeToLogical(hDC, 24*I), "Times New Roman");
        HFONT hFont = lf.CreateFont();
        SelectObject(hDC, hFont);

        TEXTMETRIC tm;
        GetTextMetrics(hDC, &tm);
        wsprintf(mess + _tcslen(mess),
                 "%d point lfHeight=%d, tmHeight=%d\n",
                 24*I, lf.m_lf.lfHeight, tm.tmHeight);

        SelectObject(hDC, GetStockObject(ANSI_VAR_FONT));
        DeleteObject(hFont);
    }
    ReleaseDC(NULL, hDC);
    MessageBox(NULL, mess, "Point Size vs. TEXTMETRIC", MB_OK);
}
```

When font point size is increased to 1536 points, lfHeight is -2048, and tm Height returns 2268. Actually 2048 is the em-square size of the "Times New Roman" font we're using, and 2268 is the actual font height stored in the physical TrueType font metrics table.

The painful conclusion is that to get the most accurate font metrics information, you need to create a logical font whose height is the negation of the font's em-square size. The core Windows font set has an em-square size of 2048; 4096 is also a common value. The maximum em-square size that is allowed by the TrueType font specification is 16384.

Here is a routine to create a reference font for an existing logical font. The reference font is created when a height is set to be the em-square size of the physical font, so as to get the most accurate font metrics information.

```
HFONT CreateReferenceFont(HFONT hFont, int & emsquare)
```

```
{  
LOGFONT      lf;  
OUTLINETEXTMETRIC otm[3]; // big enough for the strings  
HDC hDC      = GetDC(NULL);  
HGDIOBJ hOld = SelectObject(hDC, hFont);  
int size = GetOutlineTextMetrics(hDC, sizeof(otm), otm);  
SelectObject(hDC, hOld);  
ReleaseDC(NULL, hDC);  
  
if ( size )           // TrueType  
{  
GetObject(hFont, sizeof(lf), & lf);  
emsquare  = otm[0].otmEMSSquare; // get em-square size  
lf.lfHeight = - emsquare;       // font size for 1:1 mapping  
lf.lfWidth  = 0;               // original proportion  
return CreateFontIndirect(&lf);  
}  
else  
    return NULL;  
}
```

There are still more font metrics querying functions beyond those discussed in this section. They will be discussed together with the GDI text drawing and text formatting functions.

[< BACK](#) [NEXT >](#)

15.3 SIMPLE TEXT DRAWING

A device context has several attributes which are shared by text drawing functions. They include foreground color, background color, background mode, text alignment, etc.

```
COLORREF SetTextColor(HDC hDC, COLORREF crColor);
COLORREF GetTextColor(HDC hDC);
COLORREF SetBkColor(HDC hDC, COLORREF crColor);
COLORREF GetBkColor(HDC hDC);
int    SetBkMode(HDC hDC, int iBkMode);
int    GetBkMode(HDC hDC);
```

SetTextColor sets a color reference which will be used to draw the foreground pixels in a text string. Foreground pixels are defined as those forming the interior of glyphs in a text string. SetTextColor returns the original text foreground color in a device context. As a little reminder, there are three ways to specify a color reference: RGB(red, green, blue), PALETTEINDEX(index), or PALETTERGB(red, green, blue). GetTextColor returns the current text foreground color.

In the box formed by the starting point of the drawing and the height/width of the text string, the pixels not defined to be foreground pixels are called background pixels. SetBkColor sets the color reference which will be used to draw the background pixels, returning the original text background color reference. GetBkColor returns the current background color.

Drawing the background pixels together with foreground pixels may not always be desirable. For example, if you're adding a title string over a photo, if there is enough contrast between the background image and the text color, you may prefer not to draw the background pixels. SetBkMode sets the background mode attribute of a device context, which controls whether background pixels need to be drawn. GDI defines two background modes: OPAQUE mode fills background with background color, and TRANSPARENT mode leaves background unchanged. GetBkMode returns the current background mode in a device context.

Text Alignment

Finally, here is the basic function to draw a text string into a device context, along with two functions to control the alignment of text drawing:

```
BOOL TextOut(HDC hDC, int nXStart, int nYStart, LPCTSTR lpString,
             int cbString);
UINT SetTextAlign(HDC hDC, UINT fMode);
UINT GetTextAlign(HDC hDC);
```

The TextOut function draws a character string at a specified location, using the currently selected font, text foreground color, text background color, background mode, and some other control attributes. The text string to draw is specified using a pointer to the first character and number of characters. So the string does not need to be zero

terminated. Characters in the character string should be in the code page corresponding to the character set of the current font. For example, if the ANSI character set is being used, all control characters will be displayed using the missing-character glyph.

The exact interpretation of the nXStart and nYStart parameters is controlled by the text alignment attribute of the device context. Text alignment is specified using a combination of several flags, as shown in [Table 15-2](#).

Text alignment flags are divided into four groups, controlling vertical alignment, horizontal alignment, current position updating, and right-to-left order for Hebrew/Arabic. Flags in different groups can be combined using logical OR. The default text alignment attribute is TA_TOP | TA_LEFT | TA_NOUPDATECP.

Note that when the TA_UPDATECP flag is turned on, GDI ignores the position specified by (nXStart, nYStart). Instead it uses the current position in a device context as the starting point to display text and updates the horizontal coordinate of the current position after each text drawing call.

Table 15-2. Text Alignment

Group	Flag	Meaning
Vertical	TA_TOP	Top edge (ascent line) of text is aligned with nYStart.
	TA_BASLINE	Baseline of text is aligned with nYStart.
	TA_BOTTOM	Bottom edge (descent line) of text is aligned with nYStart.
Horizontal	TA_LEFT	Left edge of text is aligned with nXStart.
	TA_CENTER	Horizontal center of text is aligned with nXStart.
	TA_RIGHT	Right edge of text is aligned with nXStart.
Updating	TA_NOUPDATECP	Use nXStart, nYStart; current position is not updated after each text output.
	TA_UPDATECP	Ignore nXStart, nYStart, use current position, update current position after each text output.
Right to Left	TA_RTLREADING	Text is laid out in right-to-left order. On Windows 2000, RTL reading is available only when Arabic or Hebrew language support is installed. On previous systems, it's only for Hebrew and Arabic versions of the OS.

Here is a small piece of code to illustrate some interesting combinations of text alignment flags.

```
SetTextColor(hDC, RGB(0, 0, 0));      // black
SetBkColor(hDC,  RGB(0xD0, 0xD0, 0xD0)); // gray
SetBkMode(hDC,   OPAQUE);           // opaque

int x = 50;
int y = 110;

for (int i=0; i<3; i++, x+=250)
{
    const UINT Align[] = { TA_TOP    | TA_LEFT,
                          TA_BASELINE | TA_CENTER,
                          TA_BOTTOM  | TA_RIGHT};
```

```
SetTextAlign(hDC, Align[i] | TA_UPDATECP);
MoveToEx(hDC, x, y, NULL);      // set cp
TextOut(hDC, x, y, "Align", 5);
POINT cp;
MoveToEx(hDC, 0, 0, & cp);      // get cp
Line(hDC, cp.x-5, cp.y+5, cp.x+5, cp.y-5); // mark cp
Line(hDC, cp.x-5, cp.y-5, cp.x+5, cp.y+5);
Line(hDC, x, y-75, x, y+75); // vertical refer line
}
x -= 250 * 3;
Line(hDC, x-20, y, x+520, y); // horizontal refer line
```

The code sets up black as the text foreground color, light gray as the text background color, and the opaque background mode, so a light background will be displayed to show the range of the text display box. It then loops three times to demonstrate three combinations of horizontal and vertical alignment. The starting point for each call is marked with straight lines, while the end point is marked with a diagonal cross. [Figure 15-7](#) shows the output.

Figure 15-7. Text alignment.



When TA_TOP | TA_LEFT is given, the top left corner of the text box is aligned with the starting point. If TA_UPDATECP is turned on, the starting point is the current position, which will be updated to the top right corner of the text box.

When TA_BASELINE | TA_CENTER is given, the intersection of the baseline and the horizontal center of the text box is aligned with the starting point. If TA_UPDATECP is turned on, the starting point is the current position, which will not be updated.

When TA_BOTTOM | TA_RIGHT is given, the bottom right corner of the text box is aligned with the starting point. If TA_UPDATECP is turned on, the starting point is the current position, which will be updated to the bottom left corner of the text box.

Right-to-Left Layout and Reading

The Roman writing system, which we use in English, writes text from left to right, with horizontal lines of text filling the page from top to bottom. But it's only one of several writing systems in the world.

Most characters in Arabic and Hebrew writing systems are written from right to left, with horizontal lines of text filling the page from top to bottom.

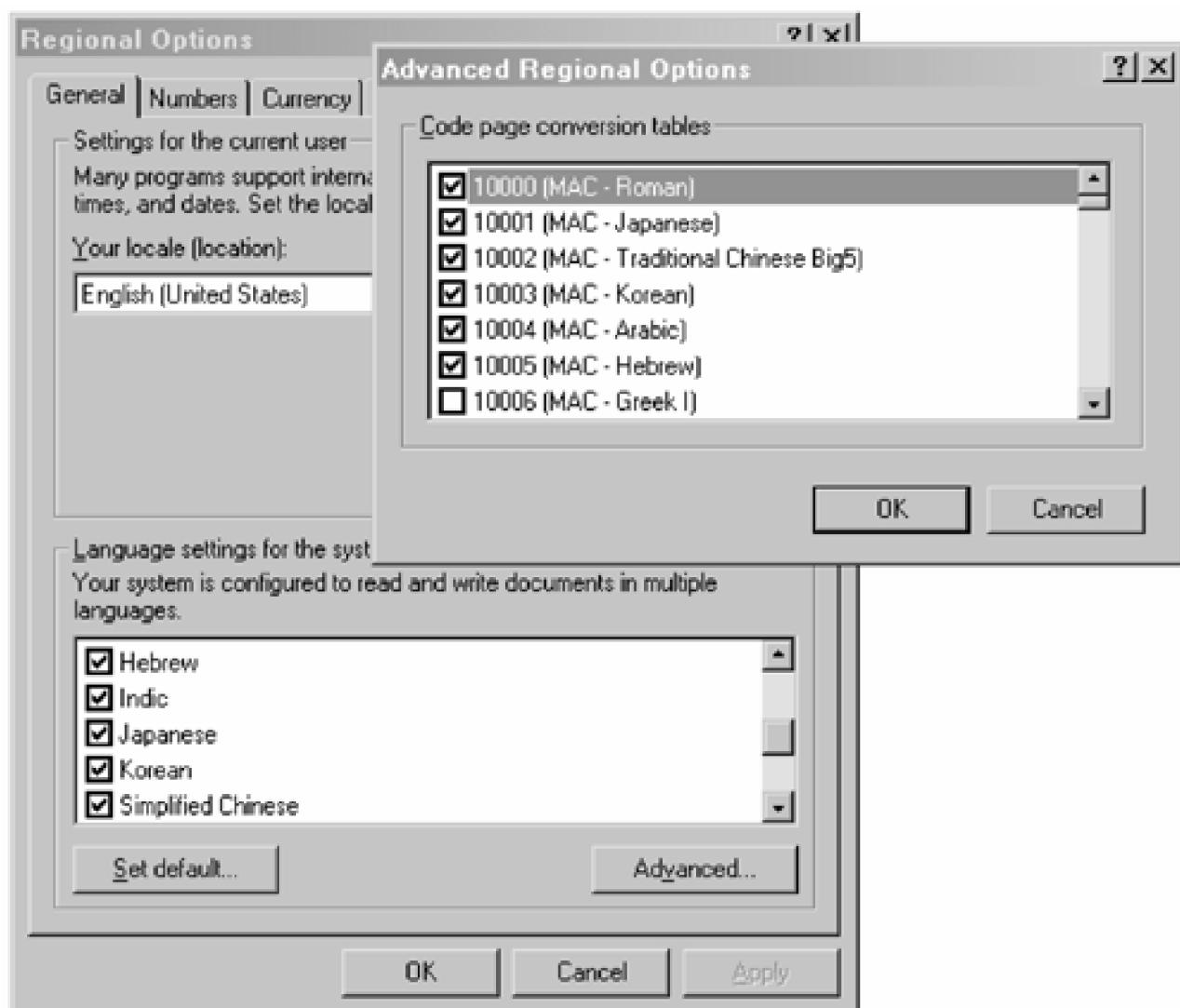
In traditional Japanese and Chinese, characters are written from top to bottom, with vertical columns of text running from right to left. Today, there are still publications written in the traditional vertical columns—for example, newspapers, novels, and magazines.

In Mongolian, characters are written in vertical columns, filling pages from left to right.

The Roman writing system is well supported by the basic text-processing features of GDI. Vertical writing systems can be supported using rotation and vertical fonts, which will be shown later in this chapter.

Supporting the right-to-left writing system is much more complicated. In Windows 2000, a single binary code is used for the whole world, which means international language support, including right-to-left reading, is built into the system. Windows 2000 is designed to support reading and writing documents in multiple languages. But different language packages are options that can be installed separately, through the Control Panel's Regional Options applet. [Figure 15-8](#) shows Window 2000's Regional Options applet.

Figure 15-8. Installing language packs for international support.



With the right language packages installed, including fonts, code page conversion files, input methods, etc., your Windows 2000 system can read and write Arabic, Armenian, Baltic, central European languages, Cyrillic, Georgian, Greek, Hebrew, Indic, simplified Chinese, Thai, traditional Chinese, Turkic, Vietnamese, western European languages, and United States English.

At the programming level, when creating a window using the CreateWindowEx function, the WS_EX_RTLREADING flag instructs texts to be displayed in right-to-left reading order if the shell language supports right-to-left reading. WS_EX_RIGHT gives the window generic “right-aligned” properties. Windows 2000 adds a new flag, WS_EX_LAYOUTRTL, which moves the window's origin to the right edge, with increasing horizontal coordinates advancing to the left.

At the GDI level, Windows 98 and Windows 2000 add a new layout attribute to a device context, which can be accessed using:

```
DWORD SetLayout(HDC hDC, DWORD dwLayout);
DWORD GetLayout(HDC hDC);
```

Only two flags are defined for a device context's layout attributes: LAYOUT_BITMAPORIENTATIONPRESERVED disables any reflection during bitmap drawing using BitBlt, StretchBlt, etc.; LAYOUT RTL sets horizontal layout to be right to left. A new bit, NOMIRRORBITMAP (0x80000000), has been added to ternary raster operations to disable horizontal reflection.

When drawing text, the TA_RTLREADING flag tells GDI to rearrange text according to right-to-left reading rules. The actual implementation of right-to-left reading on Windows 2000 is in a new GDI component, UniScribe, whose direct API offers finer control of complex script text layout.

Here is a simple example of using LAYOUT RTL and TA_RTLREADING.

```
void Demo_RTL(HDC hDC)
{
    KLogFont lf(-PointSizetoLogical(hDC, 36), "Lucida Sans Unicode");
    lf.m_lf.lfCharSet = ARABIC_CHARSET;
    lf.m_lf.lfQuality = ANTIALIZED_QUALITY;

    KGDIObject font(hDC, lf.CreateFont());
    TEXTMETRIC tm;
    GetTextMetrics(hDC, & tm);
    int linespace = tm.tmHeight + tm.tmExternalLeading;

    const TCHAR * mess = "1-360-212-0000 \xD0\xD1\xD2";

    for (int i=0; i<4; i++)
    {
        if ( i & 1 )
            SetTextAlign(hDC, TA_TOP | TA_LEFT | TA_RTLREADING);
        else
```

```
SetTextAlign(hDC, TA_TOP | TA_LEFT);

if ( i & 2 )
    SetLayout(hDC, LAYOUTRTL);
else
    SetLayout(hDC, 0);
TextOut(hDC, 10, 10 + linespace * i, mess, _tcslen(mess));
}
}
```

The Demo_RTL routine creates a logical font with the Arabic character set, using “Lucida Sans Unicode” as type face. Four lines of the same text string are drawn to test the combinations of TA_RTLREADING and LAYOUTRTL. [Figure 15-9](#) shows the display results.

Figure 15-9. RTL_LAYOUT and TA_RTLREADING.

1-360-212-0000 در 0000-212-360-1 در 0000-212-360-1 در 1-360-212-0000 در

The first line is displayed using the normal left-to-right layout and left-to-right text reading, nothing unusual. The second line is displayed using left-to-right layout, but right-to-left text reading through the TA_RTLREADING flag. Note that GDI, with the help of Uniscriber, is smart enough to break the text string into words and re arrange them according to right-to-left reading order. The string is displayed in the same place, but with character orders changed.

The third and fourth lines are the same as the first and second lines, except they are displayed after setting the device context layout to RTL_LAYOUT. Without using TA_RTLREADING, text is aligned to the right edge of the client area device surface and displayed using right-to-left reading order. If TA_RTLREADING is set as the text alignment, the text is changed back to left-to-right reading order. So the TA_RTLREADING flag actually switches reading order set in the device context.

Character Extra and Break Extra

The function TextOut handles left, center, and right alignment of text string, but it does not handle justification. Text justification normally means drawing text with a horizontal boundary defined by a left margin and right margin, making sure that the left side edge aligns with the left margin and right side edge aligns with the right margin. During justification, space characters in the string can be expanded to make the right edge align with the right margin.

Text justification with GDI is done in several steps. First, the exact width of the text string needs to be calculated; it's then compared with the difference between the right margin and left margin to find the size of the extra space that needs to be covered. Second, the number of space characters in a string needs to be known so that a GDI function can be called to set up GDI's justification. Finally, the same TextOut is used to draw a text string with justification.

A device context has another attribute that controls character spacing: *character extra*. Character extra is an integer value in logical coordinate space that will be added to every character in text drawing.

Here are related GDI functions to calculate text extent and set up character extra and justification.

```
int SetTextCharacterExtra(HDC hDC, int nCharExtra);
int GetTextCharacterExtra(HDC hDC);
BOOL GetTextExtentPoint32(HDC hDC, LPCTSTR lpString, int cbString,
    LPSIZE lpSize);
BOOL SetTextJustification(HDC hDC, int nBreakExtra, int nBreakCount);
```

The default value for character extra in a device context is zero. SetTextCharacterExtra sets it to an integer value in logical coordinate space, returning the previous character extra value. GetTextCharacterExtra retrieves the current character extra value. Character extra can be used to expand or shrink the spacing between characters.

GetTextTextentPoint32 returns the extent of a character string in logical space. The height of the extent is the height of the current font; the width of the extent is determined by the width of individual characters, character extra, and the justification setting.

SetTextJustification sets the amount of *break extra* to be distributed among *N* break characters. The break character is usually the space character, but it's possible that each font may redefine it. The actual break character for a font can be found in its TEXTMETRIC structure's tmBreakChar field.

Here is a small example of using SetTextCharacterExtra and GetTextExtentPoint32.

```
const TCHAR * mess = "Extra";
SetTextAlign(hDC, TA_LEFT | TA_TOP);

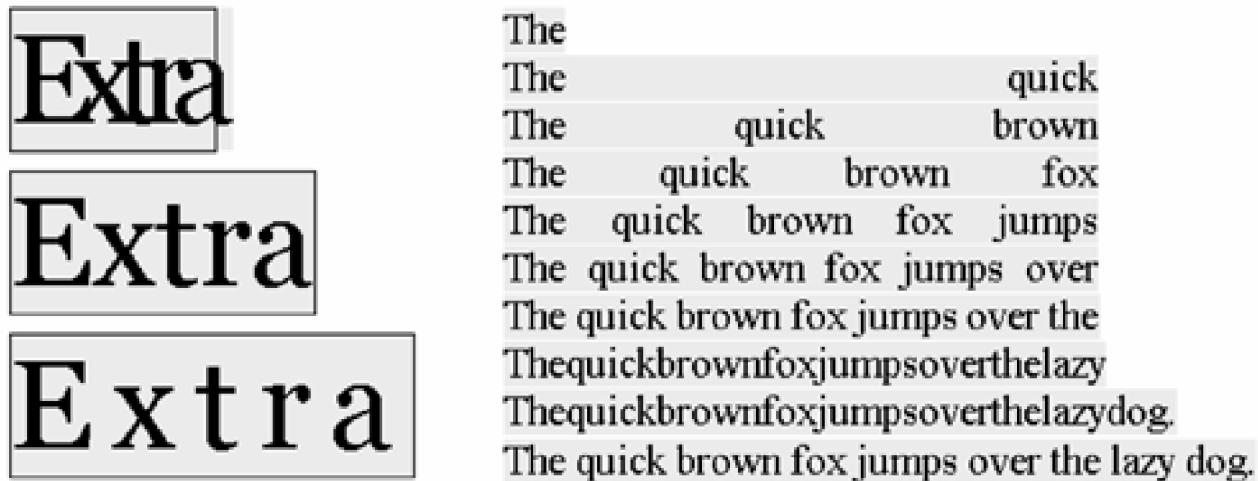
for (int i=0; i<3; i++)
{
    SetTextCharacterExtra(hDC, i*10-10); // -10, 0, 10
    TextOut(hDC, x, y, mess, _tcslen(mess));

    SIZE size;
    GetTextExtentPoint32(hDC, mess, _tcslen(mess), & size);
    Box(hDC, x, y, x + size.cx, y + size.cy); // draw text box
    y += size.cy + 10;
}
SetTextCharacterExtra(hDC, 0); // clear character extra
```

The code displays the same string three times using character extra values of -10, 0, and 10. GetTextExtentPoint32

is used to query for the dimension of the text box, which is then drawn. Its output is shown at the left side of [Figure 15-10](#). [Figure 15-10](#) shows that both positive and negative character extras are supported by GDI. Character extra is applied after each character is drawn. If character extra is negative, there is a mismatch between the text extent width as returned by GetTextExtentPoint32 and the actual text bounding box, the amount of negative character extra for the last character.

Figure 15-10. Character extra and break extra.



The following routine combines several steps in justifying a text string into a handy routine, TextOutJust. TextOutJust calculates text extent, counts the number of break characters, sets up the device context's text justification, and then draws the string justified.

```
BOOL TextOutJust(HDC hDC, int left, int right, int y, LPCTSTR lpStr,
                  int nCount, bool bAllowNegative=false, TCHAR cBreakChar=' ')
{
    SIZE size;
    SetTextJustification(hDC, 0, 0);
    GetTextExtentPoint32(hDC, lpStr, nCount, &size);

    int nBreak = 0;
    for (int i=0; i<nCount; i++)
        if ( lpStr[i]==cBreakChar )
            nBreak++;

    int breakextra = right - left - size.cx;
    if ( (breakextra<0) && ! bAllowNegative )
        breakextra =0;

    SetTextJustification(hDC, breakextra, nBreak);
    return TextOut(hDC, left, y, lpStr, nCount);
}
```

The right side of [Figure 15-10](#) shows a sample usage of the TextOutJust routine. It illustrates text justification by drawing partial sentences from a complete sentence, starting with a single word, followed by two words, three

words, and so on, until the whole sentence is complete. As can be seen from [Figure 15-10](#), with a single word, when justification is done, there is no break character. With two words, the only break character is expanded to right-justify the second word. As more words are added, all break characters share the break extra amount. When break extra becomes negative as more words are added, spaces for break characters get reduced. The minimum space for a break character is zero, after which the text string can overflow into the right margin, when it's time to break into multiple lines to form a paragraph. For comparison, the last line shows the normal sentence display with zero break and character extra.

Character Width

When you use a big point size, you will notice that the so-called left-side edge of a text box, which is aligned with TextOut call's nXStart parameter if TA_LEFT is on, is not really where the first column of the glyph is displayed. Along the horizontal axes, the exact positng and advancement of characters depends on character width and ABC character widths for TrueType/OpenType fonts.

Here are the GDI functions to query for character width and ABC character widths for the font currently selected in a device context.

```
typedef struct _ABC {
    int abcA;
    UINT abcB;
    int abcC;
} ABC;

typedef struct _ABCFLOAT {
    FLOAT abcfA;
    FLOAT abcfB;
    FLOAT abcfC;
} ABCFLOAT;

BOOL GetCharABCWidths(HDC hDC, UINT uFirstChar, UINT uLastChar, LPABC);
BOOL GetCharABCWidthsFloat(HDC hDC, UINT uFirstChar, UINT uLastChar,
                           LPABCFLOAT lpABCF);
BOOL GetCharWidth32(HDC hDC, UINT iFirstChar, UINT iLastChar,
                    LPINT lpBuffer);
BOOL GetCharWidthFloat(HDC hDC, UINT iFirstChar, UINT iLastChar,
                       PFLOAT pxBuffer);
```

GetCharABCWidths fills an array of ABC structure for all the characters within a specified range. An ABC structure specifies the width of a character in three parts. The A-width, or left-side bearing, is the amount to move from current cursor position to the starting point of a glyph. The B-width, or character glyph width, is the width of the actual glyph. The C-width, or right-side bearing, is the amount to move from the end point of a glyph to the next cursor position. The B-width is an unsigned integer, so it must be positive. But both A- and C-widths are signed integers, which can actually be negative values. GetCharABCWidthsFloat is similar to GetCharABC Widths, except that the ABCFLOAT structure it returns uses a single-precision floating-point number instead of an integer number.

The values returned by GetCharABCWidths and GetCharWidthsFloat are in logical coordinate space. You may be disappointed to know that the ABCFLOAT structure does not provide more accurate values than the ABC structure,

except when the mapping from device coordinate space to logical coordinate space is not an integer scale. That is to say, both functions get widths from an internal width table which is scaled to the current font's size in device coordinate space; the ABC structure scales them to logical coordinate space as integer values, while the ABCFLOAT structure scales them to logical coordinate space as floating-point values. For example, in MM_TEXT mapping mode, both structures hold the same value in different formats.

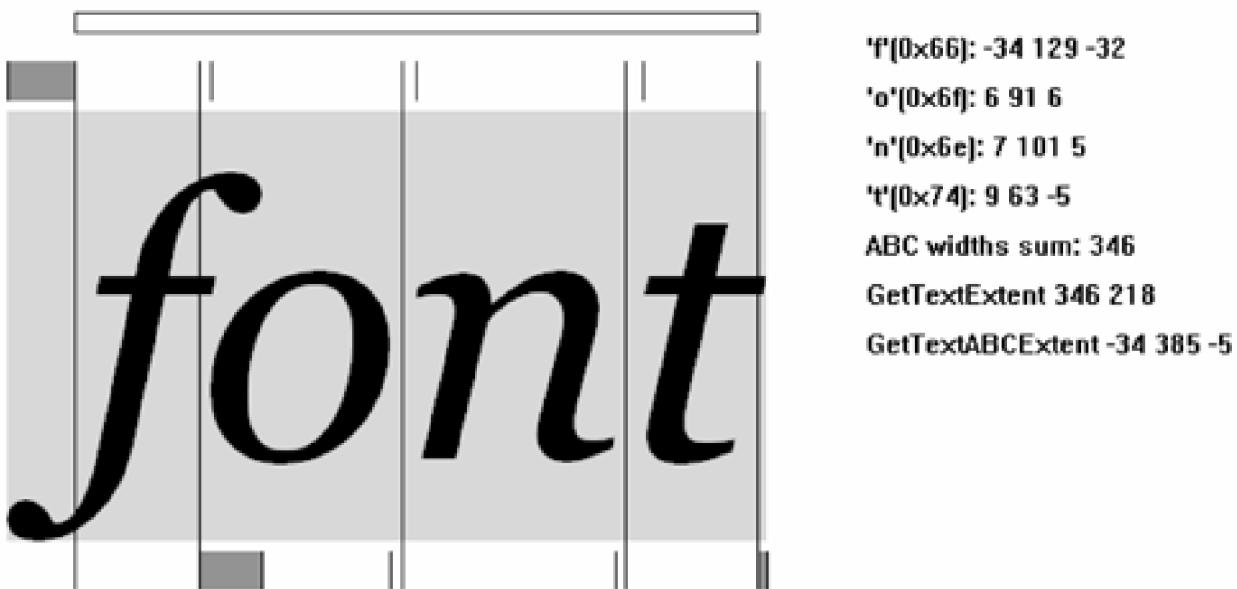
GetCharABCWidths only works for TrueType/OpenType fonts, but GetCharABCWidthsFloat more considerably supports bitmap and vector fonts at the same time. For bitmap and vector fonts, which do not have A- and C-widths for their glyphs, GetCharABCWidthsFloat just fills A- and C-widths with zeros.

For all fonts supported by Windows, you can always use GetCharWidth32 to fill an array with the advance width of characters within a certain range. For a TrueType/OpenType font, the advance width is the sum of A-width, B-width, and C-width; for bitmap and vector fonts, the advance width is the character width. GetCharWidthFloat is the floating-point version of GetCharWidth32.

On Windows 2000, there seems to be a defect in GetCharWidthFloat's implementation; the values returned are 1/16 of the actual values. It is possible that no application uses this function anyway.

[Figure 15-11](#) illustrates how ABC widths are used in GDI's text drawing and their relationship with text extent returned by GetTextExtentPoint32.

Figure 15-11. ABC widths in text drawing.



[Figure 15-11](#) shows a detailed drawing for the word "font" using an italic font at 144 points. The italic font style is chosen because glyphs in it have more visible negative A- and C-widths, especially for letters "f," "j," "t," etc. The long box on the top shows the horizontal starting point of the text drawing and its horizontal extent as returned by GetTextExtentPoint32. The right-hand side shows ABC widths for each character, their sum, and text extent.

The long vertical lines that go through the character height specify the starting and end points for each character. The distance between two subsequent lines is the advance width for that character (sum of its A-, B-, and C-widths). As you can see, the horizontal extent of a character string is the sum of all its characters' advance width, plus possible character extra and break extra, if any.

The lines and boxes immediately above the character drawing illustrate the A-widths for each character; shadowed areas mean negative A-widths. Similarly, the lines and boxes immediately below the character drawing illustrate the C-widths for each character.

For the first letter "f," which has a significant negative A-width, the left edge of the glyph is moved 34 pixels to the left. Although the B-width of the letter "f" is quite large, its negative C width moves the starting point of letter "o" much closer to it. Letter "o" and letter "n" have both positive A- and C-widths, matching the white spaces. Letter "t" has positive A width, but negative C-width.

The good news is that when GDI draws a string, ABC widths are properly taken into account to position the glyphs correctly, which does not take extra effort on the application program side. The bad news is that when the font being used has negative A-width or negative C-width, the drawing of a text string does not exactly start at the reference point the application sets and does not stop at the position reported by GetTextExtentPoint32. The A-width of the first character and the C-width of the last character in a string can move the bounding box of a text string.

There are several problems with improper handling of negative A- and C-widths. Parts of a glyph within these areas can be unintentionally clipped. You can see this problem by trying Wordpad; change font to 72-point "Times New Roman" italic, type in letter "f," and its negative A-width portion is clipped. The bounding box for a text string can be calculated wrongly, which may be needed for extra processing after a text string is drawn. Such clipping can be frequently seen in other more sophisticated applications, too.

As for a single character in a TrueType font, the width of a text string should be divided into three parts, A-width, B-width, and C-width. The A-width of a string is the A width of its first character, the C-width is the C-width of its last character, and the B-width is the sum of rest of the widths in a string.

The following routine shows how to calculate the ABC width of a text string.

```
// ( A0, B0, C0 ) + ( A1, B1, C1 ) = ( A0, B0+C0+A1+B1, C1 )
BOOL GetTextABCExtent(HDC hDC, LPCTSTR lpString, int cbString,
    long * pHeight, ABC * pABC)
{
    SIZE size;
    if ( !GetTextExtentPoint32(hDC, lpString, cbString, & size) )
        return FALSE;

    * pHeight = size.cy;
    pABC->abcB = size.cx;
    ABC abc;
    GetCharABCWidths(hDC, lpString[0], lpString[0], & abc); // first
    pABC->abcB -= abc.abcA;
    pABC->abcA = abc.abcA;

    GetCharABCWidths(hDC, lpString[cbString-1],
        lpString[cbString-1], & abc); // last
    pABC->abcB -= abc.abcC;
    pABC->abcC = abc.abcC;

    return TRUE;
```

}

For the string shown in [Figure 15-9](#), GetTextABCExtent returns an ABC structure with {-34, 385, -5} , which says that the text string is actually 385 units in length, starting at -34 units from the left reference point.

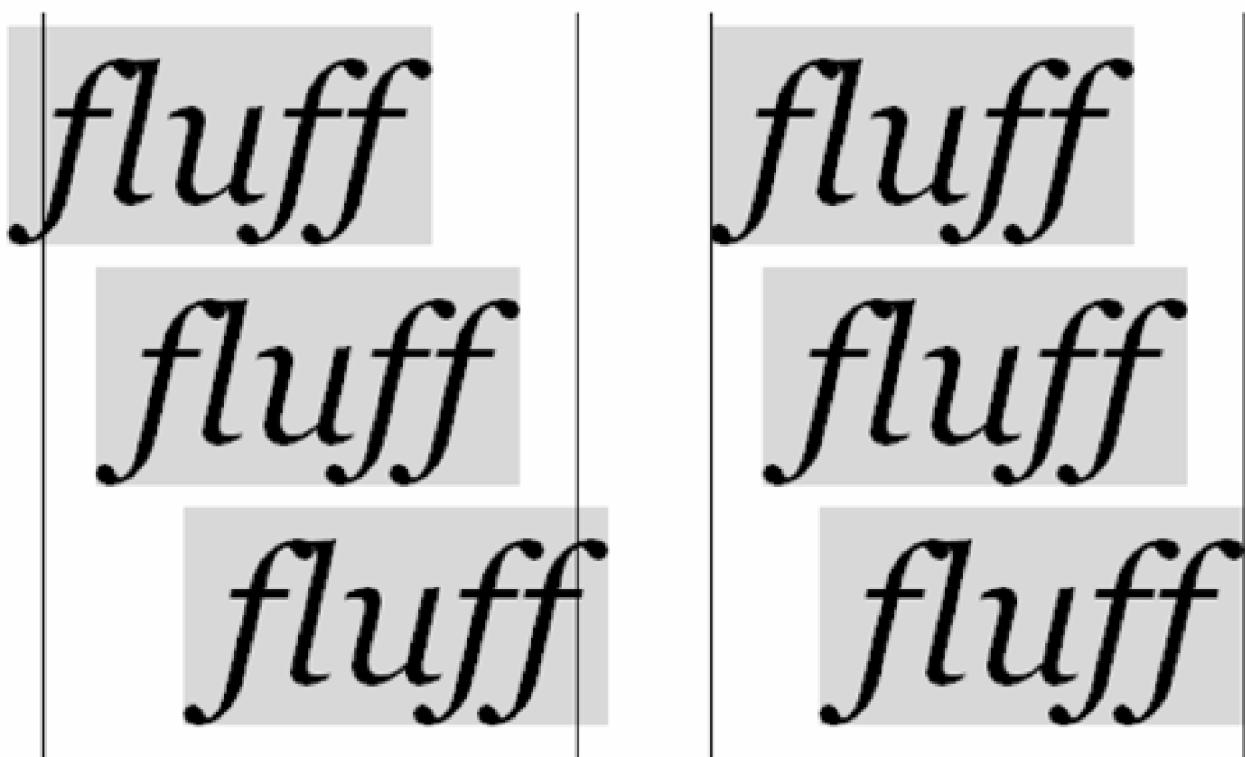
When the TextOut function aligns a text string according to text alignment attributes, A- and C-widths are not taken into consideration. If A-width for the first character is negative, this part of the glyph may be clipped. If the A-width is a positive number, the text is not really aligned precisely on the right, especially when text strings of different fonts or font sizes are aligned together. The same can be said about the right edge of the text. The following routine shows how to precisely align text at pixel-level accuracy, using the GetTextABCExtent function.

```
BOOL PreciseTextOut(HDC hDC, int x, int y, LPCTSTR lpString, int cbString)
{
    long height;
    ABC abc;

    if ( GetTextABCExtent(hDC, lpString, cbString, & height, & abc) )
        switch ( GetTextAlign(hDC) & (TA_LEFT | TA_RIGHT | TA_CENTER) )
    {
        case TA_LEFT : x -= abc.abcA; break;
        case TA_RIGHT : x += abc.abcC; break;
        case TA_CENTER: x -= (abc.abcA-abc.abcC)/2; break;
    }
    return TextOut(hDC, x, y, lpString, cbString);
}
```

PreciseTextOut calculates the ABC width of a text string, and then adjust the x-coordinate of the TextOut call according to the current horizontal text alignment flag. [Figure 15-12](#) illustrates the difference between GDI's default alignment implementation and the adjustment done by PreciseTextOut. The first column shows results from normal TextOut calls when asked to left-align, center, and right-align a string text. Parts of a glyph outside the two reference lines normally get clipped—for example, in a Word or Excel table cell. The second column shows the results of a little adjustment made by PreciseTextOut; alignment is made at pixel-level accuracy. By the way, there are few English words starting with "f" and ending with "f" with which to demonstrate this problem.

Figure 15-12. Text alignment: TextOut and PreciseTextOut.



The graphics engine internally works in device coordinate space, which is the source for all scaled text metrics information. When the mapping from device coordinate space to logical coordinate space is not an integer-ratio mapping, errors may be introduced when querying for metrics information using GetTextABCWidths. Calculation based on it may suffer from cumulative error. GetTextABCWidthsFloat can reduce this coordinate-space mapping error.

If an application is designed for both screen display and printing, care must be taken to make sure that screen display matches printing output. For example, use the reference logical font mentioned in [Section 15.2](#) to get precise text metrics information.

15.4 ADVANCED TEXT DRAWING

Function TextOut is but a simple text-drawing function which provides an easy-to-use interface to GDI's text-drawing functionality. TextOut lets an application set up a few attributes in a device context and pass a string; then it displays the string in a simple manner, while hiding the complicated processing that GDI has to go through to display the string. The cost of the simplicity is that the application does not have visibility and control over the character for glyph mapping, glyph reorder, ligature, kerning, individual glyph position, etc.

For more advanced text processing, GDI provides quite a few functions which are used by more sophisticated word processors or other graphics packages.

Character-to-Glyph Mapping

The basic information within a TrueType font is a set of glyph descriptions, which can be scaled to any resolution and raster-converted to glyph bitmaps. Characters in different code pages are normally mapped to UNICODE first and then to glyph indexes using a mapping table embedded in a TrueType font.

Some of the text-processing features not directly provided by GDI may be accessible at the glyph level. Windows 2000 GDI adds a new function, GetGlyph Indices, which allows an application to map a character string to an array of glyph indexes. For early versions of the operating system, GetCharacterPlacement, which will be discussed below, can be used to do the conversion. Windows 2000 GDI also provides functions to query for glyph text metrics information.

```
DWORD GetGlyphIndices(HDC hDC, LPCTSTR lpstr, int c,
    LPWORD pgi, DWORD fl);
BOOL GetCharWidthI(HDC hDC, UINT giFirst, UINT cgi, LPWORD pgi,
    LPINT lpBUffer);
BOOL GetCharABCWidthsI(HDC hDC, UINT giFirst, UINT cgi, LPWORD pgi,
    LPABC lpabc);
BOOL GetTextExtentPointI(HDC hDC, LPWORD pgiln, int cpi, LPSIZE lpSize);
```

Glyph indexes are stored in 16-bit unsigned integers, or WORDs. UNICODE characters are also stored in 16-bit unsigned integers, so glyph index storage is large enough to support the whole UNICODE range. Function GetFontUnicodeRanges can be used to query the UNICODE character ranges supported by a TrueType font.

GetGlyphIndices maps a character string to an array of glyph indexes. Parameter pgi points to a WORD array, which should be large enough to hold all the glyph indexes. The last parameter fl tells GDI how a missing character should be handled. If it's GGI_MASK_NONEXISTING_GLYPHS, a nonsupported character will be marked as 0xFFFF. By default, the first glyph in a TrueType is always used to represent a missing glyph.

The three text metrics querying functions, GetCharWidthI, GetCharABC WidthsI, and GetTextentPointI, are very similar to the corresponding text-string-based function, except that they accept the glyph index range or array of glyph indexes as input.

The Glyph level API can be used to do special language processing not supported directly by GDI. For example, if an application wants to implement ligature or contextual glyph substitution, it can use GetFontData to query for the glyph substitution table in a TrueType font and search the table to merge glyphs together. The details of raw TrueType font format are covered in [Chapter 14](#).

Kerning

Simple text drawing using TextOut only uses normal character spacing information provided by TrueType fonts, namely the glyph ABC widths. Character extra and break extra values at the device context level are applied to every character without considering the uniqueness of the characters. A TrueType font normally contains kerning information that provides special-case adjustment to character spacing. Kerning information is encoded in a table of kerning pairs; each of them specifies the amount of extra adjustment to apply when two specific characters are adjacent to each other.

GDI allows an application to query for kerning pairs through function GetKerningPairs:

```
typedef struct tagKERNINGPAIR {  
    WORD wFirst;  
    WORD wSecond;  
    int iKernAmount;  
} KERNINGPAIR;
```

```
DWORD GetKerningPairs(HDC hDC, DWORD nNumPairs, LPKERNINGPAIR lpkrnpair);
```

Each kerning pair consists of two character codes: wFirst and wSecond, and a kerning amount. It specifies that if character wFirst is followed by character wSecond in a text string using the same font, iKernAmount should be added to text reference point after the first character and before the second character. A kerning amount is normally negative to bring characters together. But it can also be positive to push characters apart. The kerning amount is expressed in logical coordinate space when queried through GetKerningPairs.

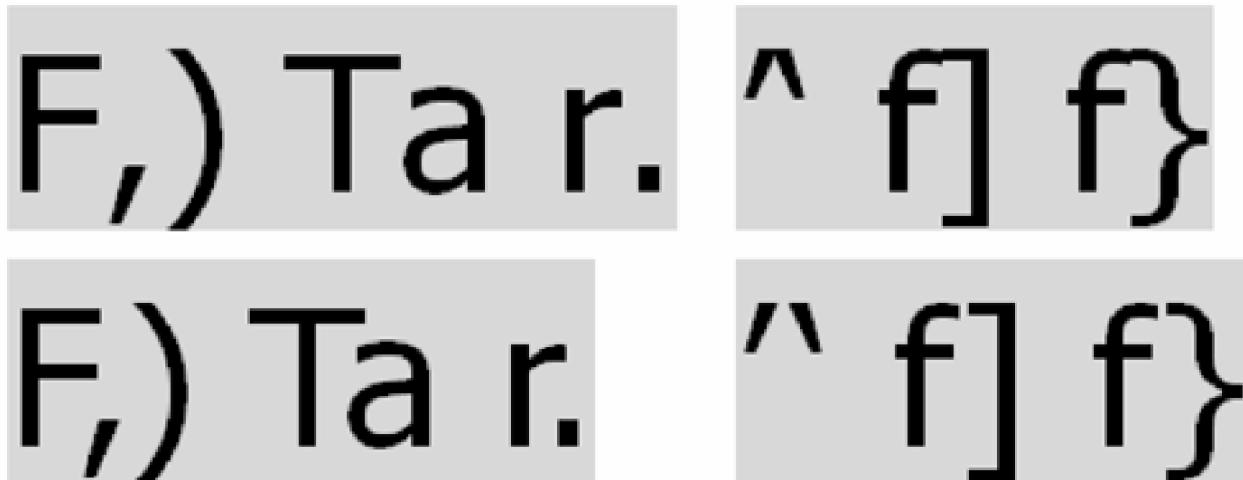
A TrueType font may contain hundreds of kerning pairs. To query for all of them, GetKerningPairs should be called first to query for the total number of kerning pairs. After sufficient memory is secured, it can be called again to get the actual kerning pairs. Here is a class declaration of a simple wrapping class for kerning pairs. Its complete implementation is on the CD.

```
class KKerningPair  
{  
public:  
    KERNINGPAIR * m_pKerningPairs;  
    int      m_nPairs;  
    KKerningPair(HDC hDC);  
    ~KKerningPair(void);  
  
    int GetKerning(TCHAR first, TCHAR second);  
};
```

The KKerningPair class has two member variables; m_pKerningPairs points to an array of KERNINGPAIR structures dynamically allocated, and m_nPairs is the number of kerning pairs for the current font. Its constructor queries for the kerning pairs, and its destructor frees allocated resources. An extra method, GetKerning, is provided to query for the kerning amount, given two character codes.

[Figure 15-13](#) shows a few sample kerning pairs. The pictures on the top show text drawing without kerning; the top left picture has four character pairs that are too far apart, while the top right picture has three character pairs that are too close together. Kerning brings characters in the left picture closer by applying a negative kerning amount, while it keeps characters from touching each other by applying a positive kerning amount.

Figure 15-13. Sample kerning pairs.



Character Placement

Using character-to-glyph mapping and querying kerning pairs directly is hard for an application to do. GDI provides a powerful function to return character placement for a string, which can then be examined or edited by the application or used for other text-processing purposes. The related definitions are:

```
typedef struct tagGCP_RESULTS {  
    DWORD  IStructSize;  
    LPTSTR lpOutString;  
    UINT   *lpOrder;  
    INT    *lpDx;  
    INT    *lpCaretPos;  
    LPTSTR lpClass;  
    LPWSTR *lpGlyphs;  
    UINT   nGlyphs;  
    UINT   mMaxFit;  
} GCP_RESULTS;
```

```
DWORD GetCharacterPlacement(HDC hDC, LPCTSTR lpString, int nCount,
```

```
int nMaxExent, LPGCP_RESULTS lpResults, DWORD dwFlags);
```

GetCharacterPlacement accepts a device context handle, a counted text string, an optional horizontal extent value, and a flag, and fills a GCP_RESULT structure with character placement information. Character placement information for a string includes character reordering, intercharacter distance, caret position, character classification, glyph index, and the number of characters to fit within a given extent. The GCP_RESULT structure itself does not hold all the information, as there can be several variable-sized arrays; instead, it holds pointers to arrays allocated elsewhere. An application is responsible to set up a GCP_RESULT properly before calling GetCharacterPlacement. If certain information is needed, a corresponding flag should be included in the last flag parameter, and the corresponding field in GCP_RESULTS should be a valid pointer; otherwise the corresponding field can be a NULL pointer.

GetCharacterPlacement is a very powerful function designed to support a wide range of text-processing features including text justification, kerning, diacritic, glyph reordering, kashida, ligature, etc. Which feature is actually implemented depends on whether it's supported by the current font and current system setting. For example, not every TrueType font has a kerning table or supports ligature. Function GetFontLangage Info can be used to query whether a certain feature is supported by certain font.

The full features of GetCharacterPlacement are complicated. Refer to MSDN documentation for details on full features supported by GetCharacterPlacement. A few simple cases, using GCP_USERKING, GCP_MAXEXTENT, and GCP_JUSTIFY in the last parameter, will be covered here.

In a GCP_RESULTS structure, lpOutString points to the output string, which is the string that should actually be displayed for an input string. Normally the output string is the same as the input string, but it could be different if the GCP_REORDER flag is set and reordering is needed, or if the GCP_MAXEXTENT flag is set and the original string is too long. The lpOrder field points to an array, which will be filled with character-order mapping from the input string to the output string. Hebrew and Arabic languages use right-to-left text ordering, which could reorder the input string.

The lpDx field points to an array, which will be filled with widths of each character in the string, which is also the distance from the current character position to next character position. The width, or distance, for a character is its advance width, plus contributions from character extra, break extra, and any kerning adjustment. The lpDx array is in output-string order. The lpCaretPos fields points to an array, which will be filled with caret positions for each character in the input-string order. It can be used by a word processor to display a moving caret during editing. OUTLINETEXTMETRIC structure contains information on character slope. The lpClass points to an array for classifying each character in the string. For example, GCPCLASS_ARABIC denotes an Arabic character.

The lpGlyphs field points to an array, which will be filled with glyph indexes for characters in a string. The glyph index array is a way to get the glyph index without using the GetGlyphIndices function, which is only provided on Windows 2000. If the same text string is used in multiple text output calls, querying the glyph indexes once and reusing them saves the time needed to do multiple translation.

The nGlyphs file initially holds the maximum number of items in the various arrays in the GCP_RESULTS structure. Upon return from GetCharacterPlacement, it holds the number of items in the arrays that are actually used.

The nMaxFit field is a return value, which is the number of characters that fit within the nMaxExtent parameter of GetCharacterPlacement. Note that nMaxFit is a character count, unlike nGlyph, which is a glyph count.

On the CD, there is a simple wrapper class for GetCharacterPlaement: class KPlacement.

Extended Text Drawing

Querying the glyph index, kerning information, and character placement eventually leads to text drawing. To apply this information in text drawing, function ExtTextOut needs to be used.

```
BOOL ExtTextOut(HDC hDC, int x, int y, UINT fuOptions, CONST RECT * lprc,  
LPCTSTR lpString, UINT cbCount, CONST INT * lpDx);
```

Function ExtTextOut is really an extended version of the TextOut call. A call to Text Out can be replaced by the following ExtTextOut call.

```
ExtTextOut(hDC, x, y, 0, NULL, lpString, cbCount, NULL);
```

Now there are only three new parameters to explain. The fuOptions parameter is a flag controlling how other parameters should be used. Parameter lprc is an optional pointer to a rectangle, which can be used as an opaque rectangle or a clipping rectangle, or both. Parameter lpDx is an optional pointer to a distance value array. [Table 15-3](#) lists acceptable values for the fuOptions parameter.

Table 15-3. Extended Text-Drawing Options

Value	Meaning
ETO_OPAQUE (0x0012)	Paint the rectangle specified by lprc using the background color before drawing the text.
ETO_CLIPPED (0x0004)	Clip text drawing to the rectangle specified by lprc.
ETO_GLYPH_INDEX (0x0010)	lpString points to an array of glyph indexes instead of character codes. Note that glyph indexes are always 16-bit values.
ETO_RTLREADING (0x0080)	Same as the TA_RTLREADING flag in text alignment. Draws text in right-to-left reading order for Arabic or Hebrew fonts.
ETO_NUMERICSLOCAL (0x0400)	Uses European digits to display numbers.
ETO_NUMERICSLATIN (0x0800)	Uses local-appropriate digits to display numbers.
ETO_IGNORELANGUAGE (0x1000)	Do not apply more language processing.
ETO_PDY (0x2000)	lpDx points to an array of pairs whose first number is horizontal distance and second is vertical distance.

The lprc parameter to ExtTextOut does not affect the normal text background drawing; instead it provides a convenient way for displaying text in a limited box—for example, a cell in a table. When the lprc parameter is not NULL, it's interpreted as the coordinates of a rectangle in device's logical coordinate space. If the ETO_OPAQUE flag is on, the rectangle is painted with the background color together with the normal text background. If the ETO_CLIPPED flag is on, the rectangle specifies an extra level of clipping using device coordinate space, instead of device coordinate space, in which the normal clipping region is expressed. To draw a text string into a table cell, an application can pass the coordinate of the cell to ExtTextOut and set both the ETO_OPAQUE and ETO_CLIPPED flags. The background of the whole cell, instead of just the text area, will be painted with the background color, and no text drawing can leak out of the box.

The `lpDX` parameter allows an application to control the exact position of each glyph, instead of relying on GDI. This design allows the application to add postprocessing to the character placement structure generated by `GetCharacterPlacement`, or to draw text in a true WYSIWYG way. Recall that the text metrics information returned from GDI to the application is in logical coordinate space, while text drawing is based on device coordinate space. When a document edited from the screen display is output to a high-resolution printer, which has much higher precision text metrics information, text layout will not match the screen display exactly. To overcome these two mismatching problems—that is, logical vs. device, and screen vs. printer—a sophisticated application can get accurate font metrics information from a reference logical font, whose size is the same as the em-square size of a physical font. All positioning calculation can be done according to an accurate em-square unit and scaled to logical coordinate space, stored in a distance array, and passed to `ExtTextOut`.

`ExtTextOut` can also be used to draw a text string in its glyph index form, as returned by `GetGlyphIndices`, or `GetCharacterPlacement`. The following method draws a text string according to the `GCP_RESULTS` structure returned by `GetCharacterPlacement`.

```
BOOL KPlacement::GlyphTextOut(HDC hDC, int x, int y)
{
    return ExtTextOut(hDC, x, y, ETO_GLYPH_INDEX, NULL,
                      (LPCTSTR) m_glyphs, m_gcp.nGlyphs, m_dx);
}
```

Here is a code fragment, which illustrates glyph indexes and kerning supported by `GetCharacterPlacement` and `ExtTextOut`.

```
void Test_Kerning(HDC hDC, int x, int y, const TCHAR * mess)
{
    TextOut(hDC, x, y, mess, _tcslen(mess)); // line 1: original

    KPlacement<MAX_PATH> placement;

    TEXTMETRIC tm;
    GetTextMetrics(hDC, & tm);
    int linespace = tm.tmHeight + tm.tmExternalLeading;

    SIZE size;
    GetTextExtentPoint32(hDC, mess, _tcslen(mess), & size);
    for (int test=0; test<3; test++)
    {
        y += linespace;
        int opt;
        switch ( test )
        {
            case 0: opt = 0; break;
            case 1: opt = GCP_USEKERNING; break;
            case 2: opt = GCP_USEKERNING | GCP_JUSTIFY; break;
        }
```

```
placement.GetPlacement(hDC, mess, opt | GCP_MAXEXTENT,  
size.cx*11/10); // 10% extra space  
placement.GlyphTextOut(hDC, x, y);  
}  
}
```

The actual sample program on the CD draws lines to mark glyph distance and shows glyph indexes and kerning amount. [Figure 15-14](#) shows a sample display.

Figure 15-14. ExtTextOut sample: glyph index and kerning.



```
gi('A')= 36, dx[ 0]= 49  
gi('V')= 57, dx[ 1]= 49  
gi('O')= 50, dx[ 2]= 50  
gi('U')= 58, dx[ 3]= 66  
gi('A')= 36, dx[ 4]= 49  
gi('L')= 47, dx[ 5]= 44  
extent 346, sum dx=315  
  
gi('A')= 36, dx[ 0]= 45, kern('A','V')= -4  
gi('V')= 57, dx[ 1]= 47, kern('V','O')= -2  
gi('O')= 50, dx[ 2]= 58, kern('O','U')= 0  
gi('U')= 58, dx[ 3]= 60, kern('U','A')= -6  
gi('A')= 36, dx[ 4]= 49, kern('A','L')= 0  
gi('L')= 47, dx[ 5]= 44  
extent 346, sum dx=303  
  
gi('A')= 36, dx[ 0]= 54  
gi('V')= 57, dx[ 1]= 56  
gi('O')= 50, dx[ 2]= 67  
gi('U')= 58, dx[ 3]= 68  
gi('A')= 36, dx[ 4]= 57  
gi('L')= 47, dx[ 5]= 44  
extent 346, sum dx=345
```

The picture on the top shows output from the normal TextOut function. The picture in the second row shows output from ExtTextOut, using the glyph index and the distance array generated by GetCharacterPlacement, but without the GCP_USE_KERNING flag. It generates exactly the same result as TextOut. The picture in the third row shows the ExtTextOut result after GCP_USERKERNING is turned on in calling GetCharacterPlacement. The bottom picture shows the result of justification using the GCP_JUSTIFY flag.

The small text display on the right shows the glyph indexes, distance information, and kerning pairs. You can see that the difference between the second and third is the result of applying kerning to the distance array, which reduces the width of the text string by 12 units. The difference between the third and fourth is due to justification; 43 units of extra space is almost evenly distributed between 5 pairs of characters (9, 9, 9, 8, 8).

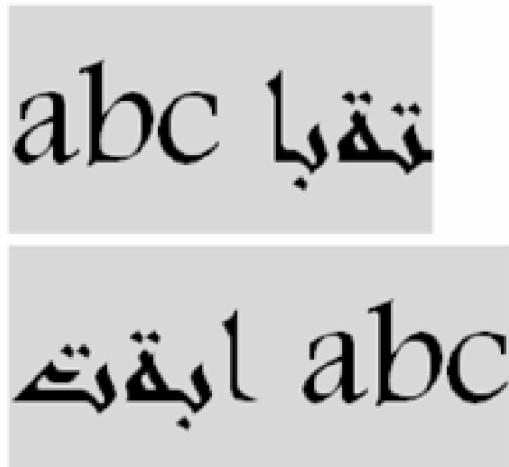
If TA_RTLREADING is set as the text alignment, and the GCP_REORDER flag is passed to GetCharacterPlacement, GDI will reorder the glyphs according to both right-to-left reading order and glyph substitution rules, if supported by the current font. Here is an example.

```
KLogFont lf(-PointSizetoLogical(hDC, 48), "Andalus");  
lf.m_lf.lfCharSet = ARABIC_CHARSET;  
lf.m_lf.lfQuality = ANTIALIASED_QUALITY;
```

```
KGDIObject font(hDC, lf.CreateFont());  
  
assert ( GetFontLanguageInfo(hDC) & GCP_REORDER );  
  
KPlacement<MAX_PATH> placement;  
  
const TCHAR * mess = "abc \xC7\xC8\xC9\xCA";  
  
SetTextAlign(hDC, TA_LEFT);  
placement.GetPlacement(hDC, mess, 0);  
placement.GlyphTextOut(hDC, x, y0);  
  
SetTextAlign(hDC, TA_LEFT | TA_RTLREADING);  
placement.GetPlacement(hDC, mess, GCP_REORDER);  
placement.GlyphTextOut(hDC, x, y1);
```

[Figure 15-15](#) shows the very interesting display results.

Figure 15-15. Glyph reordering and substitution.



```
order[0]=0, gi(0x61)= 68, dx[0]= 30  
order[1]=1, gi(0x62)= 69, dx[1]= 33  
order[2]=2, gi(0x63)= 70, dx[2]= 29  
order[3]=3, gi(0x20)= 3, dx[3]= 16  
order[4]=4, gi(0xc7)=346, dx[4]= 13  
order[5]=5, gi(0xc8)=349, dx[5]= 18  
order[6]=6, gi(0xc9)=352, dx[6]= 23  
order[7]=7, gi(0xca)=356, dx[7]= 18  
  
order[0]=5, gi(0xca)=353, dx[0]= 30  
order[1]=6, gi(0xc9)=352, dx[1]= 33  
order[2]=7, gi(0xc8)=349, dx[2]= 29  
order[3]=4, gi(0xc7)=345, dx[3]= 16  
order[4]=3, gi(0x20)= 3, dx[4]= 19  
order[5]=2, gi(0x61)= 68, dx[5]= 18  
order[6]=1, gi(0x62)= 69, dx[6]= 23  
order[7]=0, gi(0x63)= 70, dx[7]= 48
```

Compare the displays in the first and second rows: Words are swapped, characters are reordered, and characters are mapped to different glyphs. The first row displays in the normal first-character to eighth-character left-to-right order. The second row switches the order of the two words in the string, and the order of the Arabic characters. The Arabic characters are mapped to different glyphs according to their relative location within a word. For example, letter Alef (0xC7) is mapped to glyph 346 in the first row, glyph 345 in the second row.

Uniscribe

Uniscribe is a new API added to Windows 2000 that provides a fine degree of control for processing complex scripts. A complex script is any script that needs bidirectional rendering, contextual glyph shaping, ligature, specialized word-breaking and justification rules, or illegal-character-combination filtering. Hebrew, Arabic, Thai, and Indian are considered complex scripts.

The Uniscribe API is very complicated, considering that it involves nearly 30 new functions and 10 new structures. Uniscribe uses the header file usp10.h, library file usp10.lib, and runtime DLL usp10.dll. If you know that usp10.dll is even bigger than gdi32.dll, you may understand why there is hardly any space to discuss Uniscribe in this book.

Starting from Windows 2000, normal GDI text-drawing functions like TextOut and ExtTextOut have been extended to support complex scripts. But whether these features are actually used depends on font support and current system settings. If you step into GDI text output code, you can notice that GDI loads an external DLL lpk.dll, where lpk stands for language pack. Dozens of functions like LpkDrawTextEx, Lpk ExtTextOut, and LpkTabbedTextOut are exported from lpk and used by GDI, while lpk.dll itself imports functions from usp10.dll. We've already seen GDI's handling of RTL layout, RTL reading, glyph reorder, and glyph substitution, which are all based on Uniscribe in Windows 2000.

Compared with Apple's QuickDraw GX, which provides transparent complex-script processing to an application through its built-in layout mechanism, complex script processing had been a weak point of Windows GDI API. Before Uniscribe, an individual application had to understand and deal with TrueType font tables to provide advanced script-processing features. With Uniscribe, Windows applications have fewer things to do to provide complex script processing.

Accessing Glyph

As we know, glyphs in a TrueType font are described using quadratic Bezier curves. When used in text display, a glyph outline is transformed to the right size and angle, curve fitted, and then converted to bitmaps. These bitmaps are then drawn onto a device surface to fulfill the application's text-drawing request.

The text-drawing features provided by GDI are still limited. For example, GDI does not allow an application to paint the foreground of text using a brush or a bitmap; only a solid color can be specified. For an advanced application that wants to play with rendered glyphs before they are drawn, GDI provides a very low-level function, GetGlyphOutline, which allows the application to query for individual glyphs, either in their metrics form, bitmap form, or outline form. Here is a related definition for glyph querying.

```
typedef struct _GLYPHMETRICS {
    UINT gmBlackBoxX;
    UINT gmBlackBoxY;
    POINT gmptGlyphOrigin;
    short gmCellIncX;
    short gmCellIncY;
} GLYPHMETRICS;
```

```
typedef struct _FIXED {
{
    WORD fract;
    short value;
} FIXED;
```

```
typedef struct _MAT2 {
{
    FIXED eM11;
```

```
FIXED eM12;  
FIXED eM21;  
FIXED eM22;  
} MAT2;
```

```
DWORD GetGlyphOutline(HDC hDC, UINT uChar, UINT uFormat,  
LPGLYPHMETRICS lpgm, DWORD cbBuffer, LPVOID lpvBuffer,  
CONST MAT2 * lpmat2);
```

Function `GetGlyphOutline` can return three types of information for a single character at a time: glyph metrics, glyph bitmap, or glyph outline. The first parameter is a device context handle, which should have a TrueType/OpenType font selected. Parameter `uChar` is a character code, either a single-byte character or a UNICODE character, depending on whether the UNICODE version is used. Parameter `uFormat` mainly controls the data format the application wants to query; it's summarized in [Table 15-4](#). The next parameter, `lpgm`, points to a `GLYPHMETRICS` structure, which will be filled with glyph metrics information. `GetGlyphOutline` needs an application-supplied buffer to fill in glyph bitmap or outline; the buffer is specified by `cbBuffer` as buffer size and by `lpvBuffer` as buffer pointer. The last parameter, `lpmat2`, points to a partial 2D affine transformation matrix in fixed-point format.

Glyph metrics information is returned in a `GLYPHMETRICS` structure, which describes a single glyph in current device context's logical coordinate space. The `gmBlackBoxX` and `gmBlackBoxY` specify the width and height of the actual glyph's bounding box, the so-called *black box*. They are the width and height of the bitmaps returned. Note this is usually smaller than the box in which a character is defined or displayed.

Field `gmptGlphOrigin` is a `POINT` structure, which specifies coordinates of the glyph black box's top-left corner relative to the reference point of the text layout, which is on the baseline of text layout. Note that a TrueType glyph is described in em-square coordinates, whose vertical axis goes up, which is opposite of GDI's vertical axis direction in device coordinate space or logical coordinate space in `MM_TEXT` mapping mode. Fields `gmCellIncX` and `gmCellIncY` specify the amount text layout reference point should move.

Table 15-4. GetGlyphOutline Result Data Format

	Value	Meaning
Output Format	GGO_METRICS	Fill <code>GLYPHMETRICS</code> structure only. Return 0 for success, <code>GDI_ERROR</code> for failure.
	GGO_BITMAP	Fill <code>GLYPHMETRICS</code> and 1-bpp bitmap.
	GGO_NATIVE	Fill <code>GLYPHMETRICS</code> and native TrueType glyph description.
	GGO_BEZIER	Fill <code>GLYPHMETRICS</code> and glyph description in cubic Bezier curve format.
	GGO_GRAY2_BITMAP	Fill <code>GLYPHMETRICS</code> and 8-bpp bitmap, using 5 levels of grayscale.
	GGO_GRAY4_BITMAP	Fill <code>GLYPHMETRICS</code> and 8-bpp bitmap, using 17 levels of grayscale.
	GGO_GRAY8_BITMAP	Fill <code>GLYPHMETRICS</code> and 8-bpp bitmap, using 65 levels of grayscale.
	GGO_GLYPH_INDEX	<code>uChar</code> is a glyph index, instead of character code.
	GGO_UNHINTED	Do not hint (grid-fitting) glyph outline (GGO_NATIVE or GGO_BEZIER), Windows 2000 new feature.

Glyph Bitmap

GetGlyphOutline supports four glyph bitmap formats. GGO_BITMAP is for the simplest monochrome glyph bitmap, in which 0 means background pixel and 1 means foreground pixel. The remaining three glyph bitmap formats use grayscale images in 8-bits-per-pixel format, with different levels of gray levels. They are used by GDI to display antialiased text—that is, text with smooth transitions on the edges instead of sharp edges. To draw antialiased text with GDI, you just need to specify ANTI ALIASED_QUALITY as the quality field when creating a logical font. The three grayscale glyph bitmap formats, GGO_GRAY2_BITMAP, GGO_GRAY4_BITMAP, and GGO_GRAY8_BITMAP, use 4, 17, and 65 grayscale levels. You could argue that GGO_GRAY8_BITMAP should be called GGO_GRAY6_BITMAP instead.

Glyph bitmaps are either in monochrome or 8-bits-per-pixel. Each scan line is always DWORD aligned, and the scan line runs from top to bottom in the buffer returned by GDI. Therefore, the format of a glyph bitmap fits the pixel array of a top-down 1-bpp or 8-bpp DIB exactly.

The glyph bitmap is of variable size, so normally to query for the glyph bitmap, GetGlyphOutline should be called first to query for the glyph bitmap size. The second call after sufficient memory is secured really gets the glyph bitmap data.

The MAT2 structure specifies a partial 2D affine transformation using fixed-point notation in 16.16 format. A 2D affine transformation is specified by a XFORM structure, which has six floating-point numbers: eM11, eM21, eDx, eM21, eM22, and eDy. An affine transformation allows translation, scaling, reflection, rotation, shearing, and any combination of them. The MAT2 structure removes eDx and eDy from an XFORM structure, and converts the remaining four fields to fixed-point numbers. So with MAT2 structure, you can specify scaling, rotation, shearing, and reflection, but no translation. Considering that translation can be easily achieved when drawing the glyph bitmap, GetGlyphOutline really allows full affine transformation. Note that the MAT2 parameter is not optional; even if the transformation you want is an identity transformation, a correct MAT2 structure must be passed.

After getting a glyph bitmap, you can convert it to either a device-dependent or a device-independent bitmap and use GDI bitmap functions to draw it. A DIB solution is preferable because it does not involve creating new GDI objects and it's easier to control color table for the antialiased glyphs. GetGlyphOutline really provides a mechanism to the application to simulate text drawing on its own, so that the application can customize text drawing in many suitable ways. But querying the glyph bitmap and displaying can be fairly involved. Following our tradition of C++ wrapping, [Listing 15-4](#) shows the class declaration of a wrapper class for glyph bitmap handling, which wraps GetGlyphOutline and simulates GDI's drawing of a single character. The complete KGlyph class implementation is on the CD.

Listing 15-4 Bitmap Glyph Wrapping Class

```
class KGlyph
{
public:
    GLYPHMETRICS m_metrics;
    BYTE *      m_pPixels;
    DWORD      m_nAllocSize, m_nDataSize;
    int       m_uFormat;
    KGlyph();
    ~KGlyph(void);

    DWORD GetGlyph(HDC hDC, UINT uChar, UINT uFormat,
                   const MAT2 * pMat2=NULL);
```

```
BOOL DrawGlyphROP(HDC HDC, int x, int y, DWORD rop,
                   COLORREF crBack, COLORREF crFore);
BOOL DrawGlyph(HDC HDC, int x, int y, int & dx, int & dy);
};
```

The KGlyph class has four member variables, holding a GLYPHMETRICS structure, a glyph bitmap buffer, the size for the buffer, and a glyph format flag. Its constructor is fairly simple; the destructor frees memory if still allocated.

The GetGlyph method is a wrap for the GetGlyphOutline function. Compared with GetGlyphOutline, GetGlyph does not have glyph metrics and buffer parameters, because the class manages these data, and the MAT2 parameter is optional since it's simple to construct an identity matrix. KGlyph::GetGlyph sets up a default matrix, queries for data size, manages memory allocation and reuse, and finally queries for the real glyph data. It handles querying for glyph metrics, bitmap glyph, and outline glyph in a single routine.

The DrawGlyphROP method draws a glyph bitmap returned by GetGlyph Outline, given a foreground color, a background color, and a raster operation. The foreground and background colors are for simulating GDI text color and background color attributes. The raster operation field is for simulating GDI opaque and transparent background modes, or for any fancy display mode you may want to try. The Draw GlyphROP function checks the glyph bitmap format to understand the bitmap bit count and level of grays. Based on this information, it sets up a color table, using either just background and foreground colors or a linear interpolation among them according to the levels of grays. A BITMAPINFO structure for a top-down DIB is set up on the stack, and then StretchDIBits is called with the specified raster operation to display the glyph bitmap. Note that the height field in BITMAPINFOHEADER is the negation of the glyph bitmap black box's height to tell GDI it's a top-down bitmap instead of the traditional IBM bottom-up DIB.

Based on the DrawGlyphROP method, DrawGlyph simulates the text-processing part of glyph bitmap drawing. Note that the routine assumes the current vertical alignment is TA_LEFT | TA_BASELINE. It checks the device context's background mode to see if it's in opaque mode. If so, glyph background is painted with background color. After creating a brush using the current text color, the ternary raster operation 0xE20746 is used to display the glyph bitmap transparently. Recall that the unnamed raster operation 0xE20746 says: If source pixel is 1, use brush color; otherwise, leave the destination unchanged. In this case, it paints the text foreground pixels using the current text color (through the brush) and does not touch the background pixels.

As a generic routine DrawGlyph can't use the simple SRCCOPY raster operation to draw individual glyphs, because when called in a sequence to draw a string, the bounding box of a glyph may overlap with the bounding box of a previous glyph. That's why DrawGlyph needs to use a transparent raster operation to draw the glyph bitmap. An opaque background is not suggested when using DrawGlyph, unless it's used to draw a single glyph. Text background should be handled at string level before drawing any glyph.

The routine adjusts display coordinates according to glyph origin in the GLYPHMETRICS structure and draws the glyph bitmap using the DrawGlyphROP method.

Using the KGlyph class to query and draw the glyph bitmap is quite easy. Here is a simple example, which compares GDI text drawing with KGlyph text drawing using different glyph bitmap formats.

```
void Demo_GlyphOutline(HDC hDC)
{
    KLogFont lf(-PointSizetoLogical(hDC, 96), "Times New Roman");
    lf.m_lf.lfItalic = TRUE;
    lf.m_lf.lfQuality = ANTIALIASED_QUALITY;
```

```
KGDIObject font(hDC, lf.CreateFont());  
  
int x = 20; int y = 160;  
int dx, dy;  
  
SetTextAlign(hDC, TA_BASELINE | TA_LEFT);  
  
SetBkColor(hDC, RGB(0xFF, 0xFF, 0)); // yellow  
SetTextColor(hDC, RGB(0, 0, 0xFF)); // blue  
//SetBkMode(hDC, TRANSPARENT);  
  
KGlyph glyph;  
  
TextOut(hDC, x, y, "1248", 4); y+= 150;  
  
glyph.GetGlyph(hDC, '1', GGO_BITMAP);  
glyph.DrawGlyph(hDC, x, y, dx, dy); x+=dx;  
  
glyph.GetGlyph(hDC, '2', GGO_GRAY2_BITMAP);  
glyph.DrawGlyph(hDC, x, y, dx, dy); x+=dx;  
  
glyph.GetGlyph(hDC, '4', GGO_GRAY4_BITMAP);  
glyph.DrawGlyph(hDC, x, y, dx, dy); x+=dx;  
  
glyph.GetGlyph(hDC, '8', GGO_GRAY8_BITMAP);  
glyph.DrawGlyph(hDC, x, y, dx, dy); x+=dx;  
}
```

The routine creates an antialiased logical font, draws “1248” using the GDI text drawing function, and then “1,” “2,” “4,” “8” separately using GGO_BITMAP, GGO_GRAY2_BITMAP, GGO_GRAY4_BITMAP, and GGO_GRAY8_BITMAP formats. [Figure 15-16](#) shows the display result.

Figure 15-16. Drawing glyph bitmaps generated by GetGlyphOutline.



The top-left picture shows GDI output. It seems that GDI is using the GGO_GRAY4_BITMAP format internally, which offers 17 levels of grays. The bottom-left image shows the KGlyph class output. You can easily see the jagged edge for the digit "1," which uses the GGO_BITMAP format. But there is not much difference among the remaining three glyph formats. The right-hand side shows a zoom-in view on the 17-level and 65-level glyph bitmaps.

If you examined the color version of [Figure 15-16](#) carefully, you could notice that GDI is not using linear interpolation in RGB color space for intermediate grayscale levels. The color values used by GDI seem to generate nicer looking, lighter colors than what's implemented in [Listing 15-4](#).

Glyph Outline

Besides bitmap glyphs, GetGlyphOutline can return glyph outlines as a combination of lines and Bezier curves. A Bezier curve is the original data a TrueType font glyph is designed in; getting a glyph outline provides an even lower-level mechanism for an application to use original data which is very close to the raw TrueType glyph description. Lots of interesting applications can make use of glyph outlines instead of glyph bitmaps.

Three flags in the uFormat parameter control the format of glyph outline. The general format of a glyph outline is a sequence of so-called *TrueType polygons*. A TrueType polygon starts with a TTPOLYGONHEADER and is followed by a sequence of TTPOLYCURVE structures. If GGO_NATIVE is specified as the output format, only straight lines and quadratic Bezier curves are allowed in the TrueType polygons. A quadratic Bezier curve is defined by three points—two end points and one off-curve control point—while a cubic Bezier curve is defined by four points. Recall from [Chapter 14](#)'s discussion of raw TrueType data that a TrueType glyph outline is a heavily encoded mixture of lines and quadratic Bezier curves with grid-fitting instructions. So the GGO_NATIVE format does not really return the raw TrueType glyph description; nevertheless it's a nicer and much easier to parse the format for applications. Quadratic Bezier curves are not the nature of GDI level programming; GDI provides another variation of TrueType polygons. If GGO_BEZIER is specified as the output format, all quadratic Bezier curves in TrueType polygons are converted to cubic Bezier curves. By default, the outline returned is hinted—that is, glyph-fitting instructions have been applied to make the glyph look nicer and more uniform in design for small point sizes. But if the GGO_UNHINTED flag is combined with either GGO_NATIVE or GGO_BEZIER, grid-fitting instructions are not applied.

A glyph outline returned by GetGlyphOutline is scaled according to current font size with the transformation matrix applied. The glyph outline is not returned in the font's design units, neither is the transformation matrix ignored, though you may have read otherwise. Coordinates in the glyph outline are returned in high-precision 32-bit fixed-point numbers with a 16-bit signed-integer and 16-bit fraction part. The good news is that these are real high-precision numbers generated from TrueType glyph descriptions. You can even apply transformations to these numbers on your own without worrying about losing precision.

Here are the definitions for TrueType polygon.

```
typedef struct tagPOINTFX
{
    FIXED x;
    FIXED y;
} POINTFX, * LPOINTFX;

typedef struct tagTTPOLYCURVE
{
    WORD    wType;
    WORD    cpfx;
    POINTFX apfx[1];
} TTPOLYCURVE, * LPTTPOLYCURVE;

typedef struct tagTTPOLYGONHEADER
{
    DWORD   cb;
    DWORD   dwType;
    POINTFX pfxStart;
} TTPOLYGONHEADER, * LPTTPOLYGONHEADER;
```

The glyph outline for a character is returned in a data block consisting of a sequence of TrueType polygons. The number of TrueType polygons is not explicitly recorded anywhere, except that the size of the data block is known. Each TrueType polygon is a closed contour in the glyph description. It's a variable-size structure, starting with a TTPOLYGONHEADER structure, followed by a sequence of TTPOLYCURVE structures. The cb field of a TTPOLYGONHEADER is the size of a TrueType polygon, dwType is its type (the only valid value is TT_POLYGON_TYPE), and pfxStart is the starting and end point of the polygon. The pfxStart field can be seen as a MoveTo command in GDI terms. Each TTPOLYCURVE is a variable-size structure, with cpfx points inside. It can have three different types, as specified in the wType field. If wType is TT_PRIM_LINE, the curve is a polyline; if it's TT_PRIM_QSP LINE, the curve is a quadratic Bezier curve; if it's TT_PRIM_CSPLINE, the curve is a cubic Bezier curve. A polyline can be seen as a bunch of LineTo commands in GDI terms. A cubic Bezier curve always contains $N * 3$ points, which can be seen as a PolyBezierTo command. The simplest format of a quadratic Bezier curve has two points; together with the last point in the polygon, they form a segment of the curve. Generally speaking, all points except the last point in a TT_PRIM_QSPLINE curve are considered off-line control points. If there are multiple off-line control points, extra middle points between each pair of them are implicitly inserted as on-line control points. We discussed that in [Chapter 14](#) when discussing TrueType raw glyph descriptions.

The main problem of decoding a TrueType polygon is walking through the sequence of polycurves and converting them to either lines or quadratic Bezier curves which are supported by GDI. GDI provides a PolyDraw function, which draws a mixture of line segments and cubic Bezier curves represented in two arrays: a POINT array for

coordinates and a BYTE array for flags. If we can decode a glyph outline to such a structure, an application can easily manipulate it or use a single GDI function to draw it. [Listing 15-5](#) shows the declaration for a KGlyphOutline class for decoding and drawing a glyph outline. Its full implementation is on the CD.

Listing 15-5 Decoding a Glyph Outline

```
template <int MAX_POINTS>
class KGlyphOutline
{
public:
    POINT m_Point[MAX_POINTS];
    BYTE m_Flag [MAX_POINTS];
    int m_nPoints;

private:
    void AddPoint(int x, int y, BYTE flag);
    void AddQSphere(int x1, int y1, int x2, int y2);
    void AddCSpline(int x1, int y1, int x2, int y2, int x3, int y3);
    void MarkLast(BYTE flag);
    void Transform(int dx, int dy);

public:
    int DecodeTTPolygon(const TTPOLYGONHEADER * lpHeader, int size);

    BOOL Draw(HDC hDC, int x, int y);
    int DecodeOutline(KGlyph & glyph);
};

};
```

The data members of the KGlyphOutline class are exactly designed for PolyDraw, a POINT array, a BYTE array, and a point count. AddPoints method adds a point with a flag telling its type. AddCSpline adds a cubic Bezier curve with three points. Add QSphere converts a quadratic Bezier curve to a cubic Bezier curve and calls Add CSpline. MarkLast method is only used to mark the last point as a closing figure point. The coordinates in a KGlyphOutline are initially in fixed-point numbers stored as a 32-bit integer, because the original FIXED structure is not easy to operate on. The Transform method converts fixed-point numbers to integers and adds a starting point (x, y). You can easily add more transformation code, for example, to scale or rotate all the points.

The DecodeTTPolygon outline and DecodeOutline routine controls the de coding of data obtained using GetGlyphOutline and feeds it to an instance of the KGlyph Outline class. It traverses the TrueType polygon and TrueType poly curve structures, inserts implicit off-line control points for quadratic Bezier curves, and calls the three service routines to build a curve. Note that for every TrueType polygon, the code marks it closed by adding a PT_CLOSEFIGURE flag. This ensures that the correct line end-cap is being used if a wide geometric pen is being used. [Listing 15-5](#) is simpler than decoding TrueType raw data, as described in [Chapter 14](#), because font drivers, graphical device drivers, and GDI have already done the toughest part of conversion: decoding raw TrueType data, transformation, and grid fitting.

Here is a sample routine, which uses the KGlyphOutline class to draw a text string in outline format. The display result, comparing OutlineTextOut and GDI's TextOut, is shown in [Figure 15-17](#).

Figure 15-17. Drawing outline text using GetGlyphOutline.



```
BOOL OutlineTextOut(HDC hDC, int x, int y, const TCHAR * str, int count)
{
    if ( count<0 )
        count = _tcslen(str);

    KGlyph      glyph;
    KGlyphOutline<512> outline;

    while ( count>0 )
    {
        if ( glyph.GetGlyph(hDC, * str, GGO_NATIVE)>0 )
            if ( outline.DecodeOutline(glyph) )
                outline.Draw(hDC, x, y);

        x += glyph.m_metrics.gmCellIncX;
        y += glyph.m_metrics.gmCellIncY;
        str++;
        count--;
    }
    return TRUE;
}
```

15.5 TEXT FORMATTING

In this chapter, we've discussed simple text drawing using the `TextOut` function, more advanced text drawing at the glyph-index level with individual glyph positioning using `ExtTextOut`, and even lower-level glyph bitmap, glyph outline querying, and drawing using `GetGlyphOutline`. Having looked at almost everything behind GDI's text drawing function, unless you prefer to work at the raw TrueType data level, for which you can refer back to [Chapter 14](#), it's time to look at how these features can be put to real use.

This section will discuss issues related to formatting text according to different requirements—that is, putting text into the right place.

Tabbed Text Drawing

The tab character is widely used in simple text processing to align text strings across rows into columns, thus making them more readable. It's still used in user interface design, or even more advanced word processing. GDI provides several functions to allow drawing text strings with embedded tab characters and to get detailed metrics about strings with tab characters.

```
LONG TabbedTextOut(HDC hDC, int x, int y, LPCTSTR lpString, int nCount,
int nTabPosition, LPINT lpnTabStopPositions, int nTabOrigin);
DWORD GetTabbedTextExtent(HDC hDC, LPCSTR lpString, int nCoount,
Int nTabPosition, LPINT lpnTabStopPositions);
```

Compared with `TextOut`, `TabbedTextOut` has three more parameters. A counted integer array is specified by `nCount` and `lpnTabStopPositions`, which stores horizontal coordinates for tab positions in sequential order. Parameter `nTabOrigin` is a value that will be added to all tab positions. If the tab position array stores relative tab position, the `nTabOrigin` can specify the origin of the tab positions, making the tab position array independent of the drawing location.

When a drawing-text string contains tab characters ("t"), GDI draws the first part of the string until it hits a tab character. It searches the tab position table to find the next tab stop position and moves the text-drawing position to the tab position, until the drawing is complete. If there are more tab characters in a string, GDI is smart enough to calculate more tab positions based on the last tab position. That is to say, to specify uniform tab positions, only a single item array is needed. Note that tabbed text drawing is not designed specifically for table drawing, which arranges data in columns. To be more specific, characters after the *n*th tab character are not displayed at*n*th tab position. Instead GDI searches in the table for the next tab position after the current text position. Tab position is allowed to be negative; in this case, it tells GDI to right-justify text before that position, instead of left-justifying text after that position.

`TabbedTextOut` returns a 32-bit value whose high-order word is the height of the string and whose lower order word is the width of the string, both in logical coordinates. `GetTabbedTextExtent` returns the tabbed text dimension without drawing it.

Here is a simple example of using `TabbedTextOut` to draw strings with tabs in columns.

```
int tabstop[] = { -120, 125, 250 };

const TCHAR * lines[] =
{
    "Group" "\t" "Result" "\t" "Function" "\t" "Parameters",
    "Font" "\t" "DWORD" "\t" "GetFontData" "\t" "(HDC hdc, ...)",
    "Text" "\t" "BOOL" "\t" "TextOut" "\t" "(HDC hdc, ...)"
};

int x=50, y=50;

for (int i=0; i<3; i++)
    y += HIWORD(TabbedTextOut(hDC, x, y, lines[i], _tcslen(lines[i]),
        sizeof(tabstop)/sizeof(tabstop[0]), tabstop, x));
```

The code draws three lines of text in four columns, at positions x, x+120, x+125, and x+250, where x+120 has right-justified alignment. Note that the starting coordinate x is passed as the last parameter to GetTabbedTextOut, so values in the tab position array can be location independent. You have to make sure the distances between tab positions are big enough to cover all the strings so that the strings can be displayed in nice column format.

Simple Paragraph Formatting

In Windows' user interface design, it is often necessary to display single-line texts within a fixed rectangle, or format a long text string to be displayed in a rectangle big enough to hold multiple lines. Text display on buttons, single-line or multiple-line static controls, or edit boxes are just a few examples. The Windows window manager (USER32.DLL) provides two functions to do simple text formatting, mainly for user interface design.

```
int DrawText(HDC hDC, LPCTSTR lpString, int nCount, LPRECT lpRect,
    UINT uFormat);

typedef struct {
    UINT cbSize;
    int iTabLength;
    int iLeftMargin;
    int iRightMargin;
    UINT uiLengthDrawn;
} DRAWTEXTPARAMS;
```

```
int DrawTextEx(HDC hDC, LPTSTR lpchText, int cchText, LPRECT lpRect,
    UINT dwDTFormat, LPDRAWTEXTPARAMS lpDTPParams);
```

DrawText draws a text string into a rectangle region specified by the lpRect parameter in logical coordinates. The last parameter uFormat has more than two dozen options to control how control characters are interpreted, tab characters expanded, and ellipses used to fit the string into the rectangle, as well as some normal alignment options. Refer to your on-line documents for the full details of these options. The DrawTextEx function accepts one

more parameter, a pointer to a DRAWTEXTPARAMS structure, which specifies tab distance, left and right margins, and allows the system to fill in the length of string drawn.

DrawText and DrawTextEx are designed to handle two cases: single-line text and multiple-line text. Single-line text can be on menus, toolbars, buttons, static controls, etc. These two functions encapsulate the common needs for drawing these strings, including vertical and horizontal alignment, clipping, tab expansion, prefix interpretation (& for underline next character), and three ellipse options. Multiple-line texts can be used in multiple-line static control or edit control. DrawText and DrawTextEx offer easy-to-use, word-breaking paragraph formatting.

[Figure 15-18](#) shows several samples of using DrawText.

Figure 15-18. Text formatting using DrawText/DrawTextEx.

Open c:\Win\system32\gdi32.c

Open c:\Win\system32\gdi32

Open c:\Win\syst... \gdi32.dll

&Open c:\Win\system32\gdi...

Open c:\Win\system32\gdi...

The DrawText function draws formatted text in the specified rectangle. It formats the text according to

The DrawText function draws formatted text in the specified rectangle. It formats the text according to

The DrawText function draws formatted text in the specified rectangle. It

The left part of [Figure 15-18](#) shows single-line text formatting, which is generated using the text string and options shown below. It demonstrates DrawText's functionality to align text vertically, expand tabs, use different ellipsis options, and hide a prefix. File-name path ellipsis, which shortens file names to "c:\Win\sys... \gdi32.dll," is a neat feature. DrawText even allows modifying the original string to match what's displayed, which can be used elsewhere if the DT MODIFYSTRING is included.

```
const TCHAR * mess = "&Open \tc:\Win\system32\gdi32.dll";  
  
RECT rect = { 20, 120, 20+200, 120 + 32 };  
  
int opt[] = { DT_SINGLELINE | DT_TOP,  
    DT_SINGLELINE | DT_VCENTER | DT_EXPANDTABS,  
    DT_SINGLELINE | DT_BOTTOM | DT_EXPANDTABS |  
    DT_PATH_ELLIPSIS,  
    DT_SINGLELINE | DT_NOPREFIX | DT_EXPANDTABS |  
    DT_WORD_ELLIPSIS,
```

```
DT_SINGLELINE | DT_HIDEPREFIX | DT_EXPANDTABS |  
DT_END_ELLIPSIS,  
};
```

The right part shows DrawText's feature to break a long line of text into several lines for a multi-line paragraph. The three cases shown align texts using different horizontal alignment options.

Although DrawText/DrawTextEx provide convenient ways of formatting multiline text, they are designed only for simple text formatting for simple user interface needs, not WYSIWYG text formatting. [Figure 15-19](#) illustrates this point.

Figure 15-19. Inaccuracy of text formatting using DrawText.

The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth).

The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth).

The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth).

The DrawText function draws formatted text in the specified rectangle. It formats the text according to the specified method (expanding tabs, justifying characters, breaking lines, and so forth).

[Figure 15-19](#) draws text, using 6-, 9-, 12-, and 15-point fonts, into text boxes proportional to point size (width = 42 * point size, height = 5 * point size). Ideally, when the text string is broken into multiple lines, it should be broken at the same places, and the dimensions of each line should keep their relative proportion. But in [Figure 15-19](#) you can see that the same text string is broken at different places. While this may be acceptable for showing a license agreement in a dialog box on the screen, it's not acceptable to use in a word processor. Users will be very frustrated if text is formatted differently at different display zoom ratios or when printed to different resolution printers. The problem is clearly in the simple integer-based text metrics GDI is reporting, which suffers from an accumulative rounding error. Special care should be taken to ensure a WYSIWYG text formatting.

This device/resolution-dependent behavior of DrawText/DrawTextEx is part of the reason why dialog boxes designed for 96-dpi screens do not look as nice on 120-dpi screens. When switching from a 96-dpi display to a 120-dpi display, stock fonts are different heights. The relationships among logical resolution, font height, and character widths are not purely linear, due to rounding errors. So text displayed as 5 lines on a 96-dpi screen may become 4 lines on a 120-dpi screen, or even worse, 6 lines.

Device-Independent Text Formatting

When formatting text into multiple lines for paragraphs, a device-independent formatting algorithm should generate the same layout independent of the graphics device resolution and logical coordinate system setup. If this can be achieved, the screen display should match the printer output precisely, screen displays under different display settings should match each other, and on the same machine, changes in display zoom ratio should keep the same text layout.

The key to such a device-independent, text-formatting algorithm is getting precise text metrics information and using it for both calculating line breaks and controlling text display. We mentioned before that it's possible to create a reference logical font whose pixel size is the same as the em-square of a TrueType font, which is the size of TrueType font glyph grid. Text metrics based on such a high-resolution reference font should be accurate.

With accurate text metrics information for an em-square-sized font, text metrics for a given point size font can be calculated as floating-point numbers, which are also of high precision. With high-precision information, GetTextExtentPoint32, ExtTextOut, and even DrawText can be replaced with high-precision implementation.

[Listing 15-6](#) shows class declarations for two classes that implement device-independent text formatting. Their implementation is on the CD.

Listing 15-6 Classes for Device-Independent Text Formatting

```
class KTextFormatter
{
    typedef enum { MaxCharNo = 256 };
    float m_fCharWidth[MaxCharNo];
    float m_fHeight;

public:
    BOOL Setup(HDC hDC, HFONT hFont, float pointsize);
    BOOL GetTextExtent(HDC hdc, LPCTSTR pString, int cbString,
        float & width, float & height);
    BOOL TextOut(HDC hDC, int x, int y, LPCTSTR pString, int nCount);
    DWORD DrawText(HDC hDC, LPCTSTR pString, int nCount,
        const RECT * pRect, UINT uFormat);
};

class KLineBreaker
{
    LPCTSTR m_pText;
    int    m_nLength;
    int    m_nPos;

    BOOL SkipWhite(void); // skip white space
    void GetWord(void); // read next word
    BOOL Breakable(int pos); // try to break a word

public:
    float textwidth, textheight;

    KLineBreaker(LPCTSTR pText, int nCount);
    BOOL GetLine(KTextFormatter & formator, HDC hDC, int width,
        int & begin, int & end);
};


```

Class KTextFormatter implements device-independent versions of three GDI functions. Its member variables store character widths and text height in floating-point numbers, initialized by the Setup method. The KTextFormatter::GetTextExtent method is similar to the GDI function GetTextExtentPoint32, but returns floating-point numbers. The remaining two methods are similar to GDI functions TextOut and DrawText.

The Setup method creates a reference point with a TrueType font's em-square size. It then queries for the character width for 256 characters for the current character set, which is assumed to be a single-byte character set. More work is needed to extend it to double-byte character sets or UNICODE. Finally, it scales character widths based on the current font point size and device resolution. Note that the Setup method accepts a font point size parameter, instead of responding to information from the current font object handle. If you query for text height with a font object handle and calculate point size from it, the result may not be accurate. The point size is supposed to be already adjusted for the current logical coordinate system setup.

Implementation for the GetTextExtent and TextOut methods is quite simple. GetTextExtent just adds up widths for characters in the string. For device-independent TextOut, the original GDI TextOut function cannot be used, because GDI uses character widths in device coordinate space which are normally smaller than the real size. Luckily, GDI provides an ExtTextOut function which accepts a character-distance array. So we just need to fill in the distance array based on character widths in floating-point numbers and call ExtTextOut. Note that the code keeps a running length of characters, converts it for each character to an integer number, and calculates distance from it. This ensures that each character is never more than a half pixel away from its intended position.

The hard part of implementing DrawText is to break text into lines according to the left margin and right margin. Another class KLineBreaker is designed to do this. Its main method is KLineBreaker::GetLine, which returns a line's worth of characters that can fit into a given width. GetLine keeps adding words to a line until it's too long, then tries to figure out a way to break the last word such that the remaining characters can still fit into the given width. The Breakable method provides a very simple implementation for breaking a word.

The current implementation of KTextFormatter::DrawText method only implements multiple-line alignment text formatting. Other alignments can be added easily. It calls KLineBreaker::GetLine to get a line at a time, and then the TextOut method to draw the line.

Here is a test routine that compares GDI's DrawText function and device-independent text formatting implemented by the KTextFormatter class.

```
void Demo_Paragraph(HDC hDC, bool bPrecise)
{
    const TCHAR * mess =
        "The DrawText function draws formatted text in the specified "
        "rectangle. It formats the text according to the specified "
        "method (expanding tabs, justifying characters, breaking "
        "lines, and so forth).";

    int x = 20;
    int y = 20;
    SetBkMode(hDC, TRANSPARENT);

    for (int size=6; size<=21; size+=3) // 6, 9, 12, 15, 18, 21
    {
```

```
int width = size * 42;
int height = size * 5;
KLogFont lf(-PointSizetoLogical(hDC, size), "MS Shell Dlg");
lf.m_lf.lfQuality = ANTIALIASED_QUALITY;

KGDIObject font(hDC, lf.CreateFont());

RECT rect = { x, y, x+width, y+height };

if ( bPrecise )
{
    KTextFormatter format;

    format.Setup(hDC, (HFONT) font.m_hObj, size);
    format.DrawText(hDC, mess, _tcslen(mess), & rect,
        DT_WORDBREAK | DT_LEFT);
}
else
    DrawText(hDC, mess, _tcslen(mess), & rect,
        DT_WORDBREAK | DT_LEFT);

Box(hDC, rect.left-1, rect.top-1, rect.right+1, rect.bottom+1);
y = rect.bottom + 10;
}
```

The Demo_Paragraph routine accepts a Boolean parameter bPrecise. If it's true, the KTextFormatter class is used to display a multiple-line text string; otherwise GDI's DrawText is used. GDI's default implementation is illustrated in [Figure 15-19](#). A WYSIWYG version, as implemented by KTextFormatter::DrawText, is shown [Figure 15-20](#).

Figure 15-20. WYSIWYG text formatting using the KTextFormatter class.

The DrawText function draws formatted text in the specified rectangle.
It formats the text according to the specified method (expanding tabs,
justifying characters, breaking lines, and so forth).

The DrawText function draws formatted text in the specified rectangle.
It formats the text according to the specified method (expanding tabs,
justifying characters, breaking lines, and so forth).

The DrawText function draws formatted text in the specified rectangle.
It formats the text according to the specified method (expanding tabs,
justifying characters, breaking lines, and so forth).

**The DrawText function draws formatted text in the specified rectangle.
It formats the text according to the specified method (expanding tabs,
justifying characters, breaking lines, and so forth).**

[Figure 15-20](#) clearly shows that, with accurate floating-point text metrics, paragraph formatting and text display can be nicely scalable to the device resolution and display zoom ratio, making text display truly WYSIWYG.

15.6 TEXT EFFECTS

The previous sections have looked at how to control various aspects of text drawing as provided by GDI. Now we ask what we can do with these features to create the text effects the application wants.

Simple text drawing in a plain word processor is quite simple—black text on a white background. You just need to figure out how to format the text correctly. But there are lots of things you can do to create text effects for fancy titles or text-related artwork. This section will look at controlling text color and text shape, using text as a bitmap and as curves.

Coloring Text

GDI provides several attributes in a device context to control the foreground color, background color, and background mode of a text drawing. For example, to display text in blue, just do a `SetTextColor(hDC, RGB(0xFF, 0, 0))`. GDI further allows drawing antialiased text, which uses intermediate colors between foreground and background to create a smooth edge around glyphs. To draw antialiased text, a logical font needs to be created with `ANTIALIASED_QUALITY` in its quality field.

GDI allows only solid color to be used in the background and foreground colors. For the 256-color display mode, GDI always matches the specified text color to a solid color before drawing the text. Besides, GDI itself does not provide a way to color a text string using an arbitrary brush.

It is obviously a problem for user interface design, so a function is provided to draw a “grayed” string. The `GrayString` function, which is provided from the window manager module (`USER32.DLL`), allows coloring of a text string using a brush and removes certain pixels in the glyph to give the text string a look suitable for a disabled control. Because GDI does not directly support coloring text using a brush, `Gray String` has to create a memory device context, convert text to bitmap first and work on the bitmap. As with other non-GDI drawing functions, `GrayString` is a simple-minded function designed for simple on-screen usage. For example, `GrayString` does not handle negative A- and C-widths.

Although it's not directly supported, you can use other GDI features to implement the coloring of a text string in several different ways. A simple way is to use raster operation and achieve the coloring in three steps. If you want to color the foreground of text using color brush P, here are the steps:

- Paint with the brush P using raster operation `PATINVERT`. Now the destination becomes D^P .
- Draw the text string in transparent mode using black as the text color. Now the background is still D^P , but the foreground is 0 (black).
- Paint with the brush P again using `PATINVERT`. Now the background is restored to D, and the foreground is P.

Similar techniques can be used to draw opaque text with background and foreground brushes, paint text foregrounds using a bitmap, or using a gradient fill. The main problem here is actually calculating the bounding box of a text-drawing call, because negative A- and C-widths need to be taken into account for a generic solution.

[Listing 15-7](#) shows how the idea can be implemented. GetOpaqueBox calculates the bounding box of a string painted as background by TextOut/ExtTextOut. ColorText shows how to color the foreground of a text string with a brush. A similar routine to color text using a bitmap is not shown.

Listing 15-7 Coloring Text Using a Brush

```
BOOL GetOpaqueBox(HDC hDC, LPCTSTR lpString, int cbString,
    RECT * pRect, int x, int y)
{
    long height;
    ABC abc;

    if ( ! GetTextABCExtent(hDC, lpString, cbString, & height, & abc) )
        return FALSE;

    switch ( GetTextAlign(hDC) & (TA_LEFT | TA_RIGHT | TA_CENTER) )
    {
        case TA_LEFT   : break;
        case TA_RIGHT  : x -= abc.abcB; break;
        case TA_CENTER : x -= abc.abcB/2; break;
        default:       assert(false);
    }

    switch ( GetTextAlign(hDC) & (TA_TOP | TA_BASELINE | TA_BOTTOM) )
    {
        case TA_TOP   : break;
        case TA_BOTTOM : y = - height; break;
        case TA_BASELINE:
        {
            TEXTMETRIC tm;
            GetTextMetrics(hDC, & tm);
            y = - tm.tmAscent;
        }
        break;
        default:       assert(false);
    }

    pRect->left  = x + min(abc.abcA, 0);
    pRect->right = x + abc.abcA + abc.abcB + max(abc.abcC, 0);
    pRect->top   = y;
    pRect->bottom = y + height;

    return TRUE;
}

BOOL ColorText(HDC hDC, int x, int y, LPCTSTR pString, int nCount,
    HBRUSH hFore)
{
```

```
HGDIOBJ hOld    = SelectObject(hDC, hFore);

RECT rect;
GetOpaqueBox(hDC, pString, nCount, & rect, x, y);
PatBlt(hDC, rect.left, rect.top,
       rect.right-rect.left, rect.bottom - rect.top, PATINVERT);

int    oldBk   = SetBkMode(hDC, TRANSPARENT);
COLORREF oldColor = SetTextColor(hDC, RGB(0, 0, 0));

TextOut(hDC, x, y, pString, nCount);
SetBkMode(hDC, oldBk);
SetTextColor(hDC, oldColor);

BOOL rslt = PatBlt(hDC, rect.left, rect.top, rect.right-rect.left,
                   rect.bottom - rect.top, PATINVERT);

SelectObject(hDC, hOld);
return rslt;
}
```

The GetOpaqueBox function has to consider the A-width of the first character and the C-width of the last character in case they are negative. It uses the GetTextABCExtent function developed in this chapter. It also needs to consider different vertical and horizontal alignment options to adjust the opaque box properly. The Color Text function uses PatBlt to paint with the color using the PATINVERT raster operation twice. When drawing the text string, background mode and foreground color need to be set properly and restored after drawing. [Figure 15-21](#) shows samples using GrayString, ColorText, and BitmapText.

Figure 15-21. Coloring text using GrayString, ColorText, and BitmapText.



Both ColorText and BitmapText use three drawing calls, which causes flickering on the screen if used directly to draw to a screen DC. You can avoid flickering by creating an intermediate memory device context to cache the drawing; you can also use other techniques which do not require multiple drawing. Also, note that because of raster operation usage, antialiased fonts are not recommended, as they will generate pixels with strange colors.

Text Styles

To create a logical font, an application is allowed to specify different font styles using several attributes in a LOGFONT structure, such as weight, italic, antialiasing, under line, and strikeout.

A TrueType font family normally has four physical font files to support four different styles: normal style, bold, italic, and bold italic. The user's requirement is matched with what's installed on the current system to find the best match.

When the user-required font weight is not available, GDI tries to do a simple simulation when the available font is normal and the required font is no lighter than bold. The simulation is so simple that the difference is visible only at small font size on screen, because GDI just displays a text string twice with a single-pixel horizontal offset.

Simulating boldness in a TrueType font is just hard. If you have only a bold font, GDI can't simulate a normal weight font for you, either.

If an application asks for italic font, and it's not available, it's much easier for GDI to simulate an italic font. Recall that the GetGlyphOutline function's last parameter is a 2-by-2 transformation matrix, which handles shearing. Simulating italicness is done by just adding a little bit of shearing, plus managing changes in font metrics.

Antialiasing is a feature supported by font drivers, as we have seen in the GetGlyphOutline function. GDI uses 17-level grayscale glyph bitmaps for antialiasing text display. To enable antialiasing for a TrueType font, just put ANTI ALIASED_QUALITY in the quality field and make sure device context is in high color or true color mode.

Underlining and striking-out are implemented by GDI using related information in a TrueType font's OUTLINETEXTMETRIC structure. Some applications support multiple styles of underlining and striking out, which can be simulated from the same information.

Besides text styles supported by logical font, it's quite common for applications to provide other text styles like reverse video, shadowing, embossing, engraving, outline, etc.

Reverse video can be easily implemented by switching text foreground color and background and drawing text in opaque mode.

Shadowing style simulates a drop shadow, embossing simulates a popping-up 3D effect, and engraving simulates a sunken 3D effect. Photorealistic implementation of these text styles needs to work in bitmap mode using lighting models. Word processors normally do a simple job by displaying the same text string several times with different offsets and different colors.

Here is a routine that supports displaying one text string up to three times. The first five parameters to function OffsetTextOut are just like those for TextOut function. The next two groups of three parameters each specify the offset amount and the color for extra text-drawing calls. The routine uses (x + dx1, x + dy1, crt1) to draw the first offset string, and (x + dx2, x + dy2, crt2) to draw the second offset string, and finally draws the string at (x, y) using the original foreground color. Note that the code needs to paint an enlarged opaque rectangle, and the two offset string drawings should be in transparent mode.

```
BOOL OffsetTextOut(HDC hDC, int x, int y, LPCTSTR pStr, int nCount,
                   int dx1, int dy1, COLORREF cr1,
                   int dx2, int dy2, COLORREF cr2)
{
    COLORREF cr = GetTextColor(hDC);
    int     bk = GetBkMode(hDC);

    if ( bk==OPAQUE )
    {
        RECT rect;

        GetOpaqueBox(hDC, pStr, nCount, & rect, x, y);
        rect.left += min(min(dx1, dx2), 0);
        rect.right += max(max(dx1, dx2), 0);
        rect.top   += min(min(dy1, dy2), 0);
        rect.bottom+= max(max(dy1, dy2), 0);

        ExtTextOut(hDC, x, y, ETO_OPAQUE, & rect, NULL, 0, NULL);
    }

    SetBkMode(hDC, TRANSPARENT);

    if ( (dx1!=0) || (dy1!=0) )
    {
        SetTextColor(hDC, cr1);
        TextOut(hDC, x + dx1, y + dy1, pStr, nCount);
    }

    if ( (dx1!=0) || (dy1!=0) )
    {
        SetTextColor(hDC, cr2);
        TextOut(hDC, x + dx2, y + dy2, pStr, nCount);
    }

    SetTextColor(hDC, cr);
    BOOL rslt = TextOut(hDC, x, y, pStr, nCount);
    SetBkMode(hDC, bk);
    return rslt;
}
```

The following code shows how different calls to the same routine OffsetTextOut can achieve shadowing, embossing, and engraving effects.

```
// Shadow
SetBkColor(hDC, RGB(0xFF, 0xFF, 0)); // yellow background
```

```
SetTextColor(hDC, RGB(0, 0, 0xFF)); // blue text
OffsetTextOut(hDC, x, y, "Shadow", 6,
4, 4, GetSysColor(COLOR_3DSHADOW), // shadow
0, 0, 0);

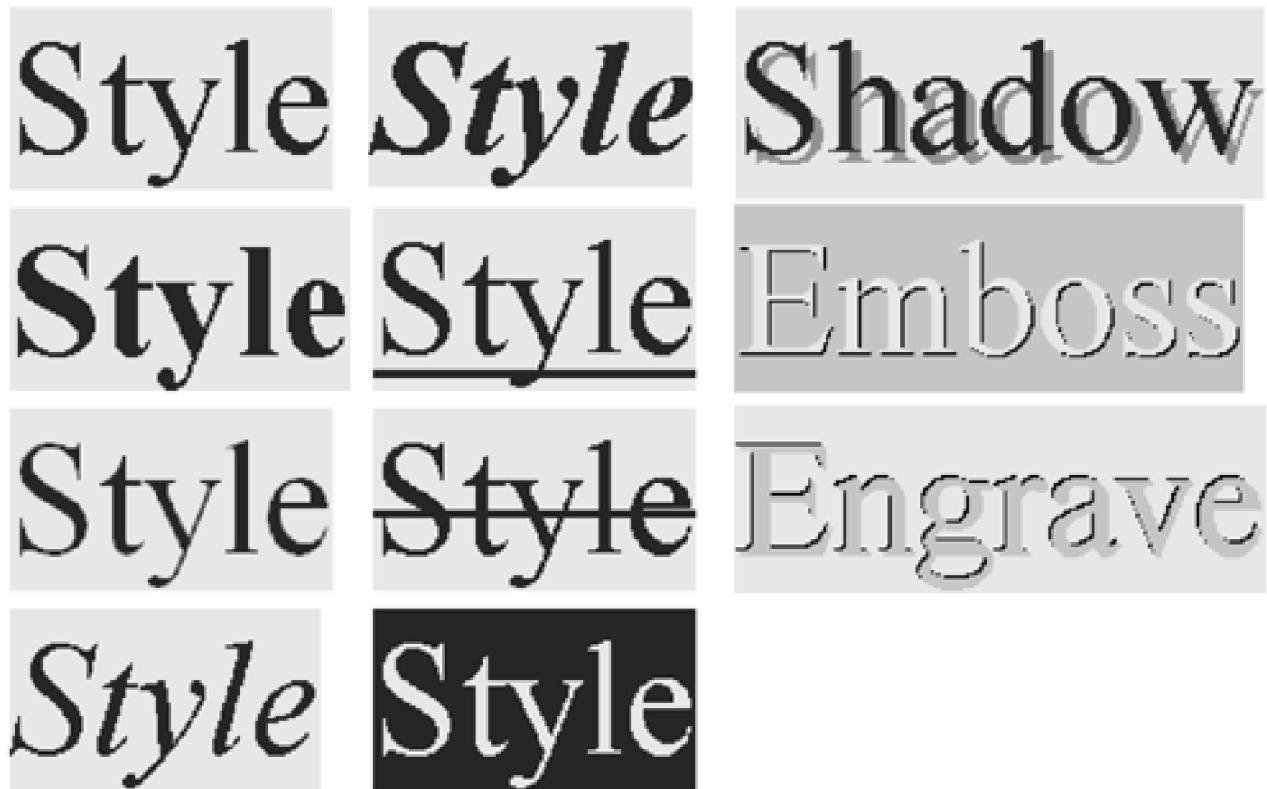
// Emboss
SetBkColor(hDC, RGB(0xD0, 0xD0, 0)); // darker yellow background
SetTextColor(hDC, RGB(0xFF, 0xFF, 0)); // yellow text
OffsetTextOut(hDC, x, y, "Emboss", 6,
-1, -1, GetSysColor(COLOR_3DLIGHT),
1, 1, GetSysColor(COLOR_3DDKSHADOW));

// engrave
SetBkColor(hDC, RGB(0xFF, 0xFF, 0)); // yellow background
SetTextColor(hDC, RGB(0xD0, 0xD0, 0)); // darker yellow text
OffsetTextOut(hDC, x, y, "Engrave", 7,
-1, -1, GetSysColor(COLOR_3DDKSHADOW),
1, 1, GetSysColor(COLOR_3DLIGHT));
```

Shadowing is simulated by displaying a gray text at offset (4, 4) first. Embossing is simulated by displaying a lighter text at offset (-1, -1), and then a darker text at offset (1, 1). Engraving is similar to embossing except the colors are reversed. Note that the amount of offsetting used here is good only for normal screen display. It should be adjusted to higher values for higher zoom ratios or for printing device context.

[Figure 15-22](#) illustrates various text styles. Starting from the first column, it shows 11 different styles of text: normal, bold, antialiased, italic, italic bold, underline, strikeout, reverse, shadow, emboss, and engrave.

Figure 15-22. Text styles.



Text Geometry

So far we've only seen proportional text in an upright position. Now it's time to play with putting text in different orientations, escapement, horizontal to vertical ratios, etc.

There are two attributes in a LOGFONT structure that are related to angles. The angle of the baseline of a row of text is specified in the lfEscapement field, while the angle of an individual character's baseline is specified in the lfOrientation. According to GDI design, a character's baseline angle may be different from a text row's baseline angle. For example, a row of text may be aligned horizontally (escapement angle equals 0), but each character may be rotated 10 degrees. But independent escapement and rotation angles are supported only in advanced graphics mode, which makes them only available in Windows NT/2000. Under compatible graphics mode, escapement specifies both escapement and rotation angles.

The escapement and rotation angles are specified as integer values in tenths of a degree, or 1/360 of a full circle, which should be quite accurate for most applications. To create a font of a certain escapement and rotation angles, you have to set the right value to its LOGFONT structure before creating a logical font. Although it's not hard, it's certainly not convenient if you switch angles frequently within a drawing routine. An alternative is keeping the same logical font setting up a different world transformation to rotate the logical coordinate space.

A really interesting application of changing a font's angle is to align text along a curve, which is a feature commonly provided by graphics packages. In GDI, any curve can be converted to a GDI path by enclosing drawing calls within BeginPath and EndPath; then the path can be converted to a polyline using FlattenPath. After that, you can use GetPath to retrieve all the points defining the path. Refer back to [Chapter 8](#) for details about lines and curves. With path data in an array, width for an individual character in a string can be queried and coordinates for a line segment on the curve with that length can be calculated. Given coordinates for a line segment $(x_0, y_0) - (x_1, y_1)$, the reference point for displaying the character and the angle of the font are both known. [Listing 15-8](#) shows a group of routines for aligning a text string on a curve defined by the current path in a device context.

Listing 15-8 Aligning Text along a Path

```
double dis(double x0, double y0, double x1, double y1)
{
    x1 -= x0;
    y1 -= y0;
    return sqrt( x1 * x1 + y1 * y1 );
}

const double pi = 3.141592654;

BOOL DrawChar(HDC hDC, double x0, double y0, double x1, double y1,
              TCHAR ch)
{
    x1 -= x0;
    y1 -= y0;

    int escapement = 0;

    if ( (x1<0.01) && (x1>-0.01) )
        if ( y1>0 )
            escapement = 2700;
        else
            escapement = 900;
    else
    {
        double angle = atan(-y1/x1);

        escapement = (int) ( angle * 180 / pi * 10 + 0.5);
    }

    LOGFONT lf;
    GetObject(GetCurrentObject(hDC, OBJ_FONT), sizeof(lf), &lf);

    if ( lf.lfEscapement != escapement )
    {
        lf.lfEscapement = escapement;

        HFONT hFont = CreateFontIndirect(&lf);

        if ( hFont==NULL )
            return FALSE;

        DeleteObject(SelectObject(hDC, hFont));
    }
    TextOut(hDC, (int)x0, (int)y0, &ch, 1);
}
```

```
return TRUE;
}

void PathTextOut(HDC hDC, LPCTSTR pString, POINT point[], int no)
{
    double x0 = point[0].x;
    double y0 = point[0].y;

    for (int i=1; i<no; i++)
    {
        double x1 = point[i].x;
        double y1 = point[i].y;
        double curlen = dis(x0, y0, x1, y1);

        while ( true )
        {
            int length;
            GetCharWidth(hDC, * pString, * pString, & length);

            if ( curlen < length )
                break;

            double x00 = x0;
            double y00 = y0;
            x0 += (x1-x0) * length / curlen;
            y0 += (y1-y0) * length / curlen;

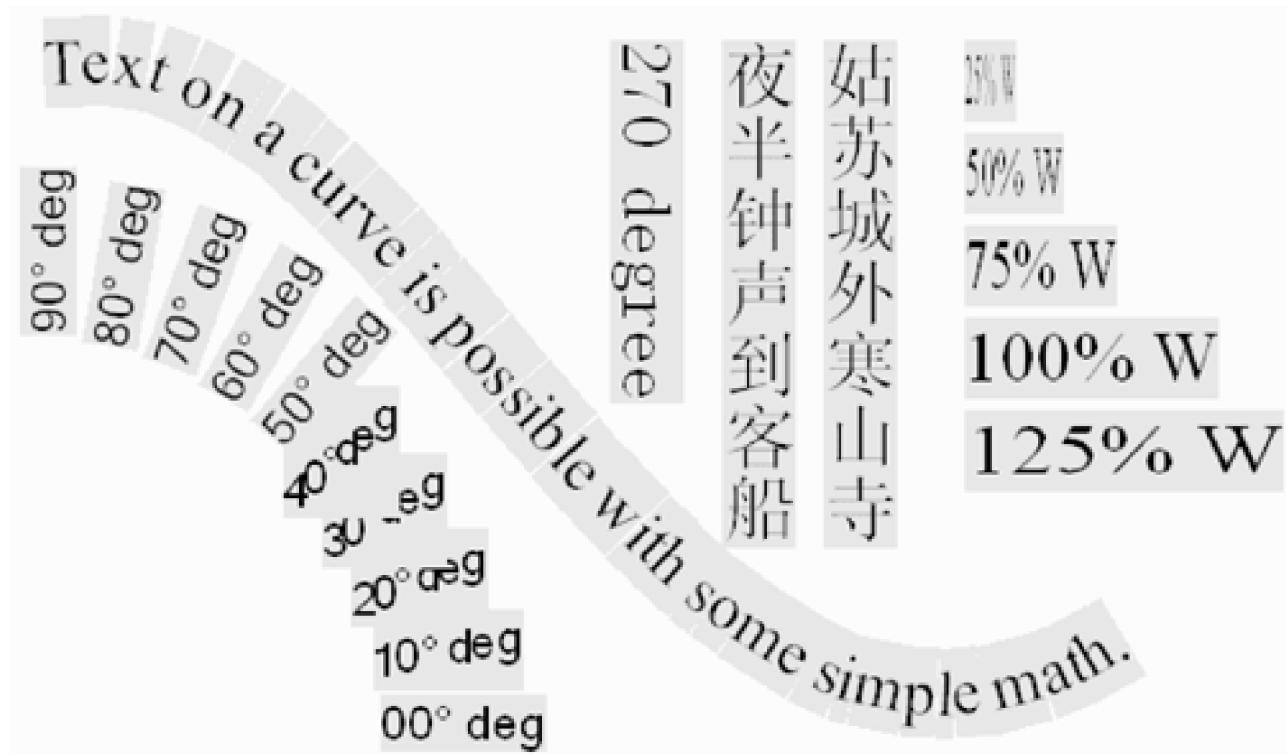
            DrawChar(hDC, x00, y00, x0, y0, * pString);
            curlen -= length;
            pString++;
        }

        if ( * pString==0 )
        {
            i = no;
            break;
        }
    }
}
```

The real font-related stuff in [Listing 15-8](#) is mostly in the DrawChar function. DrawChar accepts a device context handle, two points in a floating-point number defining a line segment on the curve, and a character code. It calculates the escapement angle using the difference between the two points, creates a logical font, and replaces the old logical font with it if necessary. After the right font is selected, drawing a single character is just a TextOut call. The first version of PathTextOut walks through points in an array defining a curve to display each character in a string. Another version of PathTextOut on the CD assumes that the curve aligns text along the path object currently in the device context. It uses the GDI path functions to convert and retrieve path data first, before calling the first version of PathTextOut.

[Figure 15-23](#) illustrates text escapement, rotation, PathTextOut function, together with vertical Asian text and a nonproportional font.

Figure 15-23. Escapement, rotation, and a nonproportional font.



The bottom-left part illustrates text escapement and rotation. Ten text strings are displayed at different angles from 0 to 90 degrees. From 0 to 40 degrees, the text strings are displayed using a rotated escapement angle with zero-degree orientation. You can see each character is still displayed in an upright position, but after a character is displayed, the text position moves both up and right to form the required escapement angle. This is also worth noticing in the rectangular-shaped opaque box, which keeps growing bigger. It's not a good idea to display opaque text when escapement and orientation are not the same. From 50 to 90 degrees, the text strings are displayed using the same escapement and orientation angles.

The long text string along a curve in the center of [Figure 15-23](#) illustrates using the PathTextOut function. With the help of these opaque boxes, you can see the text string is aligned quite evenly on the curve if there are no sharp turns.

When the text angle is 270 degrees, text goes from top to bottom, which is the same as required by top-to-bottom text rows used in traditional Chinese and Japanese writing systems. But each individual character is rotated 90 degrees to the right. For fonts supporting double-byte character sets like Chinese and Japanese, there is a font-naming conversion which allows characters to be rotated back 90 degrees to the upright positions. What you need to do is just add an "@" character before the typeface name. For example, instead of using "SimSun" use "@SimSun." [Figure 15-23](#) shows two lines from a Tang dynasty poem in traditional Chinese writing system order; text goes top-down with rows of text filling the page from right to left. The following code fragment generates the "vertical" text.

```
KLogFont lf(-PointSizetoLogical(hDC, 24), "@SimSun");
```

```
If.m_If.IfQuality = ANTIALIASED_QUALITY;
If.m_If.IfCharSet = GB2312_CHARSET;
If.m_If.IfEscapement = 2700;
If.m_If.IfOrientation = 2700;

KGDIObject font(hDC, If.CreateFont());

SetBkColor(hDC, RGB(0xFF, 0xFF, 0));

WCHAR line1[] = { 0x59D1, 0x82CF, 0x57CE, 0x5916, 0x5BD2, 0x5C71, 0x5BFA };
WCHAR line2[] = { 0x591C, 0x534A, 0x949F, 0x58F0, 0x5230, 0x5BA2, 0x8239 };

TextOutW(hDC, x0, y0, line1, 7); x0 -= 45;
TextOutW(hDC, x0, y0, line2, 7); x0 -= 50;
TextOut (hDC, x0, y0, "270 degree", 10);
```

When creating a logical font, the `IfWidth` field in the `LOGFONT` structure is normally set to zero, which tells GDI to match a font with the same aspect ratio of the graphics device. The aspect ratio of a device is the ratio formed by the width and height of a pixel on a specified device. The current device context's aspect ratio can be queried using `GetAspectRatioFilterEx`. Commonly seen graphics devices nowadays all have a 1:1 aspect ratio, so a font created with zero `IfWidth` is proportional in height and width. When a nonzero `IfWidth` is given, GDI matches it with the average width of a font to simulate a nonproportional font. But if you want to create a font a certain percentage wider or narrower than the original proportional font, you have to query the font's average width before you can set the appropriate `IfWidth` value. The average width of a font is stored in `TEXTMETRIC` structure's `tmAveCharWidth` field. The following code fragment shows how to create a logical font whose width is expressed in a percentage of the current font's width.

```
HFONT ScaleFont(HDC hDC, int percent)
{
    LOGFONT lf;
    GetObject(GetCurrentObject(hDC, OBJ_FONT), sizeof(lf), &lf);
    TEXTMETRIC tm;
    GetTextMetrics(hDC, &tm);
    lf.lfWidth = (tm.tmAveCharWidth * percent + 50) / 100;
    return CreateFontIndirect(&lf);
}
```

The right-hand side of [Figure 15-23](#) shows texts displayed with fonts 25% to 125% of their normal width. In compatible graphics mode, setting up a nonsquare logical coordinate system does not scale the font properly. You have to create the font with the right ratio yourself. But in advanced graphics, text drawing follows the logical coordinate system to device coordinate space just like other graphics elements. For accurate text formatting, font width calculation may have rounding errors; use floating-point text metrics as described in [Section 15.5](#) for accurate text-formatting information.

Text as Bitmap

GDI's text-drawing feature has limitations that makes certain effects hard to achieve in the pure text domain. For example, in compatible graphics mode, if you set up a logical-coordinate to page-coordinate mapping which flips

vertical and horizontal axes, all line and bitmap drawing follows the reflected coordinate system to go from right to left and bottom to top, but the text string is still displayed in upright position from left to right. This is very annoying if you want to implement some mirror-image effect involving text. Other restrictions on using text are that you can't use brushes on text (a solution is discussed in a previous section), and you can't apply raster operations or alpha blending. Converting text to bitmap and working in the bitmap domain instead can solve this class of problems.

There are two ways to convert a text problem into a bitmap problem. The first is to get glyph bitmap using GetGlyphOutline and to draw glyph bitmaps to simulate text drawing. The second is to convert a string into a single bitmap.

Display Text Using Glyph Bitmaps

With the KGlyph class developed in this chapter, it's easy to write a routine to display a text string using glyph bitmaps. Here is the BitmapTextOutROP routine that displays glyph bitmaps with a ternary raster operation. On the CD there is a simpler version, BitmapTextOut, which uses the KGlyph::DrawGlyph method to draw transparent glyph bitmaps.

```
BOOL BitmapTextOutROP(HDC hDC, int x, int y, const TCHAR * str,
                      int count, DWORD rop)
{
    if ( count<0 )
        count = _tcslen(str);

    KGlyph glyph;
    COLORREF crBack = GetBkColor(hDC);
    COLORREF crFore = GetTextColor(hDC);

    while ( count>0 )
    {
        if ( glyph.GetGlyph(hDC, * str, GGO_BITMAP)>0 )
            glyph.DrawGlyphROP(hDC,
                               x + glyph.m_metrics.gmptGlyphOrigin.x,
                               y - glyph.m_metrics.gmptGlyphOrigin.y,
                               rop, crBack, crFore);
        x += glyph.m_metrics.gmCellIncX;
        y += glyph.m_metrics.gmCellIncY;
        str++;
        count--;
    }

    return TRUE;
}
```

The two routines, BitmapTextOut and BitmapTextOutROP, convert a text problem to a bitmap problem, which solves problems connected with reflection in a logical coordinate system and allows for raster operations. But when using raster operations to draw glyph bitmaps, care must be taken with the background pixels, because background pixels of one glyph may overlap with background pixels of another glyph when a string of characters is displayed. The BitmapTextOutROP routine uses background color to control the color that background pixels should be

mapped to. For example, if you want to use SRCAND ROP, set the background color to white (RGB(0xFF, 0xFF, 0xFF)), which makes sure that background pixels will not affect drawing.

The following code fragment demonstrates how GDI text drawing is unable to reflect text, how to use BitmapTextOut to reflect text, and how to use raster operations with text.

```
///////////
// text reflection demo //
///////////

SaveDC(hDC);
SetMapMode(hDC, MM_ANISOTROPIC);
SetWindowExtEx(hDC, 1, 1, NULL);
SetViewportExtEx(hDC, -1, -1, NULL); // x: left, y: up
SetViewportOrgEx(hDC, 300, 100, NULL);
ShowAxes(hDC, 600, 180);

int x= 0, y = 0;

const TCHAR * mess = "Reflection";

SetTextAlign(hDC, TA_LEFT | TA_BASELINE);
SetTextColor(hDC, RGB(0, 0, 0xFF)); // blue (dark)
BitmapTextOut(hDC, x, y, mess, _tcslen(mess), GGO_GRAY4_BITMAP);

SetColor(hDC, RGB(0xFF, 0xFF, 0)); // yellow (light)
TextOut(hDC, x, y, mess, _tcslen(mess));

RestoreDC(hDC, -1);

///////////
// text raster operation demo //
///////////

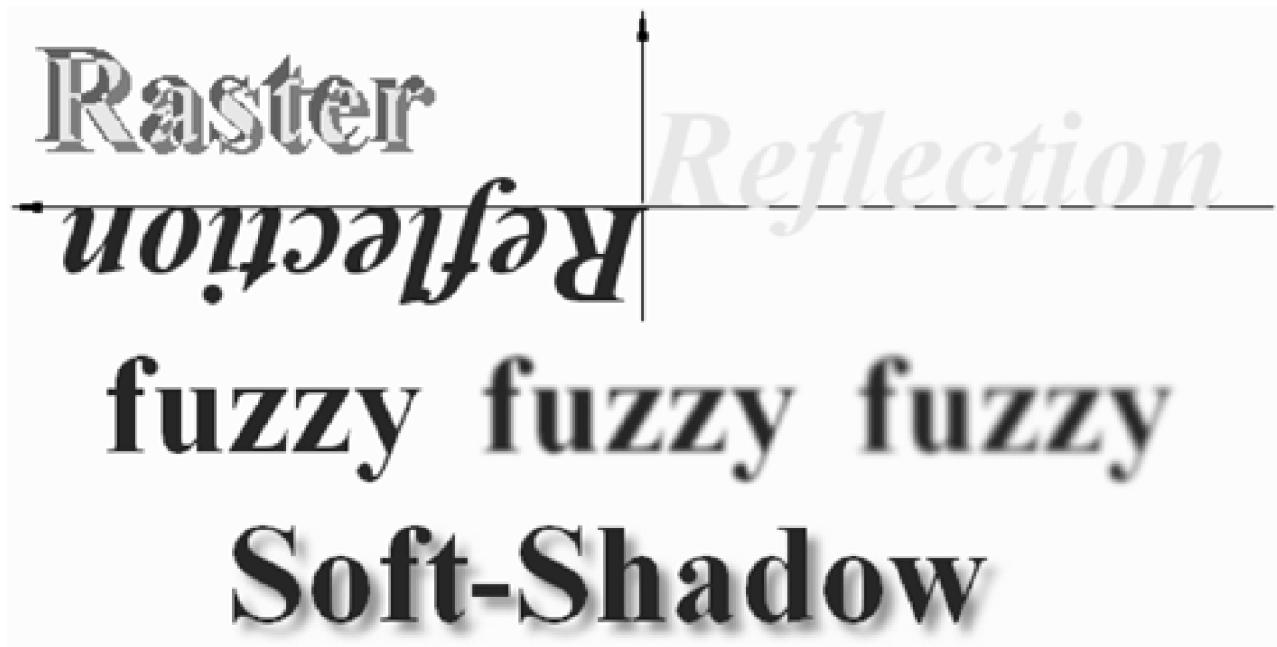
x = 10;
y = 300;

const TCHAR * mess1 = "Raster";

SetBkColor(hDC, RGB(0xFF, 0xFF, 0xFF));
SetTextColor(hDC, RGB(0xFF, 0, 0));
BitmapTextOutROP(hDC, x, y, mess1, _tcslen(mess1), SRCAND);
SetBkColor(hDC, RGB(0, 0, 0));
SetTextColor(hDC, RGB(0, 0xFF, 0));
BitmapTextOutROP(hDC, x+5, y+5, mess1, _tcslen(mess1), SRCINVERT);
```

The first part of the code sets up an anisotropic mapping mode with the x-axis going from right to left and the y-axis going from bottom to top; it displays a blue text (darker color) string using BitmapTextOut and the same string in yellow (lighter color) using GDI's function TextOut. The two strings are displayed at the same logical coordinate, but they appear in different places and different orientations on screen, as shown in [Figure 15-24](#).

Figure 15-24. Text effects using glyph.



The second part of the code uses raster operation SRCAND to draw the red string first, and the same string offset by (5, 5) pixels using a green color and the SRCINVERT raster operation. Note that a white background color is used with SRC AND, and the black background color with SRCINVERT, both to ensure that background pixels will not affect the display. The effect is shown in the top left part of [Figure 15-24](#).

Converting Text to Bitmap

If an application wants to manipulate the bitmap data before drawing it into a device context or wants to save it for later use, it's better to convert a whole string into a bitmap. This gives the application the flexibility to using bitmap or image algorithms on these data—for example, algorithms discussed in [Chapters 10–13](#) of this book.

[Listing 15-9](#) shows the class declaration for the KTextBitmap class that handles converting a text string into a DIB section, and a simple blurring filter on the bitmap. The KTextBitmap class combines several bitmap, imaging, and text techniques developed in this book. Refer to the CD for its implementation.

Listing 15-9 KTextBitmap Class: Bitmap Effects for Text

```
// Convert text string to bitmap
class KTextBitmap
{
public:
    HBITMAP    m_hBitmap;
    HDC        m_hMemDC;
    HGDIOBJ   m_hOldBmp;
    int        m_width;
    int        m_height;
```

```
int      m_dx;
int      m_dy;
BYTE *   m_pBits;

BOOL Convert(HDC hDC, LPCTSTR pString, int nCount, int extra);
void ReleaseBitmap(void);
KTextBitmap();
~KTextBitmap();
void Blur(void);
BOOL Draw(HDC hDC, int x, int y, DWORD rop=SRCCOPY);
};
```

The KTextBitmap::Convert method is the main workhorse of the KTextBitmap class. It accepts a device context handle, a string, a character count, and number of extra pixels to add onto four edges of the bitmap to be generated. The extra space is needed for the bitmap to grow when applying imaging algorithms. The main part of the routine deals with calculating the opaque box of the text to determine bitmap size, creating a 32-bit DIB section, creating a memory device context, and copying attributes from the current device context to the memory device context. The location of text display within the bitmap (m_dx, m_dy) is adjusted with the A-width of the first character to ensure that no part of glyph will be clipped. After everything is set up, drawing a text string into the bitmap is an easy call to TextOut. Note that KTextBitmap:: Convert is still a simple conversion routine, for it only handles the device context in MM_TEXT mapping mode.

To demonstrate what you can do once a text string is converted to a bitmap, a Blur method is added to the KTextBitmap class. It calls a function template, Average, to apply a 3x3 averaging filter to the RGB channels of the 32-bit DIB section.

The lower part of [Figure 15-24](#) demonstrates converting text to bitmap and the blurring filter. It shows three copies of the same word, “Fuzzy.” The first copy is the original nonantialiased bitmap, the second is blurred twice, and the last is blurred four times. The bottom part of [Figure 15-24](#) shows how to use KTextBitmap class to draw “feathered” or “soft” shadow. The following code fragment is used to generate the “soft” shadow, which uses an 8-times-blurred gray shadow bitmap.

```
KTextBitmap bmp;

SetBkMode(hDC, OPAQUE);
SetBkColor(hDC, RGB(0xFF, 0xFF, 0xFF)); // white
SetTextColor(hDC, RGB(0x80, 0x80, 0x80)); // gray

const TCHAR * mess = "Soft-Shadow";
bmp.Convert(hDC, mess, _tcslen(mess), 8);

for (int i=0; i<8; i++)
    bmp.Blur();

bmp.Draw(hDC, x, y); // draw shadow

SetBkMode(hDC, TRANSPARENT);
SetTextColor(hDC, RGB(0, 0, 0xFF)); // blue
```

```
TextOut(hDC, x-5, y-5, mess, _tcslen(mess)); // shifted solid text
```

With the DIB section generated by KBitmapText, a text problem is converted to a pure bitmap problem; you can also use imaging algorithms developed in [Chapter 12](#), creating alpha channels, or even draw to a DirectDraw surface.

Embossing and Engraving on Bitmap Background

As we saw in discussing embossing and engraving in the section on text styles, the foreground of a text string is displayed using a solid color. Embossing and engraving are often used to add annotations to a picture, where little disturbance to the background is desired. This requires embossing or engraving to display only the light and dark edges of characters, while using the background picture to form foreground pixels for the characters themselves.

With the KTextBitmap class, which converts a text string to a bitmap, we can now use raster operation techniques to generate a bitmap which contains only light-and dark-edge pixels, and draw it to any device context transparently. [Listing 15-10](#) shows the TransparentEmboss routine, which handles both embossing and engraving.

Listing 15-10 Bitmap Embossing and Engraving

```
void TransparentEmboss(HDC hDC, const TCHAR * pString, int nCount,
                      COLORREF crTL, COLORREF crBR, int offset, int x, int y)
{
    KTextBitmap bmp;

    // generate a mask bitmap with top-left and bottom-right edges
    SetBkMode(hDC, OPAQUE);
    SetBkColor(hDC, RGB(0xFF, 0xFF, 0xFF)); // white background
    SetTextColor(hDC, RGB(0, 0, 0));
    bmp.Convert(hDC, pString, nCount, offset*2); // black TL edge

    SetBkMode(hDC, TRANSPARENT); // black BR edge
    bmp.RenderText(hDC, offset*2, offset*2, pString, nCount);

    SetTextColor(hDC, RGB(0xFF, 0xFF, 0xFF)); // white main text
    bmp.RenderText(hDC, offset, offset, pString, nCount);

    // mask destination with top-left and bottom-right edges
    bmp.Draw(hDC, x, y, SRCAND);

    // create a color bitmap with top-left and bottom-right edges
    SetBkColor(hDC, RGB(0, 0, 0)); // black background
    SetTextColor(hDC, crTL);
    bmp.Convert(hDC, pString, nCount, offset); // TL edge

    SetBkMode(hDC, TRANSPARENT);
    SetTextColor(hDC, crBR);
    bmp.RenderText(hDC, offset*2, offset*2, pString, nCount); // BR edge
```

```
SetTextColor(hDC, RGB(0, 0, 0)); // black main text  
bmp.RenderText(hDC, offset, offset, pString, nCount);  
  
// draw color top-left and bottom-right edges  
bmp.Draw(hDC, x, y, SRCPAINT);  
}
```

Function TransparentEmboss works in the same way as cursor/icon display. In normal embossing or engraving, a text string is displayed three times, first at position $(x - dx, y - dy)$ using a color for top-left edge, second at position $(x + dx, y + dy)$ using a color for the bottom-right edge, and finally at position (x, y) using text foreground color. For transparent embossing or engraving, only the top-left and bottom-right edges that do not overlap with the text foreground part need to be displayed. First, the routine builds a black-on-white image with only the edge pixels, which is displayed with SRCAND raster operation to mask off edge pixels in the background image. After that, a color-on-black image is built with the required colors for top-left and bottom-right edges for the embossing or engraving effect, which is displayed with the SRCPAINT raster operation to combine with the background image.

[Figure 15-25](#) illustrates using TransparentEmbossing.

Figure 15-25. Transparent embossing and engraving.



Text as Curve

Certain text-drawing problems can't be adequately solved in the text domain, and not even in the bitmap domain. For example, drawing the outline of a glyph to achieve some special effects, applying a nonaffine transformation to text drawing, or displaying some real 3D text effects can best be attacked when text is converted to outline using lines and curves.

There are three ways to convert a text string to lines and curves. The first is by accessing TrueType font data directly using GetFontData, which is covered in [Chapter 14](#). The second is by using GetGlyphOutline to query for “native” TrueType glyph outlines, which is covered in [Section 15.4](#). Both methods can get very accurate glyph outline

definitions, making it easier to apply transformations and special effects without worrying so much about precision.

Using a GDI Path for Text

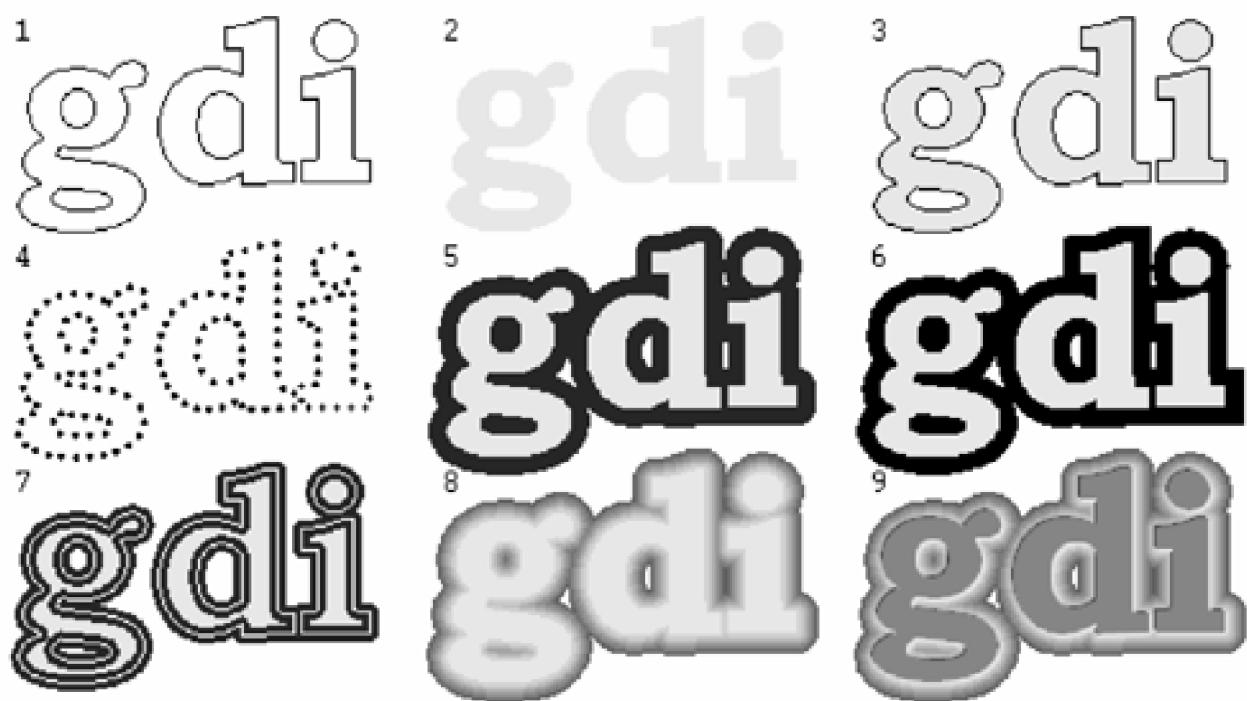
The third way of converting a text string to lines and curves is much easier: Convert text drawing to a GDI path object. When a text-drawing call using a TrueType/OpenType font is put inside a BeginPath/EndPath bracket, instead of drawing to the device context, the outlines of glyphs, together with a possible opaque rectangle, are merged into the path object that GDI is creating. After path construction is completed, an application can call path-rendering functions StrokePath, FillPath, or StrokeAndFillPath to draw a glyph outline, fill a glyph interior, or both. If the actual path data is needed for transformation, GetPath can be used to convert GDI's internal representation of a path to a POINT array and a flag array. After any transformation, PolyDraw can be used to draw from a transformed glyph outline.

Converting TrueType text drawing to a path is quite easy, as it's well supported by GDI. Here is a simple example of how to display a text outline.

```
BeginPath(hDC);           // begin path construction  
TextOut(hDC, x, y, mess, _tcslen(mess)); // convert one text call  
EndPath(hDC);           // end path construction  
StrokePath(hDC); // draw text outline using current pen (default black)
```

After a text problem is converted to a line-and-curve problem using a GDI path, lots of special effects can be quite easily achieved. [Figure 15-26](#) illustrates nine different ways a text path can be drawn. Case 1 uses StrokePath with the default stock black pen, good for outline text effects. Case 2 uses FillPath, which should be very similar to normal text display, except that some precision and antialiasing are lost. Case 3 uses Stroke AndFillPath to draw both the outline and interior fill. Case 4 uses a dotted geometric pen to draw the glyph outline using medium-sized dots. Cases 5 and 6 illustrate different geometric pen attributes: round join vs. miter join. Case 7 uses a thick pen to draw a thick outline and then a thin white pen to draw a white outline in the middle of the thick outline, achieving a double-outline effect.

Figure 15-26. Drawing text converted to path.



Cases 8 and 9 are very similar. They both use `StrokePath` to draw a sequence of outlines gradually changing from thick and darker to thin and lighter, creating a more photorealistic 3D embossing effect. Case 9 creates an engraving effect in displaying the text foreground.

Transforming Path Data

Using the path data through GDI is still limited in functionality. For example, a path object is created in device coordinate space; once it's created, changing logical to device coordinate-space mapping does not affect path drawing. So you can't move a path even by a single pixel. Luckily, GDI provides the `GetPathData` function to query for the actual data defining a path.

[Chapter 8](#) developed a simple wrapping class for using path data, `KPathData`. Class `KPathData` has a method, `MapPoints`, that applies a 2D coordinate mapping to every point controlling the path. The data after mapping can be passed to the GDI function `PolyDraw` for drawing. Note that the `MapPoints` method applies transformations to all control points in a path and only to control points. This is good only for affine transformations, which preserve lines and Bezier curves. For an arbitrary 2D transformation, a straight line may be transformed to a curve. But if you transform only the control points, linking the transformed points defining a line together will still produce a line, not a curve. To cater to arbitrary transformation, lines and curves should be broken into small enough segments, adding more points to be transformed to generate a more accurate transformed curve.

[Listing 15-11](#) shows the class declaration for a `KTransCurve` class which supports generic 2D transformation, and a new method, `KPathData::Draw`, which supports such transformations. Complete implementation is on the CD.

Listing 15-11 Generic Path Transformation

```
class KTransCurve
{
    int m_orgx; // first point in a figure
    int m_ory;
```

```
float x0;    // current last point
float y0;
float m_dstx;
float m_dsty;
int m_seglen; // segment length to break

virtual Map(float x, float y, float & rx, float & ry);
virtual BOOL DrvLineTo(HDC hDC, int x, int y);
virtual BOOL DrvMoveTo(HDC hDC, int x, int y);
virtual BOOL DrvBezierTo(HDC hDC, POINT p[]);

BOOL BezierTo(HDC hDC, float x1, float y1, float x2, float y2,
               float x3, float y3);

public:

KTransCurve(int seglen);
BOOL MoveTo(HDC hDC, int x, int y);
BOOL BezierTo(HDC hDC, int x1, int y1, int x2, int y2,
              int x3, int y3);
BOOL CloseFigure(HDC hDC);
BOOL LineTo(HDC hDC, int x3, int y3);
};

BOOL KPathData::Draw(HDC hDC, KTransCurve & trans, bool bPath)
{
if ( m_nCount==0 )
    return FALSE;

if ( bPath )
    BeginPath(hDC);

for (int i=0; i<m_nCount; i++)
{
    switch ( m_pFlag[i] & ~ PT_CLOSEFIGURE )
    {
        case PT_MOVETO:
            trans.MoveTo(hDC, m_pPoint[i].x, m_pPoint[i].y);
            break;

        case PT_LINETO:
            trans.LineTo(hDC, m_pPoint[i].x, m_pPoint[i].y);
            break;

        case PT_BEZIERTO:
            trans.BezierTo(hDC,
                           m_pPoint[i ].x, m_pPoint[i ].y,
                           m_pPoint[i+1].x, m_pPoint[i+1].y,
                           m_pPoint[i+2].x, m_pPoint[i+2].y);
    }
}
```

```
i+=2;  
break;  
  
default:  
    assert(false);  
}  
  
if ( m_pFlag[i] & PT_CLOSEFIGURE )  
    trans.CloseFigure(hDC);  
}  
  
if ( bPath )  
    EndPath(hDC);  
  
return TRUE;  
}
```

The KTransCurve class serves two purposes: handling of individual point transformation through the Map virtual method and handling drawing that can be customized by overwriting three drawing primitives. The details of KTransCurve class implementation are not shown except for the KTransCurve::BezierTo method. It handles recursive breakdown of Bezier curves into smaller enough segments, controlled by a parameter m_seglen passed in from the constructor. Note that distance calculation is approximated on points after the transformation, not before, because a generic transformation may enlarge a certain area to a much larger scale that needs more precision.

The KPathData::Draw method feeds the GDI path data returned by GetPathData to an instance of the KTransCurve, both for transformation and for drawing.

Here is a simple example of using this generic path data transformation package.

```
class KWave : public KTransCurve  
{  
    int m_dx, m_dy;  
  
public:  
    KWave(int seg, int dx, int dy) : KTransCurve(seg)  
    {  
        m_dx = dx;  
        m_dy = dy;  
    }  
    virtual Map(float x, float y, float & rx, float & ry)  
    {  
        rx = x + m_dx;  
        ry = y + m_dy + (int) (sin(x/50.0) * 20); // 100 pixel cycle +-20  
    }  
};  
  
// Using KPathData KWave class to Draw transformed Text Outline  
BeginPath(hDC);
```

```
TextOut(hDC, x, y, mess, _tcslen(mess)); // generate path
EndPath(hDC);

KPathData pd;
pd.GetPathData(hDC); // query path data

StrokeAndFillPath(hDC); // draw original data

{
    KWave wave(8, 360, 0); // transformation
    pd.Draw(hDC, wave, true); // apply transformation, build new path
    StrokeAndFillPath(hDC); // draw new path
}
```

The KWave class is derived from the KTransCurve class. It overrides the key Map method. Its constructor accepts a breakable segment length and offsets to be added to each control point to move it to a new starting address. The Map method adds the offsets and a 50-pixel-cycle sine wave to the horizontal coordinate. It's a nonaffine transformation, because it transforms horizontal lines to trigonometrical curves. [Figure 15-27](#) shows the effect of KWave class, another KTransCurve class which adds randomness, and more.

Figure 15-27. Transforming path data using KTransCurve class.



The top-left picture in [Figure 15-27](#) shows the original glyph outline. The top-right picture is the result of sine wave transformer KWave class. Note that the horizontal lines in the bottom of letter "V," top of letter "U," and on letter "E" are all transformed to curves, not just straight lines. The bottom-left picture is generated by applying a small amount of randomness (-3.3) to each transformation candidate. Check the KRandom class on CD for details. The bottom-right picture is our next topic.

3D Text

Now that we've got the outline of a text string and a class to transform each point and break curve to small line segments, it's not hard to try some simple 3D text stuff.

One 3D surface easy to create is called *extruded surface*. Generally speaking, an extruded surface is generated

from a 2D base curve by moving it in a 3D space along a generating path. For example, suppose you have a curve in the 2D $x = y$ plane, put it in a 3D space at $z = 0$, and move the curve along the z -axis to $z = 10$; you've just created an extruded surface that is defined by the trace of the movement.

In the KTransCurve class, line and curve drawing finally goes to a single routine DrvLineTo, which handles a line segment. Given a line segment defined by two end points $(x_1, y_1, 0)$ and $(x_2, y_2, 0)$, the trace of pushing the line along the z -axis for distance "depth" is a rectangle in 3D space defined by its four corners:

$(x_1, y_1, 0), (x_2, y_2, 0), (x_2, y_2, \text{depth}),$ and (x_1, y_1, depth)

The extruded surface in this case is formed by all such rectangles in 3D space; we just need to map every thing back to 2D space. Given a viewer's-eye position (e_x, e_y, e_z) , perspective projection can be used to map a point (x, y, p_z) in 3D space to 2D space.

The partial KExtrude class shown in [Listing 15-12](#) implements the simple drawing of an extruded surface and perspective projection. Complete code is on the CD.

Listing 15-12 Surface Extruding for 3D Text Effects

```
class KExtrude : public KTransCurve
{
    int m_dx, m_dy;
    int m_x0, m_y0;
    int m_depth;
    int m_eye_x, m_eye_y, m_eye_z;
public:
    KExtrude(int seglen, int dx, int dy, int depth, int ex, int ey,
              int ez) : KTransCurve(seglen);
    virtual Map(float x, float y, float & rx, float & ry);
    virtual BOOL DrvBezierTo(HDC hDC, POINT p[]);
    virtual BOOL DrvMoveTo(HDC hDC, int x, int y);

    void Map3D(long & x, long & y, int z)
    {
        x = ( m_eye_z * x - m_eye_x * z ) / ( m_eye_z - z );
        y = ( m_eye_z * y - m_eye_y * z ) / ( m_eye_z - z );
    }

    virtual BOOL DrvLineTo(HDC hDC, int x1, int y1)
    {
        POINT p[5] = { m_x0, m_y0, x1, y1, x1, y1,
                      m_x0, m_y0, m_x0, m_y0 };

        Map3D(p[0].x, p[0].y, 0);
        Map3D(p[1].x, p[1].y, 0);
        Map3D(p[2].x, p[2].y, m_depth);
    }
}
```

```
Map3D(p[3].x, p[3].y, m_depth);
Map3D(p[4].x, p[4].y, 0);
m_x0 = x1;
m_y0 = y1;
return Polygon(hDC, p, 4);
}
};
```

The KExtrude::Map3D method handles perspective projection using the viewer's-eye position defined by (m_eye_x, m_eye_y, m_eye_z). The KExtrude::DrvLineTo method handles drawing the individual rectangles forming the surface after the 3D points are mapped to 2D space. Note that the KExtrude class does not generate results in a new path object; each rectangle is an individual Polygon call. The reason is that GDI's polygon fill modes can't handle some turnings in an extruded surface.

As can be seen in [Figure 15-27](#), the KExtrude class works to generate a simple 3D-text effect. But its simple implementation does not handle hidden-surface removal and light shading, which all need to be handled in a more professional 3D-text generator. We will not cover these topics in this book.

Text as Region

Converting text to a path opens another opportunity, using the rich set of GDI region functions. Given a closed path in a device context, it can either be converted to a region object using PathToRegion, or used for clipping directly using SetClipPath.

```
HRGN PathToRegion(HDC hDC);
BOOL SelectClipPath(HDC hDC, int iMode);
```

PathToRegion converts the current path to a region object, in device coordinate space, which can be used in clipping, hit testing, or other purposes. SelectClipPath is easier but more restrictive, as it uses the path only for clipping. The second parameter, iMode, controls how the path-converted region should be combined with the current clipping region. For example, RGN_AND sets the clipping region to be the intersection of the current clipping region and the path.

Converting a text to a region through a path makes certain operations easier. For example, the easiest way to color a text string with a bitmap is to convert the string to a region and display the bitmap when the region is selected as a clipping region. Here is another implementation of coloring a text with a bitmap, which has no visible flickering:

```
BOOL BitmapText2(HDC hDC, int x, int y, LPCTSTR pString, int nCount,
HBITMAP hBmp)
{
RECT rect;
GetOpaqueBox(hDC, pString, nCount, &rect, x, y);

HDC hMemDC = CreateCompatibleDC(hDC);
HGDIOBJ hOld = SelectObject(hMemDC, hBmp);
```

```
BeginPath(hDC);
SetBkMode(hDC, TRANSPARENT);
TextOut(hDC, x, y, pString, nCount); // generate path
EndPath(hDC);
SelectClipPath(hDC, RGN_COPY); // path->clip region

BOOL rslt = BitBlt(hDC, rect.left, rect.top,
    rect.right-rect.left,
    rect.bottom - rect.top, hMemDC, 0, 0, SRCCOPY);

SelectObject(hMemDC, hOld);
DeleteObject(hMemDC);
return rslt;
}
```

[< BACK](#) [NEXT >](#)

15.7 SUMMARY

Finally, this exciting chapter about Win32 GDI text drawing comes to an end. We've covered a broad range of subjects in lots of depth, from creating logical fonts, querying font metrics information, simple text drawing, more advanced text drawing, text formatting, and various text effects.

We have seen that GDI's limitation in generating WYSIWYG text formatting is due to the integer-based metrics GDI uses. To overcome this limitation, an application can query for precise font metrics information using a reference font whose size is the same as a font's em-square size. As demonstrated in this chapter, using this precise font metrics information, it's possible to format text in a way that is independent of graphic devices and the display zoom ratio.

Special text effects often can be best achieved when the text is converted to either bitmap or outline format. In this chapter, several reusable classes have been developed to wrap the common task of querying the glyph bitmap and outline from GDI, converting a text string to a bitmap, and converting a text string to a GDI path object. A generic class to apply a nonaffine transformation to path data is demonstrated; this class is even capable of doing simple 3D text effects.

With this chapter, we have covered all the major groups of GDI drawing, which are pixels, lines and curves, bitmaps, and final text. The next chapter will talk about how to record GDI commands into meta objects, GDI meta files, and enhanced meta files, which can themselves be manipulated and played back.

[Chapter 17](#) will revisit device-independent text formatting in the context of printing. [Chapter 18](#) will discuss text display in DirectDraw.

Further Reading

Donald Knuth's book *Digital Typography* has several chapters on text formatting. For example, Chapter 3 is titled "Breaking Paragraphs into Lines," Chapter 4 is about "Mixing Right-to-Left Texts with Left-to-Right Texts."

Avery Bishop, David Brown, and David Multzer have written a very informative article titled "Supporting Multilanguage Text Layout and Complex Scripts with Windows NT 5.0." Windows NT 5.0 is none other than Windows 2000. Their article was published in the November issue of *MSJ*, also included with MSDN.

Rod Stephens' book *Visual Basic Graphics Programming* has lots of interesting computer graphics algorithms explained, demonstrated, and coded in Visual Basic. For example, Chapter 10 is about surface, of which the extruded surface in generating 3D fonts is a sample application. His book also covers hidden-surface removal, shading models, and ray tracing, which can be used to generate true and more photorealistic 3D text effects.

Sample Program

There is only one big sample program for this chapter, “[Text](#).” More important than the sample program are the generic and reusable functions and classes developed in this chapter (see [Table 15-5](#)).

Table 15-5. Sample Program for Chapter 15

Directory	Description
Samples\Chapt_15\Text	Under the File menu, there are options to go to the expanded choose font dialog box, PANOSE font matching, TEXTMETRIC analysis dialog boxes, and—most importantly—the demo pages. Once the “Demo” menu item under “File” menu is chosen, there are about 20 different demo pages which cover topics ranging from stock fonts and text terminology all the way to transforming text curves to achieve special text effects.

[< BACK](#) [NEXT >](#)

Chapter 16. Metafile

Applications often need to exchange graphics data with each other, which requires that graphics data be put into a file format. The Windows bitmap file format is good only for raster data exchange. The 16-bit Windows metafile format and the 32-bit Windows enhanced metafile format are graphics file formats designed to support both bitmap and vector graphics data. The Windows metafile and enhanced metafile are widely used in clip art libraries, clipboard, data exchange between the OLE server and the client, and the Windows printer spooler.

This chapter will discuss the two GDI metafile file formats, how to create meta files, how to use metafiles, and how to decode metafiles.

16.1 METAFILE BASICS

Different graphics applications have different strengths and weaknesses, but when an experienced user uses them appropriately, they all contribute in their own ways to achieve a big goal. For example, a user may use CorelDraw to design line art with complex special effects, use PhotoShop for photorealistic image editing, Visio for flowcharts and diagrams, and Word for word processing. When these applications work together, there needs to be a common data format they can all read and write to in order to exchange graphics data.

The Windows bitmap format is good only for bitmaps and photos, although it's not well suited for high-resolution photos because of its lack of good compression. A generic graphics data exchange format needs to support all major elements of graphics, including pixels, lines, curves, filled areas, texts, and bitmaps.

For the 16-bit Windows operating systems, Microsoft designed a Windows-format metafile (WMF). It consists of a sequence of recorded GDI commands, covering all major areas of 16-bit GDI drawing primitives. The Windows-format metafile has its serious limitations in device and application dependency, similar to problems with device-dependent bitmaps. Basically, a Windows-format metafile does not have information about picture dimensions, the original recording device resolution, and the palette. So when it's used on another device with a different color setting and resolution, the application does not have a way to scale it to the proper size and make sure the same set of colors is produced. There are also some other limitations imposed by GDI during the generation of a Windows-format metafile.

For 32-bit Windows operating systems, starting from Windows NT 3.1, Microsoft uses a newer 32-bit metafile format called Enhanced Metafile (EMF). Compared with WMF, EMF supports a 32-bit coordinate system, newer 32-bit GDI functions, and a header with geometric information and a palette, and even provides some support for OpenGL.

Although the WMF is still widely used in clipart libraries, the EMF with its improved device independence and support for newer GDI functions is getting more and more popular. This chapter will focus on the EMF, with some attention paid to the WMF.

There are two closely related concepts about metafiles: a metafile as a GDI object and a metafile as an external file format. Their distinction is similar to a DIB section as a GDI object and a bitmap file in a BMP format, although the connection between a metafile as a GDI object and a physical file is much tighter.

Creating an Enhanced Metafile

An enhanced metafile is a GDI object, similar to a DIB section object, or a path object. As an enhanced metafile object is identified with its GDI object handle, which is of type HENHMETAFILE. The type identifier for an enhanced metafile, as returned by the GetObjectType function, is OBJ_ENHMETAFILE.

An enhanced metafile consists of a sequence of 32-bit GDI commands. So the fundamental way to create an enhanced metafile is by recording a series of GDI commands. Two GDI functions are provided to start and finish the recording of an enhanced metafile.

HDC CreateEnhMetaFile(HDC hdcRef, LPCTSTR lpFileName,

```
CONST RECT * IpRect, LPCTSTR IpDescription);
HENHMETAFILE CloseEnhMetaFile(HDC hDC);
```

Function CreateEnhMetaFile creates a special device context, which will be used to record an enhanced metafile. So CreateEnhMetaFile only sets the stage for creating an enhanced metafile, in the same way BeginPath starts the construction of a GDI path object. The first parameter, hdcRef, refers to a reference device context, whose device information will be queried to record the EMF. If NULL is passed as hdcRef, GDI uses the current display device for reference. The second parameter, lpFileName, can be a disk file name or a NULL pointer. If a valid file name is given, a file version of the metafile is written to disk when construction is complete. If NULL is given, a metafile is memory based.

The third parameter specifies the real-world frame of an enhanced metafile in 0.01-millimeter units, or 10-micrometer units. This rectangle will be stored as the picture frame rectangle in an enhanced metafile, which is the basis for determining the origin and dimensions of an EMF playback. If NULL is passed as the frame rectangle, GDI will calculate a frame rectangle based on the bounding box of all drawing commands, which may not be the same as the intended picture frame rectangle. The following routine converts a rectangle in a device context's logical coordinate space to 0.01-millimeter units.

```
// map a rectangle in logical coordinates to 0.01-mm units
void Map10um(HDC hDC, RECT & rect)
{
    int widthmm = GetDeviceCaps(hDC, HORZSIZE);
    int heightmm = GetDeviceCaps(hDC, VERTSIZE);
    int widthpixel = GetDeviceCaps(hDC, HORZRES);
    int heightpixel = GetDeviceCaps(hDC, VERTRES);

    LPtoDP(hDC, (POINT *) & rect, 2); // map from logical to device

    rect.left = (rect.left * widthmm * 100 + widthpixel / 2) / widthpixel;
    rect.right = (rect.right * widthmm * 100 + widthpixel / 2) / widthpixel;
    rect.top = (rect.top * heightmm * 100 + heightpixel / 2) / heightpixel;
    rect.bottom = (rect.bottom * heightmm * 100 + heightpixel / 2) / heightpixel;
}
```

Note that the function uses HORZSIZE, VERTSIZE, HORZRES, and VERTRES device capability fields to convert the coordinate to physical units, which are used by GDI to fill other fields in the enhanced metafile header structure. But for a screen device, the resolution (pixels per inch) normally does not match the logical resolution reported by the LOGPIXELSX and LOGPIXELSY. For example, LOGPIXELSX and LOGPIXELSY are normally 96 dpi for small-font display modes, and HORZRES is always 320 mm, so if HORZSIZE is 1152 pixels, to an EMF generation the screen is 91.44 dpi.

The last parameter of CreateEnhMetaFile, lpDescription, is an optional text description of the metafile the creator wants to put into a metafile. Its recommended usage is for embedding an application name and a document name in a metafile, with a null character separating them. So if a non-NUL string is given, it must be double-zero terminated.

When the function is successful, CreateEnhMetaFile returns an enhanced meta file device context handle, which can be passed to any GDI function to record it. Once the recording is complete, CloseEnhMetafile should be called to close an enhanced meta file device context handle and return an enhanced metafile handle.

Constructing a metafile is similar to constructing a GDI path object. A metafile is just much more complicated than a point array and a flag array in a GDI path object. So GDI uses a special metafile device context to manage the metafile creation, which is different from putting a device context into a path construction mode for constructing a path.

Here is a simple example of creating an enhanced metafile. The TestEMFGen routine uses the current display as a reference device, and uses a desktop window to calculate the frame rectangle. After a metafile device context is created, a simple rectangle is drawn in the center of the device surface. The routine creates an enhanced metafile GDI object and a disk file with the data for the enhanced metafile. When the metafile is displayed in a 1:1 scale, it should measure 320 by 240 mm with a rectangle in the center.

```
HENHMETAFILE TestEMFGen(void)
{
    RECT rect;

    HDC hdcRef = GetDC(NULL);

    GetClientRect(GetDesktopWindow(), &rect);
    Map10um(hdcRef, rect);
    HDC hDC = CreateEnhMetaFile(hdcRef, "c:\\test.emf", & rect,
        "EMF.EXE\\0TestEMF\\0");
    ReleaseDC(NULL, hdcRef);

    if ( hDC )
    {
        GetClientRect(GetDesktopWindow(), &rect);
        Rectangle(hDC, rect.right/3, rect.bottom/3,
            rect.right*2/3, rect.bottom*2/3);
        return CloseEnhMetaFile(hDC);
    }
    return NULL;
}
```

Playing an Enhanced Metafile

Once an enhanced metafile is created, it can be played back to a device context through the enhanced metafile handle. You can also open an enhanced metafile stored in a file on disk and get back an enhanced metafile handle. Here are the related functions.

```
HENHMETAFILE GetEnhMetaFile(LPCTSTR lpszMetaFile);
BOOL     DeleteEnhMetaFile(HENHMETAFILE hemf);
BOOL     PlayEnhMetaFile(HDC hdc, HENHMETAFILE hemf,
    CONST RECT * lpRect);
```

Function GetEnhMetaFile opens the file named, creates a GDI enhanced meta file object to manage the internal

data for it, and returns an enhanced metafile GDI object handle. When CloseEnhMetaFile finishes constructing an enhanced metafile, a similar handle is returned. After either CloseEnhMetaFile or GetEnhMetaFile, the specified file is being used by GDI and can't be deleted until the GDI object is deleted using DeleteEnhMetaFile.

After an application is done with an enhanced metafile, it should call DeleteEnhMetaFile, in the same way DeleteObject should be called for other GDI objects. Function DeleteEnhMetaFile deletes the GDI enhanced metafile GDI object together with any resource GDI allocates for it. After the call, the physical file on the disk is free from duty, but it's still there. To delete the physical disk file, you have to call the file management function DeleteFile. If NULL is passed as a file name in calling CreateEnhMetaFile, there is no external file to be deleted.

Once you have a valid enhanced metafile handle, you can use PlayEnhMetaFile to play every GDI command embedded in it to a device context. Although the function prototype for PlayEnhMetaFile is quite simple, the internal working of this function is very complicated. PlayEnhMetaFile accepts three parameters: a destination device context handle, a source enhanced metafile handle, and a rectangle. The rectangle is specified in the logical coordinate space of the destination device. It is matched with the frame rectangle specified in calling CreateEnhMetaFile.

Here is a simple example that illustrates the relationship between EMF generation and playing back.

```
void SampleDraw(HDC hDC, int x, int y)
{
    Ellipse(hDC, x+25, y+25, x+75, y+75);

    SetTextAlign(hDC, TA_CENTER | TA_BASELINE);

    const TCHAR * mess = "Default Font";
    TextOut(hDC, x+50, y+50, "Default Font", _tcslen(mess));
}

void Demo_EMFScale(HDC hDC)
{
    // generate EMF
    HDC hDCEMF = CreateEnhMetaFile(hDC, NULL, NULL, NULL);
    SampleDraw(hDCEMF, 0, 0);
    HENHMETAFILE hSample = CloseEnhMetaFile(hDCEMF);

    HBRUSH yellow = CreateSolidBrush(RGB(0xFF, 0xFF, 0));

    // draw the same commands which are put into EMF
    {
        RECT rect = { 10, 10, 10+100, 10+100 };

        FillRect(hDC, &rect, yellow);
        SampleDraw(hDC, 10, 10); // original
    }

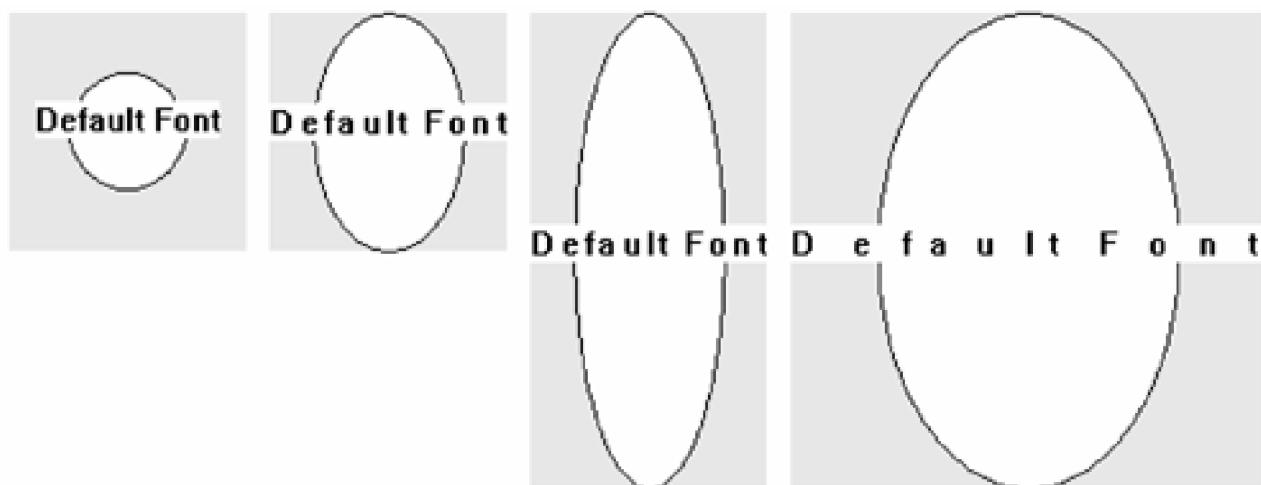
    for (int test=0, x=120; test<3; test++)
    {
        RECT rect = { x, 10, x+(test/2+1)*100, 10+((test+1)/2+1)*100 };
    }
}
```

```
FillRect(hDC, & rect, yellow);
PlayEnhMetaFile(hDC, hSample, & rect);
x = rect.right + 10;
}

DeleteObject(yellow);
DeleteEnhMetaFile(hSample);
}
```

The routine SampleDraw is used to generate the contents of a simple EMF. It's supposed to draw a 50-by-50-unit circle in the middle of a 100-by-100-unit square, and display a text string in the center of the circle using the default font. The Demo_EMF Scale routine controls the whole demonstration. It first creates an in-memory EMF (an EMF without a named disk-file) using the SampleDraw routine. The same SampleDraw routine is used to display what's put into the EMF on the top-left corner of the screen, to act as something to compare with. After that the routine goes through a loop to play the EMF into three rectangles of different sizes, 100 by 100, 100 by 200, and 200 by 200. To help visualize location and dimension, a yellow background is painted over the targeted display regions. [Figure 16-1](#) shows the display result.

Figure 16-1. Playing back an enhanced metafile with a default frame rectangle.



The first picture shows the two drawing commands put into an EMF, a circle with a string in the center of a 100-by-100 square. The second picture shows playing back the EMF into the same size 100-by-100 square. But all the white margins are gone, and the circle and text are scaled nonproportionally. The third and fourth pictures are similar to the second, except they scale rectangles of different size.

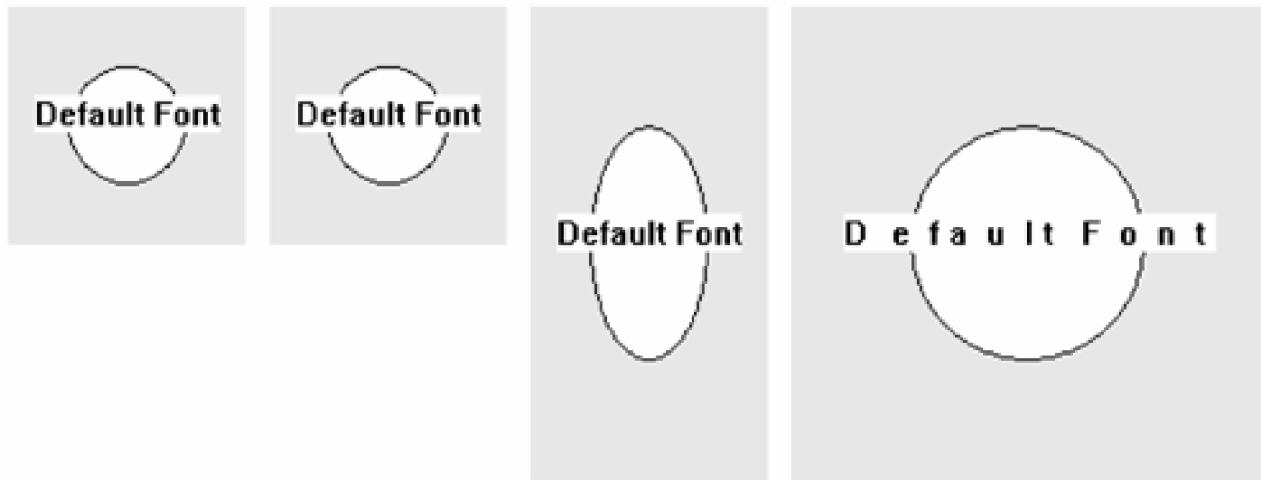
What's happening is that GDI is generating an EMF without a given frame rectangle; it automatically assigns a frame rectangle based on the bounding rectangle of all the drawing calls. In this example, the bounding rectangle is { 11, 25, 88, 74} , and the frame rectangle is { 3.06, 6.94, 24.44, 20.56} (in millimeters). Note that the ratio between width and height is 1.56:1, not 1:1, and all white margins are removed. When playing back the EMF using PlayEnhMetaFile, GDI maps within the frame rectangle to the rectangle given to PlayEnhMetaFile.

To properly set up the frame rectangle in this case, change the call to Create EnhMetaFile to:

```
RECT rect = { 0, 0, 100, 100 }; // logical frame rectangle
Map10um(hDC, rect);           // map to 0.01 mm
hDCEMF = CreateEnhMetaFile(hDC, NULL, & rect, NULL); // create with frame
```

[Figure 16-2](#) illustrates the playing back of an EMF with the proper frame rectangle.

Figure 16-2. Playing back an enhanced metafile with a proper frame rectangle.



[Figure 16-2](#) shows that setting up a 100-by-100 frame rectangle during an EMF generation guarantees that playing back to a 100-by-100 rectangle would generate the extended scale and proportion. The circle is definitely scaled nicely in [Figure 16-2](#). But you may have noticed that the text string is not scaled properly, although each character appears positioned properly; the reason is the use of the default font in a device context. We will cover the details in the next section.

Querying an Enhanced Metafile

We now know that the relationship between the frame rectangle in an EMF and the picture rectangle used by PlayEnhMetaFile is crucial to the positioning and scaling of an enhanced metafile. If you do not know how the EMF is generated, you need a way to query for information like the frame rectangle of an EMF. When generating an EMF, GDI writes an EMF header record that holds important information about it. Several functions are provided for querying for related information.

```
typedef struct tagENHMETAHEADER
{
    DWORD iType;          // EMR_HEADER
    DWORD nSize;          // Size in bytes, including attachment
    RECTL rclBounds;     // Bounds in device units
    RECTL rclFrame;       // Frame rectangle in .01-mm units
    DWORD dSignature;     // ENHMETA_SIGNATURE
    DWORD nVersion;       // Version number
    DWORD nBytes;          // Size of the metafile in bytes
    DWORD nRecords;        // Number of records in the metafile
    WORD nHandles;         // Number of handles in the handle table
```

```
WORD sReserved;
DWORD nDescription; // Chars in UNICODE description string
DWORD offDescription; // Offset to description string
DWORD nPalEntries; // Entries in the metafile palette
SIZEL szlDevice; // Size of reference device in pels
SIZEL szlMillimeters; // Size of reference device in millimeters
DWORD cbPixelFormat; // 4.0 Size of PIXELFORMATDESCRIPTOR
DWORD offPixelFormat; // 4.0 Offset to PIXELFORMATDESCRIPTOR
DWORD bOpenGL; // 4.0 TRUE if OpenGL commands are present
SIZEL szlMicrometers; // 5.0 Reference device size in micrometers
} ENHMETAHEADER;
```

```
UINT GetEnhMetaFileHeader(HENHMETAFILE hemf, UINT cbBuffer,
LPENHMETAFILEHEADER);
UINT GetEnhMetaFileDescription(HENHMETAFILE hemf, UINT cchBuffer,
LPTSTR lpszDescription);
```

A normal EMF always starts with the ENHMETAHEADER structure. The first two fields in the ENHMETAHEADER conform to the generic structure of an EMF record that requires each record to start with a 32-bit record type identifier and a 32-bit size. Each type of EMF record has a unique type identifier, ranging from EMR_MIN to EMR_MAX. Currently EMR_MIN is 1, and EMR_MAX is 122. New EMF record types are constantly being added to EMF to accommodate new GDI features. For example, EMR_ALPHABLEND is 114, which handles the GDI AlphaBlend function, not supported by other systems before Windows 98 and Windows 2000. The nSize holds the total number of bytes in an EMF record, including the iType, nSize, other public fields and possible attachments. Quite a few EMF records are of variable size, so having the nSize field is crucial for GDI to move from one EMF record to the next.

The rclBounds rectangle is the bounding box for all the drawing calls in the EMF, in the device coordinate space. A pair of seldom-used GDI functions manages accumulating the bounding rectangle during drawing to a device context. SetBoundsRect can enable, disable, set, or reset bounding rectangle accumulation; GetBoundsRect can query the accumulated bounding rectangle. Clearly GDI is storing the bounding rectangle gathered during EMF generation in the EMF header. The bounding rectangle can be used by the application to remove white margins around a picture, if it's intended.

The rclFrame rectangle is the frame rectangle the application specifies in calling CreateEnhMetaFile. If NULL is passed for it, GDI automatically calculates it according to the bounding rectangle. The frame rectangle is stored in 0.01-millimeter units, which is equivalent to a 2540-dpi device. The field is crucial for the application to scale an EMF to the indented physical size when playing back on another device.

Following the frame rectangle is some bookkeeping information. Field dSignature should be the unique signature of an enhanced metafile, 0x464d4520 in hexadecimal, or "EMF" in character form. Field nVersion is the version of the EMF being used. Although there are several minor EMF versions being used, the nVersion is found to be always 0x10000. For example, not all metafiles have the three fields for embedding OpenGL, and the last field, szMicrometers, is a new addition for Windows 98 and Windows 2000. But all enhanced metafiles use the same version number. The nBytes field, of course, specifies the total byte length of an EMF. The number of EMF records is kept in the nRecords field, which includes the header record, all the GDI commands, and an end-of-file record.

GDI object handles are handled specially in an EMF. Recall that handles refer to runtime GDI objects which have no meaning across the process address space, not to say to a remote machine at an unknown time. When recording GDI commands involving GDI handles—for example, SelectObject—GDI handles are converted to indexes into a

GDI handle table which exists only during playback of an EMF. When playing back an EMF, GDI creates a table of GDI handles to store the actual handles, and manages translating indexes in the EMF to real GDI handles good for the playing back. A special convention is used to specify GDI stock objects, so only nonstock objects need to be put into this handle table. When a GDI object in EMF is deleted, its slot is freed and can be reused.

The nHandles field in an EMF header specifies the size of the object table GDI needs to create when playing back the metafile. So the nHandles field does not reflect the total number of GDI object handles used by an EMF; instead it reflects the maximum number of open nonstock object handles used by the EMF at a given time. The first entry in the table is reserved to mean white stock brush GDI object, so nHandles is at least one.

The next two fields are for the text description passed to the CreateEnhMetaFile function, which is always stored in a UNICODE format in an EMF. If a description string is given, it's stored after the public fields in the header. The nDescription field specifies the number of characters, while the offDescription field specifies the relative offset from the starting of the header to the description string. If no description string is given, both fields are zero. Recall that there are actually at least three versions of the ENHMETAHEADER structure; the offDescription field can be used to tell their difference, if it's not zero. The oldest version does provide fields for OpenGL embedding; its offDescription field would be 88. The latest version used for Windows 2000 supports OpenGL and the szMicrometers field; its offDescription field is 108.

The nPalEntries field specifies the number of entries in the accumulated palette used in the metafile. The actual color table is stored in the last record of an EMF, not in the header, because when it starts generating the EMF, GDI does not yet know the size of the color table. If no palette is used in the EMF, this field is zero.

The next two fields contain information about the recording reference device. Field szDevice specifies the device surface size in pixels. Field szMillimeters specifies the device surface size in millimeters. GDI calls GetDeviceCaps with HORZRES, VERTRES, HORZSIZE, and VERTSIZE to query for the information from the device driver. For an EMF recorded using a display device context, the typical values for szDevice are 1024 by 768, or 1152 by 864. The value for szMillimeters is always 320 mm by 240 mm (400-mm or 15.75-inch diagonal), even when your monitor is 21 inches.

The next three fields, cbPixelFormat, offPixelFormat, and bOpenGL, are for supporting OpenGL in a metafile format. If the device context is not an OpenGL device context, these three fields are all zeros.

The last field, szMicroMeters, is a new addition for version 5.0 of Win32 API. It specifies the recording device surface size in micrometers, which for the screen is 320,000 by 240,000. It's not certain why it's needed anyway, considering that there is already a device surface size in millimeters.

Given a valid EMF handle, GetEnhMetaFileHeader can be used to query for the EMF header. As the header is a variable-sized structure due to embedded description and OpenGL pixel format information, GetEnhMetaFileHeader should be called twice, first to query for actual size, and the second time for real data. But if you don't care about the attachment, you can just use a single call to query for the fixed members of the header.

Querying the header directly to get the EMF description always gets a UNICODE version of the string. If you don't want to deal with UNICODE in an ANSI program, you can use GetEnhMetaFileDescription. Again, this function could be called twice, first to query for the number of characters, then for the real description string. Recall that the description is usually made up of the company name and document name, separated by the null character. So the whole string is double-null terminated.

When an application displays an arbitrary enhanced metafile, it's important to maintain the same physical size in which it was created. The following routine uses information from the EMF header record to calculate the display rectangle size needed for a given device context.

```
void GetEMFDimension(HDC hDC, HENHMETAFILE hEmf,
    int & width, int & height)
{
    ENHMETAHEADER emfh[5]; // big enough for description
    GetEnhMetaFileHeader(hEmf, sizeof(emfh), emfh);

    // picture size in 0.01 mm
    width = emfh[0].rcIFrame.right - emfh[0].rcIFrame.left;
    height = emfh[0].rcIFrame.bottom - emfh[0].rcIFrame.top;
    // convert to pixels for current device
    int t = GetDeviceCaps(hDC, HORZSIZE) * 100;
    width = ( width * GetDeviceCaps(hDC, HORZRES) + t/2 ) / t;

    t = GetDeviceCaps(hDC, VERTSIZE) * 100;
    height = ( height * GetDeviceCaps(hDC, VERTRES) + t/2 ) / t;

    RECT rect = { 0, 0, width, height };

    // convert to logical coordinate space
    DPtoLP(hDC, (POINT *) & rect, 2);

    width = abs(rect.right - rect.left);
    height = abs(rect.bottom - rect.top);
}
```

The GetEMFDimension routine accepts a destination device context handle where an EMF is supposed to be drawn to an EMF handle. It uses GetEnh Meta File Header to query for the EMF header, from which the size of its frame rectangle is calculated. The width and height of the frame rectangle are converted to the destination device's device coordinate space and then to its logical coordinate space.

Results returned by GetEMFDimension can be used to display an EMF at its original size, or sized to a given ratio. Here is a generic routine for displaying an EMF at a given scale, with 100 by 100 being the original size.

```
BOOL DisplayEMF(HDC hDC, HENHMETAFILE hEmf, int x, int y,
    int scalex, int scaley)
{
    int width, height;
    GetEMFDimension(hDC, hEmf, width, height);

    RECT rect = { x, y,
        x + (width * scalex + 50)/100,
        y + (height * scaley + 50)/100 };

    return PlayEnhMetaFile(hDC, hEmf, & rect);
}
```

Exchanging an Enhanced Metafile

To exchange graphics data with other applications or save for later use, an EMF can be saved to a disk file, read from a disk file, copied to the clipboard, pasted from the clipboard, or attached as a resource to an executable file.

Saving Drawings to an EMF File

The CreateEnhMetaFile function has an option to name a disk file, to which the EMF will be written after the EMF construction is complete. So it's an easy way to save GDI drawings into an EMF. A typical window has a drawing routine that handles displaying graphics on the screen. To add support for saving to an EMF file, an application just needs to add code to open a file-save dialog box to query the user for the file name, set up a frame rectangle, then create an EMF device context, call the same drawing routine to put the drawing into the EMF, and finally close the EMF device context. The following code fragment illustrates this process.

```
HDC QuerySaveEMFFile(const TCHAR * desp,
                      const RECT * rcFrame, TCHAR szFileName[])
{
    KFileDialog fd;

    if ( fd.GetSaveFileName(NULL, "emf", "Enhanced Metafiles") )
    {
        if ( szFileName )
            _tcscpy(szFileName, fd.m_TitleName);

        return CreateEnhMetaFile(NULL, fd.m_TitleName, rcFrame, desp);
    }
    else
        return NULL;
}

int KMyView::OnCommand(int cmd, HWND hWnd)
{
    if ( cmd==IDM_FILE_SAVE )
    {
        hDC = QuerySaveEMFFile("EMF Sample\0", NULL, NULL);

        if ( hDC )
        {
            OnDraw(hDC, NULL); // calling drawing routine
            HENHMETAFILE hEmf = CloseEnhMetaFile(hDC);
            DeleteEnhMetaFile(hEmf); // handle not needed
        }
    }
}
```

The code shows how to handle a menu command IDM_FILE_SAVE in a window. It calls QuerySaveEMFFile to prompt the user to enter a file name for the EMF file to be created and return an EMF device context. After that, it calls the OnDraw method that originally handles painting for the window. The code uses a file dialog box wrapping class developed earlier in the book.

If you have an enhanced metafile handle, it can be saved to a disk file using CopyEnhMetaFile:

```
HENHMETAFILE CopyEnhMetaFile(HENHMETAFILE hemfSrc, LPCTSTR lpszFile);
```

Function CopyEnhMetaFile copies the contents of an EMF to a disk file specified by lpszFile and returns the handle to the new EMF object. If NULL is given as the file name, the copy is made in memory. After the new EMF copy is used, Delete Enh MetaFile should be called to delete the GDI object, but the disk file remains unless DeleteFile is called explicitly.

Creating an EMF object from an existing EMF file is just a call to GetEnhMeta File, which we have already discussed.

Loading an EMF from a Resource

If an EMF is attached as a binary resource to an executable, you can use FindResource, LoadResource, and LockResource to get a pointer to its image and call SetEnhMetaFileBits to create an EMF object from it.

```
HENHMETAFILE SetEnhMetaFileBits(UINT cbBuffer, CONST BYTE * lpData);
```

Function SetEnhMetaFileBits accepts two simple parameters, a metafile size and a pointer to an in-memory metafile. The Win32 functions LoadBitmap and Load Image do not support loading EMF from a resource. So here is a simple routine for loading an EMF from a binary resource.

```
// load an EMF attached as a resource, use RCDATA as resource type
HENHMETAFILE LoadEMF(HMODULE hModule, LPCTSTR pName)
{
    HRSRC hRsc    = FindResource(hModule, pName, RT_RCDATA);

    if ( hRsc==NULL )
        return NULL;
    HGLOBAL hResData = LoadResource(hModule, hRsc);
    LPVOID pEmf    = LockResource(hResData);

    return SetEnhMetaFileBits(SizeofResource(hModule, hRsc),
                           (const BYTE *) pEmf);
}
```

An EMF is not a common resource type, but you can use RCDATA as its resource type, whose runtime resource-type identifier is RT_RCDATA. The LoadEMF routine shows how to load an EMF resource and convert it

to a GDI EMF object.

Putting an EMF on Static Control

An EMF GDI object can be attached to a static control, which can then be automatically displayed in a dialog box or property sheet. This offers a simple way of displaying vector graphics, without using owner-drawn control and adding special code for drawing. Compared to bitmaps and icons, which are commonly used on static controls and buttons, an EMF is more flexible for the scaling required for different screen logical resolution.

To set a static control to display an EMF, set its style to SS_ENHMETAFILE. Or in the resource editor, set a static control's type to "Enhanced Metafile." In the initialization of the control's parent window, send a STM_SETIMAGE message to the control to pass an EMF handle to it. The following code shows how this can be done in a dialog box's message procedure.

```
switch (uMsg)
{
    case WM_INITDIALOG:
    {
        hEmf = LoadEMF((HMODULE) GetWindowLong(hWnd,
            GWL_HINSTANCE), MAKEINTRESOURCE(IDR_EMF1));
        SendDlgItemMessage(hWnd, IDC_EMF, STM_SETIMAGE,
            IMAGE_ENHMETAFILE, (LPARAM) hEmf);
        return TRUE;
    }

    case WM_NCDESTROY:
        if ( hEmf )
            DeleteEnhMetaFile(hEmf);
        return TRUE;
    ...
}
```

[Figure 16-3](#) shows a dialog box with a static control, displaying an EMF generated from one of the text effect demonstration screens in [Chapter 15](#).

Figure 16-3. An EMF resource on static control.



Putting an EMF on a static control offers advantages that the bitmap does not have. With an EMF, you can put lines, curves, area fill, and text on static controls. It's much easier to achieve transparency with an EMF, as shown in [Figure 16-3](#). Another advantage is that the EMF display in a static control gets automatically scaled to the proper size when the screen display switches from a small-font mode to a large-font mode, or vice versa, which is a big pain for bitmap static controls.

Exchanging with the Clipboard

A surprisingly easy and powerful way to exchange graphics data in an EMF format is by using the clipboard. Most graphics applications support the old Windows meta file format; a few of them support an enhanced metafile. When you do a copy from a graphics application, a copy of the graphics data is passed to the clipboard in a WMF or an EMF format. The operating system is smart enough to convert the WMF data to an EMF format, so a client-side application can always query for the EMF data from the clipboard. The clipboard functions are quite easy to use. Here are two routines to support copying and pasting the EMF data to and from the clipboard.

```
void CopyToClipboard(HWND hWnd, HENHMETAFILE hEmf)
{
    if ( OpenClipboard(hWnd) )
    {
        EmptyClipboard();
        SetClipboardData(CF_ENHMETAFILE, hEmf);
        CloseClipboard();
    }
}
```

```
}

HENHMETAFILE PasteFromClipboard(HWND hWnd)
{
    HENHMETAFILE hEmf = NULL;

    if ( OpenClipboard(hWnd) )
    {
        hEmf = (HENHMETAFILE) GetClipboardData(CF_ENHMETAFILE);

        if ( hEmf )
            hEmf = CopyEnhMetaFile(hEmf, NULL);

        CloseClipboard();
    }

    return hEmf;
}
```

The CopyToClipboard routine handles copying an EMF to the clipboard. It uses OpenClipboard to gain access to the clipboard, EmptyClipBoard to remove the existing contents in it, SetClipBoardData to copy data to the clipboard, and finally Close Clip board to release access to the clipboard.

The PasteFromClipboard routine has a similar structure. It uses GetClipboardData to retrieve an EMF handle from the clipboard. But the handle is still owned by the clipboard, so we need to make an in-memory copy of it.

Adding an EMF copy and paste is quite a rewarding experience. Instantly, you can create a chart in Visio and copy to your application, or create word art in Word and copy to your application. You can certainly copy graphics data to the clipboard, and have it pasted to a Word document you're writing, without doing a fixed-resolution screen capture.

[< BACK](#) [NEXT >](#)

16.2 INSIDE AN ENHANCED METAFILE

The basic enhanced metafile features discussed in the last section allow easy creation, displaying, loading, saving, and exchanging with the clipboard of enhanced metafiles. For simple uses of the EMF, this is pretty much what you should know. But the EMF is an important enough tool in GDI that we should look inside an enhanced metafile to understand it fully and make the best use of it.

This section will discuss the design of the Windows enhanced metafile format, converting from GDI commands to an EMF, enumerating an EMF, and dynamically changing an EMF to achieve special effects.

EMF Records

An EMF is a simple sequence of EMF records, having the same general structure. Each EMF record starts with two fixed 32-bit fields, followed by a variable-size parameter unique to that type of record. The EMR structure describes the two fixed 32-bit fields.

```
typedef struct tagEMR
{
    DWORD iType;          // Record type EMR_XXX
    DWORD nSize;          // Record size in bytes
} EMR;
```

The first field, iType, specifies the EMF record type. Each valid EMF record type has a symbolic name and a numeric value defined by GDI using a C macro. Here is a very short list of these macros; refer to the WINGDI.H header file for full details.

```
// Enhanced metafile record types.
#define EMR_HEADER          1
#define EMR_POLYBEZIER       2
#define EMR_POLYGON          3
...
#define EMR_RESERVED_120      120
#define EMR_COLORMATCHTOTARGETW 121
#define EMR_CREATECOLORSPACEW 122

#define EMR_MIN              1
#define EMR_MAX              122
```

If you read WINGDI.H, you will find that new EMF record types are being added for every major OS release. But if a record type is used, it's never changed; only new record types are added. For example, for Windows NT 3, the last defined record type (EMR_MAX) is EMR_POLYTEXTOUTW (97), for Windows NT 4.0, it becomes EMR_PIXELFORMAT (104), and for Windows 2000, it moves up to EMR_CREATECOLORSPACEW (122). There

are even some undocumented record types with names like EMR_RESERVED_120, which may be used by the printer spooler for printing.

Each nonreserved EMF record has a defined structure in WINGDI.H and is documented in Microsoft documentation. These structures can be seen as derived from the generic EMR structure. Here is an example—EMRSETPIXELV for the SetPixelV function.

```
typedef struct tagEMRSETPIXELV
{
    EMR    emr;
    POINTL ptlPixel;
    COLORREF crColor;
} EMRSETPIXELV, *PEMRSETPIXELV;
```

An EMF record may have extra attachments beyond the public members defined in the record structure. For example, a bitmap drawing EMF record has bitmap information and a pixel array as attachments. For each attachment, there are usually two fields that specify the offset from the starting of a record to the data, and the byte length of the data. The description string in the ENHMETAHEADER structure is such an example. This simple and uniform design makes attaching, accessing, and parsing variable-size data much easier.

The second field in the EMR structure is the total byte size of an EMF record, which includes two fixed fields, other public fields, and any attachment. EMF records are always double-word aligned.

A minimum EMF has two records, a header and an end-of-file record. We've already seen the header ENHMETAHEADER. Here is the end-of-file record.

```
typedef struct tagEMREOF
{
    EMR    emr;
    DWORD  nPalEntries; // Number of palette entries
    DWORD  offPalEntries; // Offset to the palette entries
    DWORD  nSizeLast; // Last DWORD
} EMREOF, *PEMREOF;
```

Besides signaling the end of an EMF, the EMREOF record contains accumulated palette entries used in the EMF. One strange thing is that the palette attachment is inserted after the offPalEntries field and before the nSizeLast field.

If you want to get a real feeling of an EMF, here is an annotated binary dump of a simple EMF with a single SetPixelV command.

```
// EMRMETAHEADER
00 iTyp      1
04 nSize     0x84
08 rclBounds { 3, 5, 3, 5 }
18 rclFrame  { 0, 0, 0x49BB, 0x2311 }
```

```
28 dSignature    ' EMF'
2c nVersion     0x10000
30 nBytes       0xAC
34 nRecords     3
38 nHandles     1
3A sReversed    0
3C nDescription 0xC
40 offDescription 0x6C
44 nPalEntries   0
48 szlDevice     { 0x500, 0x400 }
50 szlMillimeters { 0x140, 0xF0 }
58 cbPixelFormat 0
5C offPixelFormat 0
60 bOpenGL       0
64 szlMicrometers { 0x4E200, 0x3A980 }
6C Description   L'EMF Sample\0\0'
```

// EMRSETPIXELV

```
84 iTyp        0xF
88 nSize      0x14
8C ptlPixel   { 3, 5 }
94 crColor    RGB(0xFF, 0, 0)
```

// EMREOF

```
98 iTyp        0x0E
9C nSize      0x14
A0 nPalEntries 0
A4 offPalEntries 0x10
A8 nSizeLast   0x14
```

When an EMF is written to a disk file, or attached as a resource, it has exactly the same structure, nothing hidden and nothing extra.

Classifying EMF Record Types

We know an EMF is a recording of GDI commands, but the exact mapping from the GDI commands to an EMF is not documented. On Windows 2000, GDI32.DLL exports 543 functions. Discounting some of them that are for user-mode printer driver support only, there is still roughly a 4:1 ratio between the exported GDI functions and the 122 known EMF record types.

The starting point to understand the mapping from the GDI commands to an EMF record type is classifying the EMF records into several major categories. [Table 16-1](#) divides the EMF records into 12 categories.

By comparing the categorized EMF record types listed in [Table 16-1](#) with the Win32 API functions from the same category, you can figure out the decisions and tradeoffs the EMF designer had to make when designing the EMF. Here are some of my observations.

Table 16-1. EMF Record Type Classification

Category	EMF Record Types
GDI Objects	EMR_CREATEBRUSHINDIRECT, EMR_CREATEDIBPATTERNBRUSH PT, EMR_CREATEMONOBRUSH, EMR_CREATEPALETTE, EMR_CREATEPEN, EMR_EXTCREATEFONTINDIRECTW, EMR_EXT CREATE PEN, EMR_DELETEOBJECT, EMR_RESIZEPALETTE, EMR_SETPALETTEENTRIES
Device Context	EMR MODIFYWORLDTRANSFORM, EMR_REALIZEPALETTE, EMR_RESTOREDC, EMR_SAVEDC, EMR_SCALEVIEWPORTEXTEX, EMR_SCALEWINDOWEXTEX, EMR_SELECTOBJECT, EMR_SELECTPALETTE, EMR_SETARCDIRECTION, EMR_SETBKCOLOR, EMR_SETBKMODE, EMR_SETBRUSHORGEX, EMR_SETLAYOUT, EMR_SET MAP MODE, EMR_SETMAPPERFLAGS, EMR_SETMITERLIMIT, EMR_SETPOLYFILLMODE, EMR_SETROP2, EMR_SETSTRETCHBLTMODE, EMR_SETTEXTALIGN, EMR_SETTEXTCOLOR, EMR_SETVIEWPORTEXTEX, EMR_SETVIEWPORTORGEX, EMR_SETWINDOWEXTEX, EMR_SETWINDOWORGEX, EMR_SETWORLDTRANSFORM
Clipping	EMR_EXCLUDECLIPRECT, EMR_EXTSELECTCLIPRGN, EMR_INTERSECTCLIPRECT, EMR_OFFSETCLIPRGN, EMR_SELECTCLIPPATH, EMR_SETMETARGN
Path	EMR_ABORTPATH, EMR_BEGINPATH, EMR_CLOSEFIGURE, EMR_ENDPATH
ICM	EMR_CREATECOLORSPACE, EMR_CREATECOLORSPACEW, EMR_COLORCORRECTPALETTE, EMR_COLORMATCHTOTARGETW, EMR_DELETECOLORSPACE, EMR_SetCOLORADJUSTMENT, EMR_SetCOLORSPACE, EMR_SetICMMODE, EMR_SetICMPROFILEA, EMR_SetICMPROFILEW
Lines and Curves	EMR_ANGLEARC, EMR_ARC, EMR_ARCTO, EMR_FLATTENPATH, EMR_LINETO, EMR_MOVETOEX, EMR_POLYBEZIER, EMR_POLYBEZIER16, EMR_POLYBEZIERTO, EMR_POLYBEZIERTO16, EMR_POLYDRAW, EMR_POLYDRAW16, EMR_POLYLINE, EMR_POLY LINE16, EMR_POLYLINETO, EMR_POLYLINETO16, EMR_POLYPOLYLINE, EMR_POLYPOLYLINE16, EMR_STROKEPATH, EMR_WIDENPATH
Area Fill	EMR_CHORD, EMR_ELLIPSE, EMR_FILLPATH, EMR_FILLRGN, EMR_FRAMERGN, EMR_GRADIENTFILL, EMR_INVERTRGN, EMR_PAINT RGN, EMR_PIE, EMR_POLYGON, EMR_POLYGON16, EMR_POLYPOLYGON, EMR_POLYPOLYGON16, EMR_RECTANGLE, EMR_ROUND RECT, EMR_STROKEANDFILLPATH
Bitmap	EMR_ALPHABLEND, EMR_BITBLT, EMR_EXTFLOODFILL, EMR_MASKBLT, EMR_PLGBLT, EMR_SETDIBITSTODEVICE, EMR_SET PIXELV, EMR_STRETCHBLT, EMR_STRETCHDIBITS, EMR_TRANSPARENTBLT
Text	EMR_EXTEXTOUTA, EMR_EXTEXTOUTW, EMR_POLYTEXTOUTA, EMR_POLYTEXTOUTW
OpenGL	EMR_GLSBOUNDEDRECORD, EMR_GLSRECORD, EMR_PIXEL FORMAT
Undocumented	EMR_RESERVED_105, EMR_RESERVED_106, EMR_RESERVED_107, EMR_RESERVED_108, EMR_RESERVED_109, EMR_RESERVED_110, EMR_RESERVED_119, EMR_RESERVED_120
Miscellaneous	EMR_EOF, EMR_GDICOMMENT, EMR_HEADER

- An EMF is a constant-only graphics data format (or you may call it a language). Everything contained in the EMF is constant, which means there are no variable expressions, no direct GDI handles, and no dependency on the result of a function call.
- All computation and querying are executed during the EMF generation, and only the result is put into the

EMF. As such, there are no EMF records for GDI querying functions like GetDeviceCaps, GetBkMode, GetBkColor, etc. The generation of an EMF is not a purely simple recording processing; it's a mix of execution and recording. If your drawing code contains conditional branches, or relies on function return values, a snapshot is taken for the reference device context at the time of generation. There is no guarantee that playing back a recorded EMF generates the exact same result as the original drawing code.

- For GDI objects, only the pen, brush, font, palette, and color space (for ICM) are encoded as GDI objects, meaning that they have the object creation, selection, deletion EMF records. A path is still an implicit GDI object, but there is no EMF record for querying its data using GetPath, because it would involve variables. Heavy-weight GDI objects—device-dependent bitmaps (DDB), a DIB section, region, memory device context, and enhanced meta file—are not recorded as GDI objects in an EMF. For example, there is no EMF record for creating a DDB, or an EMF embedded within an EMF. As we will see later, the actual complete data of a DDB, DIB section, or region is embedded as an attachment to EMF records for drawing commands that use it. Function calls involving a memory device context, or other device context besides the destination device context, the enhanced metafile is just executed, instead of recorded.
- Drawing functions provided by the window management module (USER32.DLL) are not directly put into an EMF. For example, you can't find Draw Edge, DrawFrameControl, DrawCaption, DrawIcon, DrawText, etc., in [Table 16-1](#). Some of these functions rely on GDI for actual drawing; their GDI calls will be recorded in an EMF. But others use direct system service calls that bypass the user mode portion of GDI, where the EMF recording is actually implemented. For example, there is a USER module system call, Nt UserDrawCaption. Drawing calls that bypass the user mode GDI module can't be recorded in the EMF.
- The EMF supports OpenGL through special information in the header and three special EMF record types. But DirectX is not supported in the EMF.
- Printing commands are not supported in an EMF, although GDI and a spooler can silently record a printing job into an EMF spool file and play it back to the device driver later.

Decoding EMF Records

Given an EMF handle, you can call the GDI function GetEnhMetaFileBits to retrieve all the EMF records, which can then be walked through on your own. GetEnhMeta File Bits is defined as:

```
UINT GetEnhMetaFileBits(HENUMETAFILE hEmf, UINT cbBuffer,  
LPBYTE lpbBuffer);
```

An application should call GetEnhMetaFileBits with an empty buffer first to query for the EMF size, and then the actual data. The following routine shows how to walk through every record in an EMF.

```
int DumpEMF(HENHMETAFILE hEmf, ofstream & stream)  
{  
    int size = GetEnhMetaFileBits(hEmf, 0, NULL);  
    if ( size<=0 )  
        return 0;  
  
    BYTE * pBuffer = new BYTE[size];
```

```
if ( pBuffer==NULL )
    return 0;

GetEnhMetaFileBits(hEmf, size, pBuffer);
const EMR * emr = (const EMR *) pBuffer;

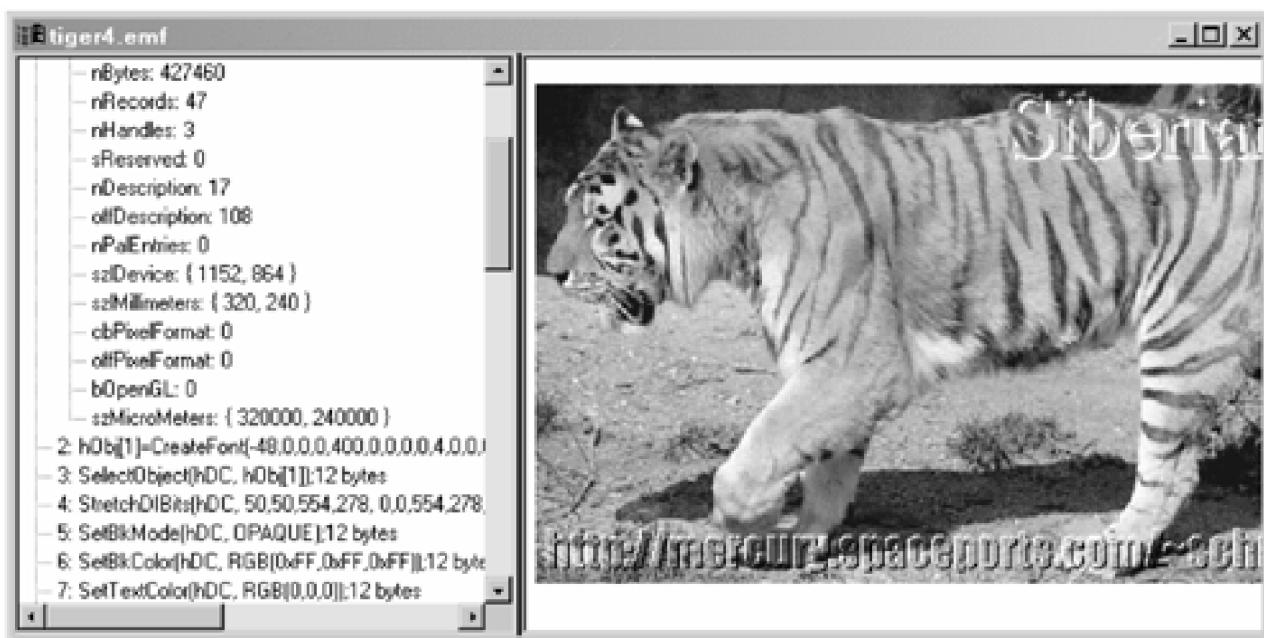
TCHAR mess[MAX_PATH];
int recno = 0;

// enumerate all EMF records
while ( (emr->iType>=EMR_MIN) && (emr->iType<=EMR_MAX) )
{
    recno++;
    wsprintf(mess, "%3d: EMR_%03d (%4d bytes)\n", recno,
             emr->iType, emr->nSize);
    stream << mess;
    if ( emr->iType== EMR_EOF )
        break;
    emr = (const EMR *) ( ( const char * ) emr + emr->nSize );
}

delete [] pBuffer;
return recno;
}
```

The DumpEMF routine walks through every record in an EMF and dumps the sequence number, type, and size information for each record to the C++ file stream. The Dump EMF is just an illustration of how walking through an EMF can help you understand the internal structure of the EMF. In the sample program developed for this chapter, which is on the CD, there is a sophisticated EMF analyzing tool, implemented by the class KEmfDC. The KEmfDC class can dump the contents of an EMF into a TreeView, with the EMF record decoded into the GDI commands and parameters. [Figure 16-4](#) shows a window with a decoded EMF on the left and a visual display of the EMF on the right.

Figure 16-4. An EMF decoder and viewer in the sample program.



[Figure 16-4](#) shows a sample screen with an EMF picture showing text embossing, which is generated by [Chapter 15](#)'s sample program. This EMF viewer and decoder is part of the sample program for this chapter. Opening a disk EMF file, or pasting an EMF from the clipboard, the program can decode and display it. What's shown on the left is part of the EMF header, and five decoded GDI commands. The EMF header shows that this EMF has 47 records, 427,460 bytes long, and is captured on an 1152-by-864-pixel screen. The GDI commands show logical font creation, selection, bitmap drawing, and setting the background mode/color to prepare for other drawing. The right part of the window shows a 1:1 display of the metafile.

Simple GDI Objects in an EMF

This EMF decoder and viewer shown in [Figure 16-4](#) is a valuable tool to understand how an EMF works and to analyze problems related to using an EMF. Now let's look at some details about EMF design.

Only four types of common GDI objects are stored as GDI objects in an EMF; they are logical pen, logical brush, logical font, and logical palette. The object creation functions for these object types have a similar structure in EMR records, in which the parameters start with an index to the EMF handle table followed by a logical definition for the object. As an example, here is the EMF record structure for CreatePen.

```
typedef struct tagEMRCREATEPEN {
    EMR    emr;    // standard header
    DWORD  ihPen;   // index to handle table
    LOGPEN lopn;   // logical definition
} EMRCREATEPEN;
```

When playing back an EMF, GDI creates a small EMF handle table according to the nHandles field in the EMR header record. The index in the EMRCREATEPEN record refers to this EMF handle table, not the hidden systemwide GDI object table. The first entry in the EMF handle table is reserved. The EMF handle table is described by the HANDLETABLE structure.

```
typedef struct tagHANDLETABLE {  
    HGDIOBJ objecthandle[1]; // variable size, determined by nHandles  
} HANDLETABLE;
```

Stock GDI objects are not stored in the EMF handle table. When a stock object is referenced, the negation of its identifier is used as its index. Currently the documented stock GDI objects are GetStockObject(WHITE_BRUSH) to GetStock Object(DC_PEN). GDI32.H defines WHITE_BRUSH as 0 and DC_PEN as 19, so their indexes in the EMF are 0 and -19, respectively. This also explains why index 0 in the EMF handle table is reserved.

For an EMF, records need to reference logical pen, brush, font, or palette; for example, for EMRSELECTOBJECT for SelectObject, only the index into the EMF handle table is used. The example in [Figure 16-4](#) shows that the second EMF record creates a logical font and puts it into entry 1 in the EMF handle table; the third EMF record selects entry 1 in the table to the device context.

The use of the simple EMF handle table elegantly avoids the evasive, non-deterministic nature of GDI object handles. But converting GDI handles into indexes is not as simple as it may appear to be. First, GDI can't simply record the GDI object creation routine, because it may not be used by the recording device context at all. It can create an object creation record only when a certain handle is actually being used the first time. This means every time a pen, brush, palette, or font handle is used, GDI needs to search the EMF handle table to find if the handle is already recorded. If the handle is new, GDI can use GetObject to retrieve the original definition used in creating the handle to generate an object creation record; otherwise, an index is found.

DeleteObject is also recorded in the EMF using the EMRDELETEOBJECT record. Of course, it should be recorded only if the handle is actually to be used.

When done playing an EMF, there may still be handles in the EMF handle table which are not yet deleted. GDI guarantees the table will be cleaned up to eliminate any GDI object leakage due to EMF playback, although it would be a bug in the recording application. An application can check the number of handles in the EMF handle table and also check what's left undeleted after finishing playing an EMF to make sure there is no possible resource leak.

Bitmaps in an EMF

GDI supports three types of bitmaps: a device-dependent bitmap (DDB), a device-independent bitmap (DIB), and a DIB section. A DIB is not a GDI object, in the sense that GDI does not manage its storage. But DDBs and DIB sections are GDI objects. But there are no DDB/DIB section, GDI object creation records in an EMF.

A DDB by nature is device-dependent and even device-setting dependent. So it certainly can't be saved directly in an EMF, which is supposed to be a device-independent representation of graphics data. Another trouble with a DDB is that it cannot be drawn directly into a device context; a memory device context needs to be involved. These may be the reasons why the GDI designer decided not to treat a DDB or a DIB section as a GDI object in an EMF. Instead, a DDB or a DIB section is always converted to an unpacked DIB.

In GDI, an unpacked DIB is represented by two pointers—one to its BITMAPINFO structure and one to the pixel bits. Without GDI object handles for bitmaps, there is no existing way to reference bitmaps in an EMF; somehow it was decided that bitmaps should be attached to drawing calls using them.

Let's look at how the GDI function BitBlt is encoded as an EMF record EMR BITBLT.

```
typedef struct tagEMRBITBLT
{
    EMR     emr;
    RECTL   rclBounds; // Inclusive-inclusive bounds in device units
    LONG    xDest;
    LONG    yDest;
    LONG    cxDest;
    LONG    cyDest;
    DWORD   dwRop;
    LONG    xSrc;
    LONG    ySrc;
    XFORM   xformSrc; // Source DC transform
    COLORREF crBkColorSrc; // Source DC BkColor in RGB
    DWORD   iUsageSrc; // Source bitmap info color table usage
                      // (DIB_RGB_COLORS)
    DWORD   offBmiSrc; // Offset to the source BITMAPINFO structure
    DWORD   cbBmiSrc; // Size of the source BITMAPINFO structure
    DWORD   offBitsSrc; // Offset to the source bitmap bits
    DWORD   cbBitsSrc; // Size of the source bitmap bits
} EMRBITBLT;
```

Recall that the GDI function BitBlt has nine parameters: one destination device context handle, four parameters for a destination rectangle, one source memory device context handle, two parameters for the source origin, and finally, one raster operation. In the EMRBITBLT structure, a destination device context is not needed because it's implicit; a destination rectangle is represented by {xDest, yDest, cxDest, cyDest}, the source origin is represented by { xSrc, ySrc} , and the raster operation is dwRop. Now we only have the source device context handle parameter unaccounted for, but eight fields in the EMRBITBLT structure to explain.

The source device context in BitBlt could be a memory device context backed by a DDB or a DIB section, or a display device context. It's not likely that it could be a printer device context, because printer device contexts are usually not readable. A memory device context can be represented using the bitmap it's holding, and a display device context can be converted to a bitmap formed by its current pixels. Anyway, it can be converted to a DIB. This DIB is represented by the last parameters in the EMRBITBLT, iUsageSrc for color table interpretation, (offBmiSrc, cbBmiSrc) for BITMAP INFO, and (offBitsSrc, cbBitsSrc) for the actual bits. The state of the source device context could also affect BitBlt drawing. The xformSrc field captures any possible logical-to-device-coordinate mapping in the source coordinate space. The cbBkColorSrc parameter is the background color of the source DC, which may be used in drawing a color bitmap to a monochrome device context.

The rclBounds is the bounding rectangle in the EMF recording device's device coordinate space. Although it can certainly be calculated during runtime, saving it in EMR BITBLT may have some performance benefit.

All bitmaps in the EMF are represented the sample way: a color table usage flag, a BITMAPINFO attachment, and a bitmap pixel attachment. The problem with this representation is that a bitmap is stored every time it's used. If the same bitmap is used 100 times, it's stored 100 times in an EMF. While this may be not be a problem for small bitmaps and for bitmaps used only once, for more advanced graphic applications that frequently use bitmap fills it could be a huge problem.

According to a test with a sample program that draws the sample bitmap multiple times, a full bitmap is embedded each time into the EMF. The size of the EMF is proportional to the number of times a bitmap is used. GDI also allows part of a bitmap to be drawn in a drawing call. It seems that when a horizontal strip of bitmap is drawn, GDI is

nice enough to trim the embedded bitmap to a smaller size; but in more complicated cases, GDI just embeds the whole bitmap.

As bitmaps with high color depth become easily available through the Internet and digital camera, and considering that most printer drivers take advantage of EMF spooling, bitmap handling in EMF could become a performance problem that application programmers have to deal with more and more often. The EMF decoder and viewer shown in [Figure 16-4](#) displays the size of each record and the dimension of each bitmap to help identify size-related problems.

A more interesting question is, given the current general EMF design, how could the problem be fixed with minimum changes to GDI, or even better, could the application do something to control the size of the EMF? We know that an attachment like a bitmap in the EMRBITBLT record is always put after the public fields. But a reference to such an attachment is through a 32-bit offset. If we allow the offsets to be negative, or go beyond the current EMF record, multiple EMF records can share the same bitmap. Another idea is to just add a bitmap object creation record to the EMF, such that a unique bitmap is embedded only once in the EMF, and all the references use indexes just as for pens and brushes.

Regions in an EMF

Regions are handled much like bitmaps in the EMF. There is no region GDI handle, region creation and pure region operations are executed, instead of recorded. When a region is used on the recording device context, the full region data is embedded with the EMF record using it.

Here is an example—the EMREXTSELECTCLIPRGN record that handles both SelectClipRgn and ExtSelectClipRgn.

```
typedef struct tagEMREXTSELECTCLIPRGN
{
    EMR    emr;
    DWORD  cbRgnData;      // Size of region data in bytes
    DWORD  iMode;
    BYTE   RgnData[1];
} EMREXTSELECTCLIPRGN;
```

As can be seen from the EMREXTSELECTCLIPRGN record, the region data is embedded into the record, even without the usually offset field. The variable-size array RgnData actually uses the RGNDTA structure, which is returned by GetRegionData function.

As with bitmaps, when the same region is used repetitively, multiple copies of region data are stored in the EMF. Care should be taken when an application uses complicated regions.

Paths in an EMF

The EMF's handling of a path is quite close to GDI functions. BeginPath, CloseFigure, End Path, and the path drawing functions are all supported by corresponding EMF records. So a path is still implemented as an implicit GDI object. SaveDC and RestoreDC are also supported in the EMF. So you can use them to make multiple usage of the same path.

SelectClipPath is also supported by an EMF record, which also reduces the need to embed path data into the EMF. For example, if an application wants to use an elliptical clip region, instead of using CreateEllipticRgn and SelectRegion, it could use Begin Path, Ellipse, EndPath, and SelectClipPath to avoid embedding region data into the EMF.

If you use GetPath to query for path data, then call PolyDraw to draw the path; path data is embedded with the EMRPOLYDRAW record.

Pallettes in an EMF

Palette-related GDI functions, such as CreatePalette, SelectPalette, ResizePalette, and RealizePalette, are recorded using separate EMF records. A logical palette is treated as a GDI object in the EMF.

But GDI changes the interpretation of SelectPalette in the EMF. Recall that SelectPalette has three parameters: a device context handle, a palette handle, and a flag. The flag controls whether the palette should be forced as a background palette. If the window drawn into is a foreground window and the bForceBackground parameter to SelectPalette is FALSE, the palette will later be realized as a foreground palette. The difference between a foreground palette and a background palette is that only a foreground palette can remove nonstatic color entries from the system palette, such that more colors can be realized as unique solid colors to generate an accurate color reproduction; a background palette can use only unoccupied entries in the system palette, or match with existing entries. In the EMF record for SelectPalette, which is EMRSELECTPALETTE, the bForceBackground parameter is not recorded. When an EMF is played back, a logical palette is always selected as a background palette, never a foreground palette.

The rationale behind this background-palette-only design is that playing back an EMF should not affect the colors currently displayed on the screen; otherwise, the screen display could become really flickering and ugly. The foreground palette should not be controlled at the EMF playback level; it should be controlled at the higher window message-handling level—that is, when handling WM_PAINT and WM_PALETTECHANGED messages.

If an application really wants the colors in an EMF to be in the current system palette—that is, realizing the palette as a foreground palette—GDI provides a complete color table in the end-of-file EMR record EMREOF. During the EMF recording, GDI accumulates palette entries and places them in a “metafile palette,” which becomes part of the EMREOF record. An application can use the GetEnhMetaFile PaletteEntries function to query for a metafile palette.

```
UINT GetEnhMetaFilePaletteEntries(HENHMETAFILE hemf, UINT cEntries,  
    LPPALETTEENTRY lppe);
```

Now think for a moment, how is this function implemented in GDI? From an EMF handle, GDI can surely get to the EMF header record, but how can GDI get to the end-of-file record when there is no direct reference to it? Walking through the whole EMF would touch every record and take a long time if the EMF needed to be read into memory or swapped into physical memory. The trick is in the last field in the EMR EOF record, nSizeLast. Recall that nSizeLast is after the actual color table. From the EMF header, GDI can locate the address of the nSizeLast field, because the total size of an EMF is recorded in the header. The nSizeLast field holds the size of the EMR EOF record, which can be used to locate the starting point of the EMREOF record.

The full details about palette handling are covered in [Chapter 13](#) of this book. The following routine shows how to create a logical palette from an EMF.

```
HPALETTE GetEMFPalette(HENHMETAFILE hEmf, HDC hDC)
{
    // query number of entries
    int no = GetEnhMetaFilePaletteEntries(hEmf, 0, NULL);
    if ( no<=0 )
        return NULL;

    // allocation
    LOGPALETTE * pLogPalette = (LOGPALETTE *) new
        BYTE[sizeof(LOGPALETTE) + (no-1) * sizeof(PALETTEENTRY)];

    pLogPalette->palVersion = 0x300;
    pLogPalette->palNumEntries = no;

    // get real data
    GetEnhMetaFilePaletteEntries(hEmf, no, pLogPalette->palPalEntry);

    HPALETTE hPal = CreatePalette(pLogPalette);

    delete [] (BYTE *) pLogPalette;
    return hPal;
}
```

An application can use the logical palette returned by GetEMFPalette to realize a foreground palette if it's going to be displayed in a foreground window, and respond to palette messages. One final note about a palette and an EMF: when you use Play Enh Meta File to play an EMF, GDI resets the device context to its default state before actually playing, which will later be restored. So you can't get the EMF to use the logical palette selected into the device context before playing the EMF.

Coordinate Spaces in an EMF

When playing an EMF, two device contexts are involved, the *recording device context* and the *destination device context*. The recording device context is the one used to generate the EMF, the destination device context is the one that PlayEnhMetaFile is playing the EMF into. Although the recording device context does not visibly exist outside GDI, conceptually it's used by the coordinates kept in the EMF records.

Each device context has a logical coordinate space and a device coordinate space, so at least four coordinate spaces are involved:

- A logical coordinate space on the recording device
- A device coordinate space on the recording device
- A logical coordinate space on the destination device
- A device coordinate space on the destination device

We say at least four, not just four, because multiple logical coordinate spaces may be involved in the recording device. Setting up and changing logical coordinate spaces using SetWindowExtEx, SetViewportExtEx, SetWorldTransform, etc., is fully supported in the EMF.

Now here is the confusing part: When an EMF record draws a one-inch line using an English mapping mode (MM_LOENGLISH or MM_HIENGLISH), how long is the line when PlayEnhMetaFile plays the EMF? A trickier question is, when you select a clip region which is supposed to be in the device coordinate space, how is it interpreted when drawing the EMF?

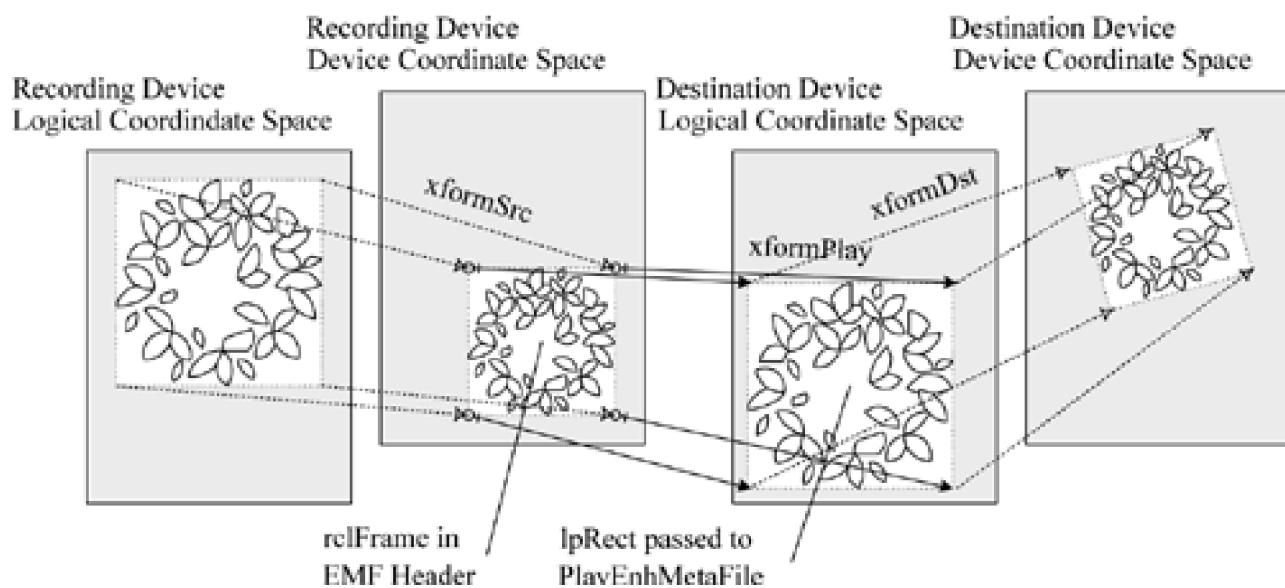
When playing an EMF, GDI can interpret coordinate-space-related EMF records to map everything in the recording device's logical coordinate space to its device coordinate space. Let's name its transformation matrix *xformSrc*.

Similarly, everything in the destination device's logical coordinate space can be mapped to its device coordinate space. Let's name its transformation matrix *xformDst*.

So the key to coordinate-space mapping in playing back an EMF is the connection between the recording device's device coordinate space and the destination device's logical coordinate space. Let's name the transformation matrix for this mapping *xformPlay*.

[Figure 16-5](#) illustrates the two device contexts, four coordinate spaces, and three transformations in playing an EMF.

Figure 16-5. Coordinate spaces and transformations in playing an EMF.



The transformation matrix *xformPlay* can be calculated from the frame rectangle kept in the EMF header record (*rclFrame*) and the picture rectangle passed to Play Enh MetaFile (*lpRect*). The only minor problem is we have to convert *rclFrame* from 0.01-millimeter units to the recording device's device coordinate space. The following routine shows how to calculate this transformation matrix.

```
// Transformation to map recording device's device coordinate space  
// to destination device's logical coordinate space  
BOOL GetPlayTransformation(HENHMETAFILE hEmf, const RECT * rcPic, XFORM & xformPlay)
```

```
{  
ENHMETAHEADER emfh;  
  
if ( GetEnhMetaFileHeader(hEmf, sizeof(emfh), & emfh)<=0 )  
    return FALSE;  
  
try  
{  
// frame rectangle in 0.01 mm->1 mm-> percentage-> device pixels  
double sx0 = emfh.rclFrame.left / 100.0 /  
    emfh.szMillimeters.cx * emfh.szDevice.cx;  
double sy0 = emfh.rclFrame.top / 100.0 /  
    emfh.szMillimeters.cy * emfh.szDevice.cy;  
double sx1 = emfh.rclFrame.right / 100.0 /  
    emfh.szMillimeters.cx * emfh.szDevice.cx;  
double sy1 = emfh.rclFrame.bottom / 100.0 /  
    emfh.szMillimeters.cy * emfh.szDevice.cy;  
  
// source to destination ratio  
double rx = (rcPic->right - rcPic->left) / ( sx1 - sx0 );  
double ry = (rcPic->bottom - rcPic->top) / ( sy1 - sy0 );  
  
// x' = x * eM11 + y * eM21 + eDx  
// y' = x * eM12 + y * eM22 + eDy  
xformPlay.eM11 = (float) rx;  
xformPlay.eM21 = (float) 0;  
xformPlay.eDx = (float) (- sx0 * rx + rcPic->left);  
  
xformPlay.eM12 = (float) 0;  
xformPlay.eM22 = (float) ry;  
xformPlay.eDy = (float) (- sy0 * ry + rcPic->top);  
}  
catch (...)  
{  
    return FALSE;  
}  
    return TRUE;  
}
```

Now, instead of having four coordinate spaces, we have three transformation matrices to deal with when playing back an EMF. The destination transformation matrix xformDst is set up by the application in the destination device context before playing the EMF. The source transformation matrix xformSrc starts with an identity matrix and dynamically changes when the coordinate-system-related EMF records are played. Between them is the xformPlay transformation matrix that is determined by the EMF header and the parameter passed to PlayEnhMetaFile.

Normally, you don't have to care about the destination transformation matrix, because when you draw in the destination surface, GDI will do the coordinate transformation on logical coordinates. Every logical coordinate in the EMF needs to go through the source transformation matrix xformSrc and then the xformPlay transformation matrix before it can be drawn into the destination surface.

Certain coordinates in the EMF are meant to be in the recording device's device coordinate space—for example, the coordinates in a clip region. These coordinates need to be translated to the destination device's device coordinate space. GDI can use the xformPlay matrix followed by the xformDst matrix to do the mapping. Recall that all the region data in the EMF are in the RGNDATA structure, which can be converted to a GDI region object using the ExtCreateRegion function. The ExtCreateRegion function is conveniently designed to accept a transformation matrix that will be applied to all the coordinates, and two transformation matrices can be combined into one using the combine CombineTransform function.

In dealing with the EMF, the so-called device coordinate space is really relative, in that it can be mapped to a rectangle in another logical coordinate space. Because of the nature of scaling and mapping involved with the EMF, the resolution of the recording logical coordinate space plays a big role in graphics quality. If an EMF is generated in MM_TEXT mapping mode on a 96-dpi screen, all the coordinates are integers in this 96-dpi coordinate space. When the EMF is displayed at a larger size, or to a high-resolution device, the coordinates are scaled up, which may show misalignment for text or roughness for polygons and path data.

Draw Commands in an EMF

The EMF supports quite a rich set of GDI drawing commands. Of the 122 known EMF record types, 47 of them are known to correspond to GDI commands. A few other record types are for recording OpenGL drawing commands and may be undocumented drawing commands.

Normally all coordinates in EMF are recorded in 32-bit values. But there are eight EMF record types that store their main data as 16-bit values. For example, PolyPolygon has two EMF record types, a 32-bit version EMRPOLYPOLYGON and a 16-bit version EMRPOLYPOLYGON16. The only difference between them is the point array. While the 16-bit version can save almost half of the space if the logical coordinate space is limited to 16 bits, it's not known which operating system actually uses it. Even Windows 98 seems to generate EMRPOLYPOLYGON instead of EMR POLYPOLYGON16 during an EMF printing spooling.

For simple drawing functions like LineTo or Rectangle, only the original coordinates appear in their EMF records. For more complicated functions, GDI stores the bounding rectangle of all the coordinates in the device coordinate space in their EMF record. For example, EMRPOLYLINE, EMRPOLYGON, and EMRSTRETCHBLT all have a bounding rectangle field. A bounding rectangle can be very handy in determining which EMF record should be skipped because it's out of boundary or clipped. For example, it can be used when GDI despools an EMF spool file to the printer driver on a per-band basis, in which the printer driver accepts only a horizontal band of the page at a time and GDI needs to be efficient in sending only the commands touching the band rectangle.

For the most basic pixel drawing functions, SetPixel and SetPixelV use the same EMF record type EMRSETPIXELV. The reason is, of course, that dependency on the function return value is resolved during the EMF generation time.

Line-and curve-drawing GDI functions are fully supported, except that MoveToEx does not return the last “cursor” point position and LineDDA is not supported. LineDDA is not a real drawing function, as it just breaks down lines into a sequence of pixel coordinates in the logical coordinate space. Any possible drawing will be done from callback functions provided by the caller, which will be recorded.

The GDI area-filling functions are fully supported. For example, Chord, Ellipse, Polygon, and even PolyPolygon all have their corresponding EMF records. One thing strange is that for functions like Rectangle and Ellipse, which define a geometric shape using a rectangular bounding box, GDI removes the right bottom edge when recording the EMF. For example { 0, 0, 50, 50} is recorded as { 0, 0, 49, 49} . So such rectangles in the EMF are interpreted as

both top-left and bottom-right inclusive. This is the same interpretation GDI uses when such drawing commands are executed in advanced graphics mode. For normal 1:1 scaling, GDI is able to make sure playing back such EMF records still draws the exact same shape. But where upscaling is involved, the pixel-level details could be different. For example, if you draw two rectangles, Rectangle(0, 0, 50, 50) and Rectangle(50, 0, 100, 50), they should just be touching each other, even when they are scaled up. But now that they are represented as { 0, 0, 49, 49} and { 50, 50, 99, 49} in the EMF, there will be a small gap when they are scaled up.

Three area-filling functions are not GDI functions. They are FillRect, Frame Rect, and InvertRect. They are functions provided by the window management module USER32.DLL, which calls GDI functions for drawing. They will be recorded in the EMF as brush creation, selection, and simple bitbltling calling.

The region drawing functions, FillRgn, FrameRgn, InvertRgn, and PaintRgn, are fully supported. But the GDI region object is converted to the region data and attached to these drawing commands.

Unlike region-construction functions, path-construction functions are fully supported. Path-drawing functions, FillPath, StrokeAndFillPath, and StrokePath, are supported. As they refer to the implicit path object in the device context, their EMF records just contain a simple bounding rectangle. GDI would have to call GetPath just to calculate their bounding rectangle during EMF generation.

Bitmap-drawing functions, from the simple BitBlt to AlphaBlend, are fully supported, except the simplest PatBlt function. PaltBlt is merged into its more generic form BitBlt, using the same EMRBITBLT record. Because of the merge, PaltBlt is represented in 100 bytes. If there were an EMRPATBLT record, it would be 66 bytes instead.

Texts in an EMF

The remaining four EMF records for drawing are for texts. They represent the GDI functions ExtTextOut and the less known PolyTextOut, with ANSI and UNICODE versions. PolyTextOut is simply multiple ExtTextOut calls combined into a single call.

TextOut is converted to ExtTextOut by GDI. It shares the same representation in an EMF using the EMREXTTEXTOUT record. Recall that the character-distance array in ExtTextOut is optional in GDI. But in an EMF, it's always filled with the right character-distance data. It gives a fixed, unambiguous character position for every character in the string. If you look back to [Figures 16-1](#) and [16-2](#), you can find that when an EMF is scaled up, the character distance is also scaled up. Character glyphs in those two pictures are not themselves scaled up, because the default font in the device context is being used. They will be scaled up accordingly if a custom logical font is used. We know that scaling up is not always accurate, because the original data is truncated from more precise floating-point numbers. To generate accurate character-distance arrays, an application can set up a higher-resolution logical device context and use the techniques described in the last chapter to get high-precision character-spacing data.

On Windows 2000 with Uniscribe and multiple language packages installed, it's found that logical font creation, selection, and deletion calls are added after EMREXTTEXTOUT record for TextOut and ExtTextOut. They can add over a dozen unnecessary EMF records to each text-drawing call. This behavior is not found on Windows NT or even on early releases of Windows 2000.

DrawText, DrawTextEx, and TabbedTextOut function calls are all converted to multiple ExtTextOut calls, which are mixed with SetTextAlign, SetBkMode, MoveToEx, and even SelectClipRgn EMF records. You will be surprised how many EMF records are used to represent one DrawText or TabbedTextOut call.

EMF Device Independence

Now that we've looked at some of the details of the EMF design, let's come back to the very basic question about the EMF—that is, to what extent is the EMF device independent?

Device independence is a key feature of the enhanced metafile design. When Microsoft documentation mentions EMF device independence, it's saying that when a picture measuring 2 inches by 4 inches on a VGA display is stored in an EMF, it will maintain the original dimensions when it's printed on a 300-dpi printer or displayed on a Super VGA monitor. Function `CreateEnhMetaFile` defines the recording picture-frame rectangle, which is kept in the EMF header structure, together with device resolution and dimension information. An application can query for an EMF header, get the picture's dimensions in real-world dimensions, convert it to the current device context's logical coordinate space, and call `PlayEnhMetaFile` with it to scale the EMF to the exact size as it's recorded. You can also scale the EMF to different scales. In terms of dimension device independence, the EMF does quite a good job. But the EMF generated using display device context as a reference depends on the display monitor dimension that is always reported as 320 mm by 240 mm. The resolution calculated according to it is not the same as the display monitor logical resolution most applications are using.

Another issue the EMF tries to solve is color device dependency. All device-dependent bitmaps are converted to device-independent bitmaps. Palettes are collected and put into a metafile palette which can be easily queried by application and realized before playing back an EMF. So if an EMF uses a logical palette, its color can be reproduced on another palette-based device quite well. Of course, there will be a little problem reproducing the color on a high-color or true-color device. But if an EMF is generated on a true-color device and it does not use a logical palette, playing it back on a palette-based device is not well supported by the basic features of the EMF. For such an EMF, even if an application selects and realizes a halftone palette before playing back, GDI resets the palette to the default 20-color system palette before actual drawing calls.

Yet another aspect of device independence is different implementations of the ever-growing Win32 GDI API. An EMF generated on Windows NT/2000 can't always be played back truthfully on a Windows 95/98 machine, because it may use advanced features available only on Windows NT/2000. If you want your EMF to be usable on all major currently active Win32 platforms, you have to restrict your GDI function set to its Windows 95 implementation.

Font could also be a concern. The EMF record for logical font creation, `EMR_EXTCREATEFONTINDIRCTW`, contains very detailed font description using the `EXTLOGFONTW` structure. It contains the `LOGFONT` structure, plus full name, style name, version, vendor identifier, and PANOSE number. Using this information, GDI can find either the exact match or a very close match. But if a good match can't be found, an EMF can't be displayed properly on another machine. Windows NT/ 2000 spooler solves this font dependency problem by embedding missing fonts into an EMF spool file, before it's transmitted to a printer server for printing. But the core EMF format does not support font embedding and runtime installation from within the EMF. There is no EMF record for functions like `AddFontResource`.

16.3 ENUMERATING AN EMF

In the last section, we discussed a way to walk through every EMF record in an EMF. The Win32 API also provides an interesting function to allow an application to enumerate every EMF record, in the context of playing an EMF to a device context. This function is unique and interesting because it provides a way for the application to monitor and customize the EMF playback. Here are the related definitions:

```
typedef struct tagHANDLETABLE {
    HGDIOBJ objecthandle[1]; // variable size
} HANDLETABLE;

typedef struct tagENHMETARECORD {
    DWORD iType;
    DWORD nSize;
    DWORD dParm[1]; // variable size
} ENHMETAFILERECORD;

typedef int (CALLBACK* ENHMFENUMPROC)(HDC hdc, HANDLETABLE * IpHTable,
                                         CONST ENHMETARECORD * IpEMFR, int nObj, LPARAM IpData);

BOOL EnumEnhMetaFile(HDC hdc, HENHMETAFILE emf,
                     ENHMFENUMPROC IpEnhMetaFunc, LPVOID IpData,
                     CONST RECT * IpRect);
BOOL PlayEnhMetaFileRecord(HDC hDC, LPHANDLETABLE IpHandleTable,
                           CONST ENHMETARECORD * IpEnhMetaRecord, UINT nHandles);
```

Function `EnumEnhMetaFile` accepts five parameters: a destination device context handle, a source EMF handle, a pointer to an application-provided callback routine, a pointer to application-provided data, and a picture-frame rectangle on the destination surface. Compared with `PlayEnhMetaFile`, only the third and fourth parameters are new. Actually, `EnumEnhMetaFile` is quite similar to `PlayEnhMetaFile`, in that they both play an EMF into a rectangle area on a destination device surface, except that the former calls a callback function for every EMF record.

A callback function for `EnumEnhMetaFile` accepts five parameters, too: a destination device context handle, a pointer to the EMF handle table, a pointer to the current EMF record, a maximum number of handles in the EMF handle table, and a pointer to the user-provided data. The EMF handle table is a table of GDI objects GDI maintains during playback of an EMF, which is responsible for translating object indexes used in the EMF to real GDI object handles. The maximum number of handles is originally kept in the EMF header record.

Each EMF record is exposed to the callback function through a generic ENH METARECORD structure, which basically says that an EMF record has a 32-bit type identifier, a 32-bit size field, and a number of double words.

Function `PlayEnhMetaFileRecord` is the workhorse used by `PlayEnhMetaFile` to handle playing back every EMF record. GDI exposes this function such that it can be called in the callback function to let GDI draw an EMF record into the destination device context.

C++ Class for EMF Enumerating

Any callback function that will be passed by application-provided data should be considered object friendly, because you can just pass this pointer through it, using a static callback function that dispatches to the C++ virtual function based on this information. Here is a generic C++ wrapping class for an EMF enumeration.

```
class KEnumEMF
{
    // virtual function to process EMF record, return 0 to terminate
    virtual int ProcessRecord(HDC hDC, HANDLETABLE * pHTable,
        const ENHMETARECORD * pEMFR, int nObj)
    {
        return 0;
    }

    // static function, dispatch to virtual function ProcessRecord
    static int CALLBACK EMFProc(HDC hDC, HANDLETABLE * pHTable,
        const ENHMETARECORD * pEMFR, int nObj, LPARAM lpData)
    {
        KEnumEMF * pObj = (KEnumEMF *) lpData;

        if ( IsBadWritePtr(pObj, sizeof(KEnumEMF)) )
        {
            assert(false);
            return 0;
        }
        return pObj->ProcessRecord(hDC, pHTable, pEMFR, nObj);
    }

public:
    BOOL EnumEMF(HDC hDC, HENHMETAFILE hemf, const RECT * lpRect)
    {
        return ::EnumEnhMetaFile(hDC, hemf, EMFProc, this, lpRect);
    }
};
```

The KEnumEMF class provides a virtual function ProcessRecord, which replaces the role of the callback function. Its default implementation returns 0, which stops the EMF enumeration. The main entry point EnumEMF calls GDI function EnhMetaFile with a static callback function EMFProc, which dispatches to the C++ virtual function.

The beauty of such C++ wrapping of Win32 API features is that you only have to deal with the Win32 API in one place, you add data members in derived classes, you can override the virtual function to provide different functionality, and you can have multiple instances of the classes.

Slow Motion in an EMF Playback

The easiest implementation of the KEnumEMF::ProcessRecord virtual function is just to call PlayEnhMetaFileRecord. Essentially, you're just reimplementing the GDI function PlayEnhMetaFile, with a little bit of delay because of the extra code.

While the word "delay" has a negative connotation, the right amount of delay could be perfect for learning. Here is the KDelayEMF class, whose only contribution is letting other threads have more time to run.

```
class KDelayEMF : public KEnumEMF
{
    int m_delay;

    virtual int ProcessRecord(HDC hDC, HANDLETABLE * pHTable,
                           const ENHMETARECORD * pEMFR, int nObj)
    {
        Sleep(m_delay);

        return PlayEnhMetaFileRecord(hDC, pHTable, pEMFR, nObj);
    }

public:
    KDelayEMF(int delay)
    {
        m_delay = delay;
    }
};

// sample usage
KDelayEMF delay(10);
delay.EnumEMF(hDC, hEmf, lpPictureRect);
```

If you have ever wondered how high-quality 3D text effects are implemented, now you can copy–paste a 3D text into this chapter's EMF sample program and examine the drawing process in slow motion. [Figure 16-6](#) shows a 3D-text string forming.

Figure 16-6. Playing back an EMF in slow motion.



Tracing an EMF Playback

You can even trace the process of the EMF playback and log information into a text window. In this way, you can understand how GDI really implements the EMF playback.

Here is a KTraceEMF class that uses the KEmfDC class to decode each EMF record, which is logged to a text window implemented by the KLogWindow class.

```
class KTraceEMF : public KEnumEMF
{
    int         m_nSeq;
    KEmfDC      m_emfdc;
    int         m_value[32];
    HGDIOBJ     m_object[8];
    FLOAT       m_float[8];

    virtual int ProcessRecord(HDC hDC, HANDLETABLE * pHTable,
                           const ENHMETARECORD * pEMFR, int nObj)
    {
        CompareDC(hDC);

        m_pLog->Log("%4d: %08x %3d % 6d ", m_nSeq++, pEMFR,
```

```
pEMFR->iType, pEMFR->nSize);

m_pLog->Log(m_emfdc.DecodeRecord((const EMR *) pEMFR));
m_pLog->Log("\r\n");
return PlayEnhMetaFileRecord(hDC, pHTable, pEMFR, nObj);
}

public:
KLogWindow * m_pLog;

void CompareDC(HDC hDC);

KTraceEMF(HINSTANCE hInst)
{
    m_pLog = new KLogWindow; // de-allocated in WM_NCDESTROY handling
    m_pLog->Create(hInst, "EMF Trace");
    m_nSeq = 1;
    memset(m_value, 0xCD, sizeof(m_value));
    memset(m_object, 0xCD, sizeof(m_object));
    memset(m_float, 0xCD, sizeof(m_float));
}
};
```

One extra feature implemented by the KTraceEMF class is comparing the attributes of the device context before the playing back, between each EMF record, and after the playing back. The attributes of a device context are queried using normal GDI functions like GetBkMode, kept in three arrays, and compared with previous values. By logging the changes in the device context attributes, you can get a deeper understanding of how the EMF playback is implemented. Here is an incomplete sample trace generated using the KTraceEMF class.

/////////// Before Drawing //////////

```
GraphicsMode : 1
WT.eM11     : 1.00000
WT.eM12     : 0.00000
WT.eM21     : 0.00000
WT.eM22     : 1.00000
WT.eDx      : 0.00000
WT.eDy      : 0.00000
Pen        : 0x01b00017
Brush       : 0x01900010
Font        : 0x018a0021
Palette     : 0xa50805ca
```

/////////// Starting Drawing //////////

```
GraphicsMode : 2
WT.eDx      : 5.00000
WT.eDy      : 5.00000
Font        : 0x018a0026
Palette     : 0x0188000b
```

```
1: 012e0000 1 132 // Header
2: 012e0084 27 16 MoveToEx(hDC, 300, 50, NULL);
3: 012e0094 54 16 LineTo(hDC, 350, 50);
4: 012e00a4 27 16 MoveToEx(hDC, 350, 51, NULL);
5: 012e00b4 54 16 LineTo(hDC, 400, 51);
6: 012e00c4 43 24 Rectangle(hDC, 300, 60, 349, 109);
7: 012e00dc 43 24 Rectangle(hDC, 350, 80, 399, 129);
8: 012e00f4 42 24 Ellipse(hDC, 410, 60, 459, 149);
9: 012e010c 42 24 Ellipse(hDC, 460, 60, 509, 149);
10: 012e0124 76 100 PatBlt(hDC,300,150,100,100,BLACKNESS);
11: 012e0188 76 100 PatBlt(hDC,400,160,100,100,PATINVERT);
12: 012e01ec 39 24 hObj[1]=CreateSolidBrush(RGB(0x59,0x97,0x64));
13: 012e0204 37 12 SelectObject(hDC, hObj[1]);
Brush      : 0x5f10045e
...
71: 012e0978 14 20 // EMREOF(0, 16, 20)
```

/////////// After Drawing //////////

```
GraphicsMode : 1
WT.eDx      : 0.00000
WT.eDy      : 0.00000
Font        : 0x018a0021
Palette     : 0xa50805ca
```

What's shown here is very interesting. Before calling EnumEnhMetaFile, the destination device context is in its default compatible graphics mode with default attributes, except that its palette is a halftone palette. When the callback routine is called the first time to process the first EMF header record, the graphics mode is changed to the advanced graphics mode, the world-transformation matrix is updated, and the font/palette handles are changed to stock GDI objects. This indicates that on Windows NT/2000, GDI uses the advanced graphics mode with world transformation to implement the EMF playback, and other attributes in the device context are always reset to their default state before drawing an individual EMF record.

Using the advanced graphics mode is really convenient for GDI to implement the EMF playback. GDI can just combine the three transformation matrices shown in [Figure 16-5](#), set it as the world-transformation matrix during the EMF playback, and then draw all the EMF records using the original coordinates recorded in the EMF. The only trick is in translating the clipping region from the recording device's device coordinate space to the destination device's device coordinate space, which can be handled by ExtCreate Region. But on Windows 95/98, where the advanced graphics mode is not really implemented by GDI, GDI has to use the MM_ANISOTROPIC mapping mode with an adjusted window-to-viewport mapping which is equivalent to the combined transformation matrix.

The trace also shows changes in GDI objects associated with the device context. You can see that GDI always resets them to GDI stock objects before rendering the EMF, which explains why you can't hope that selecting the logical palette before the playing back can help an EMF without a logical palette to display the right colors.

The KTraceEMF class can be extended to do more useful things like monitoring GDI object creation, selection, and deletion to delete a possible resource leak in the EMF. Although GDI always frees leftover handles in the EMF handle table, finding a possible resource leak can help fix the EMF generation code.

Changing an EMF Dynamically

The EMF record passed to the callback function is read-only. It can't be modified and passed back to GDI. But an application can make a copy of the record, change it on the fly, and pass the new copy to GDI for drawing. That is to say, you can dynamically modify the EMF and let GDI draw the modified version.

Here is a simple class that maps all foreground color, background color, pen color, and brush color to gray. If your EMF does not contain color bitmaps, displaying it using the KGrayEMF class turns color EMF to grayscale EMF. More code needs to be added to handle bitmap grayscale conversion, which is covered in [Chapter 12](#) of this book.

```
inline void MaptoGray(COLORREF & cr)
{
    if ( (cr & 0xFF000000) != PALETTEINDEX(0) ) // not paletteindex
    {
        BYTE gray = ( GetRValue(cr) * 77 + GetGValue(cr) * 150 +
                      GetBValue(cr) * 29 + 128 ) / 256;
        cr = (cr & 0xFF000000) | RGB(gray, gray, gray);
    }
}

class KGrayEMF : public KEnumEMF
{
    virtual int ProcessRecord(HDC hDC, HANDLETABLE * pHTable,
                             const ENHMETARECORD * pEMFR, int nObj)
    {
        int rsIt;

        switch ( pEMFR->iType )
        {
            case EMR_CREATEBRUSHINDIRECT:
            {
                EMRCREATEBRUSHINDIRECT cbi;
                cbi = * (const EMRCREATEBRUSHINDIRECT *) pEMFR;
                MaptoGray(cbi.lb.lbColor);
                rsIt = PlayEnhMetaFileRecord(hDC, pHTable,
                                             (const ENHMETARECORD *) & cbi, nObj);
            }
            break;

            case EMR_CREATEPEN:
            {
                EMRCREATEPEN cp;
                cp = * (const EMRCREATEPEN *) pEMFR;
                MaptoGray(cp.lopn.lopnColor);
                rsIt = PlayEnhMetaFileRecord(hDC, pHTable,
                                             (const ENHMETARECORD *) & cp, nObj);
            }
        }
    }
}
```

```
    }

    break;

    case EMR_SETTEXTCOLOR:
    case EMR_SETBKCOLOR:
    {
        EMRSETTEXTCOLOR stc;
        stc = * (const EMRSETTEXTCOLOR *) pEMFR;
        MapToGray(stc.crColor);
        rsIt = PlayEnhMetaFileRecord(hDC, pHTable,
            (const ENHMETARECORD *) & stc, nObj);
    }
    break;

    default:
        rsIt = PlayEnhMetaFileRecord(hDC, pHTable, pEMFR, nObj);
    }
    return rsIt;
}
};
```

The KGrayEMF class represents a class of on-the-fly transformations you can apply to an EMF. Using the same approach, you can change pen width to make sure the pens are of the right thickness for a graphic device, replace any hatch pattern usage which is too fine for printing, remove all bitmaps from a document, or adjust all the colors used in an EMF document.

Deriving an EMF from an EMF

The destination device context for PlayEnhMetaFile and thus the KEnumEMF::Enum EMF method can be any reasonable device context, including an enhanced metafile device context. If an EMF device context is passed to PlayEnhMetaFile, and you provided a customized callback routine, essentially you're controlling deriving a new EMF from an existing EMF. Such an editing opportunity is always exciting, but there are also some surprises and enlightenment.

Here is the FilterEMF routine and its supporting function Map10umToLogical.

```
void Map10umToLogical(HDC hDC, RECT & rect)
{
    POINT * pPoint = (POINT *) & rect;

    // convert from 0.01 mm to pixels for current device
    for (int i=0; i<2; i++)
    {
        int      t = GetDeviceCaps(hDC, HORZSIZE) * 100;
        pPoint[i].x = (pPoint[i].x*GetDeviceCaps(hDC,HORZRES) + t/2) / t;

        t = GetDeviceCaps(hDC, VERTSIZE) * 100;
        pPoint[i].y = (pPoint[i].y*GetDeviceCaps(hDC,VERTRES) + t/2) / t;
    }
}
```

```
pPoint[i].y = (pPoint[i].y*GetDeviceCaps(hDC,VERTRES) + t/2) / t;  
}  
// convert to logical coordinate space  
DPtoLP(hDC, pPoint, 2);  
}  
  
HENHMETAFILE FilterEMF(HENHMETAFILE hEmf, KEnumEMF & filter)  
{  
    ENHMETAHEADER emh;  
    GetEnhMetaFileHeader(hEmf, sizeof(emh), & emh);  
  
    RECT rcFrame;  
    memcpy(& rcFrame, & emh.rcFrame, sizeof(RECT)); // in .01 mm  
  
    HDC hDC = QuerySaveEMFFile("Filtered EMF\0", & rcFrame, NULL);  
    if ( hDC==NULL )  
        return NULL;  
  
    Map10umToLogical(hDC, rcFrame); // convert to logical space  
    filter.EnumEMF(hDC, hEmf, & rcFrame);  
    return CloseEnhMetaFile(hDC);  
}
```

The FilterEMF function generates a new EMF from an existing EMF, and the generation process is fully controlled by an instance of the KEnumEMF class, or the class derived from it. It uses the frame rectangle in the source EMF as the frame rectangle for the new EMF and converts it to logical coordinate space units for the new EMF. This ensures that the new EMF has the same dimensions and white margin as the original EMF. The rectangle conversion is done using the help function Map10 umToLogical.

If you use the KGrayEMF class, which is referred to as the filter in FilterEMF, a grayscale version of your EMF will be generated if there are no bitmaps in the EMF. The new EMF looks identical in shape but different in color.

But the new EMF is 424 bytes longer and contains 26 more EMF records for a sample test case. There is something to learn here. Here are the extra EMF records as shown by the KTraceEMF class.

```
2: SaveDC(hDC);  
3: SetLayout(hDC, 0);  
4: SetMetaRgn(hDC);  
5: SelectObject(hDC, GetStockObject(WHITE_BRUSH));  
6: SelectObject(hDC, GetStockObject(BLACK_PEN));  
7: SelectObject(hDC, GetStockObject(DEVICE_DEFAULT_FONT));  
8: SelectPalette(hDC, (HPALETTE)GetStockObject(DEFAULT_PALETTE), TRUE);  
9: SetBkColor(hDC, RGB(0xFF,0xFF,0xFF));  
10: SetTextColor(hDC, RGB(0,0,0));  
11: SetBkMode(hDC, OPAQUE);  
12: SetPolyFillMode(hDC, ALTERNATE);  
13: SetROP2(hDC, R2_COPYPEN);  
14: SetStretchBltMode(hDC, STRETCH_ANDSCANS);
```

```
15: SetTextAlign(hDC, TA_NOUPDATECP | TA_LEFT | TA_TOP);
16: SetBrushOrgEx(hDC, 0, 0, NULL);
17: SetMiterLimit(hDC, 0.00000);
18: // Unknown record [120]
19: MoveToEx(hDC, 0, 0, NULL);
20: SetWorldTransform(hDC, 1, 0, 0, 1, 0, 0);
21: ModifyWorldTransform(hDC, 1, 0, 0, 1, 0, 0, 0x4 /*Unknown*/);
22: SetLayout(hDC, 0);
23: // GdiComment(52, GDIC, 0x2)

...
93: // GdiComment(8, GDIC, 0x3)
94: RestoreDC(hDC, -1);
95: DeleteObject(hObj[1]);
96: DeleteObject(hObj[2]);
```

What's shown here are the exact steps GDI uses to play an EMF. Because the destination device context is an EMF device context, everything is truthfully recorded instead of silently executed.

EMF Playback Details

Now let's follow the exact steps by which GDI plays an EMF. GDI first saves the current status of a device context using SaveDC, which is finally restored using Restore DC, so the caller to PlayEnhMetaFile or EnumEnhMetaFile will not notice any change in the device context.

A rarely used function, SetMetaRgn, is called next. Recall that a meta region is a device context attribute which, together with the clipping region, forms a two-layer control of clipping. SetMetaRgn combines the current clipping region application sets before calling EnumEnhMetaFile into the meta region, and resets the clipping region to NULL. During the playing back of the EMF, the clipping region in the EMF can be treated as a clipping region, but it's always combined with the meta region for the actual drawing. This is a clever usage of the meta region feature. Or the meta region was originally designed just for playing back the EMF. Note that neither the frame rectangle, nor the display rectangle, has anything to do with clipping.

The dozen or so EMF records after that are easy to understand, as they just set some attributes in a device context to some default values, to insulate the EMF playback from existing attributes in a device context.

The SetWorldTransformation and ModifyWorldTransformation EMF records look very interesting. Because we do the calculation in FilterEMF the right way, both records shown use the identity matrix. If translation or scaling is involved, the transformation matrices will change accordingly. What's strange is that there is no EMF record to switch the graphics mode to the advanced graphics mode. We know from previous sample output from the KTraceEMF class that GDI does switch the graphics mode during the EMF playback. The only explanation why GDI does not put the graphics mode switching the EMF record explicitly into the EMF is that the designer wanted to make sure the new EMF could be used on Windows 95/98, too. Actually Windows 98 GDI also generates the world-transformation EMF records, but uses the MM_ANISOTROPIC mapping mode during runtime.

If GDI uses world transformation to play back the EMF, special care should be taken with the text. Recall that the text in the compatible graphics mode is always displayed in the upright position, even if the logical coordinate system is set up for the y-axis to go bottom-up. But in the advanced graphics mode, the text truly follows the axis direction like the rest of graphics primitives. Windows NT/2000 solves this mismatch by changing the world coordinate system before the text drawing calls. It is safe to use any mapping mode with an EMF. Note also that the

ModifyWorldTransform EMF record uses an undocumented flag 4.

The last three extra EMF records restore the device context status and delete two GDI objects. Impressively, GDI detects that there are two GDI object resource leaks in the original EMF. But the resource leak is not caused by my code; it was caused by GDI itself during the original EMF generation. The original code uses two GetSysColorBrush to get brushes with system colors. Because the brushes are returned from a table kept by USER32.DLL, an application is responsible for deleting them. Actually, system color brushes are associated with the system process with process identifier of 0, so they are shared systemwide. But a system brush's color is device dependent, so GDI has to convert it to a normal solid-color brush during EMF generation. GDI forgets to delete these brushes in the original EMF generation.

We still have two more things to explain: undocumented EMF record types and GdiComment records.

Undocumented EMF Record Types

One undocumented EMF record type is used by the initialization phase, record type 120, which is defined as EMR_RESERVED_120 in WINGDI.H. With Enum EnhMetaFile, it's very easy to figure out what it is. You just need to add a check for EMR_RESERVED_120 in the callback routine, pass the record to GDI using Play Enh MetaFileRecord, set a breakpoint, and venture just a few steps in the assembly code.

You will find that PlayEnhMetaFileRecord checks if the EMF record type is within the valid range supported on a system (EMR_MIN to EMR_MAX), and calls a function from a function table. For EMR_RESERVED_120, it calls bPlay::MR_SETTEXT JUSTIFICATION, which in turn calls SetTextJustification. So EMR_RESERVED_120 should be EMR_SETTEXTJUSTIFICATION. You may wonder why it's needed in the EMF, as all text drawing calls use an explicit character-distance array.

Here is the whole list of undocumented EMF record types:

EMR_RESERVED_105 ESCAPE
EMR_RESERVED_106 ESCAPE
EMR_RESERVED_107 STARTDOC
EMR_RESERVED_108 SMALLTEXTOUT
EMR_RESERVED_109 FORCEUFIMAPPING
EMR_RESERVED_110 NAMEDESCAPE
117 unused
EMR_RESERVED_119 SETLINKEDUFIS
EMR_RESERVED_120 SETTEXTJUSTIFICATION

GDIComment

GDIComment is a special function for attaching extra data to an EMF. A comment can include any kind of private information known to the writer and reader. For example, an application can include a PostScript version of an object represented in an EMF. The knowledgeable reader can choose to use the PostScript version instead of trying to decode GDI commands.

Microsoft documents several standard GDI comments; they all start a 32-bit identifier GDICOMMENT_IDENTIFIER, which is "GDIC" in character form. For example, GDICOMMENT_WINDOWS_METAFILE attaches the old Windows

meta file data to an EMF; GDICOMMENT_BEGINGROUP signals the starting of a drawing record group, and GDICOMMENT_ENDGROP signals the end of a drawing record group.

What we see in our sample is GDICOMMENT_BEGINGROUP (2) and GDICOMMENT_ENDGROP (3), which separates extra commands added by GDI from original commands sent from the application.

Deriving a new EMF from an existing EMF can be put into more advanced real usage that's what's demonstrated here. For example, there are lots of optimizations you can apply to EMF records, to remove unused object creation, settings, and parts of bitmaps not actually used. You can also apply a nonaffine transformation to an EMF to generate some special effects. In the next section, we will talk about similar surgery operations done in another domain, C code generated from an EMF.

[< BACK](#) [NEXT >](#)

16.4 AN EMF AS A PROGRAMMING TOOL

An EMF, as we have discussed it so far, is a graphics object that can be used to encode, transmit, and reproduce graphics data, across application, across devices, and across operating systems. But the ability to record GDI commands into an easy-to-manipulate piece of data can be put to other good uses.

In this section we will explore how an EMF can help you in programming, instead of just providing media to graphic artists.

An EMF Decompiler

Throughout this chapter we have been using the KEmfDC class, which can decode EMF records into a TreeView window or into a text log window. Actually, decoding individual EMF records is its part-time job; its original goal is to decompile an EMF into a piece of C code which can be compiled and run to achieve the same display result as drawing the EMF.

The EMF decompiler uses special pattern strings to encode how a simple EMF record can be decoded to C code. Here are a few examples.

```
const EmrInfo Pattern [] =
{
    { EMR_SETWINDOWEXTEX , "SetWindowExtEx(hDC, %d, %d, NULL);" },
    { EMR_EXTCREATEFONTINDIRECTW,
        "#o=CreateFont(%d,%d,%d,%d,%d,%b,%b,%b,%b,%b,%b,%b,%b,%b,%S);"
    },
    { EMR_SETPIXELV      , "SetPixelV(hDC, %d, %d, #c);" },
    { EMR_POLYGON        , "\nstatic const POINT Points_%n[]={#P;\n"
                            "Polygon(hDC, Points_%n, %4);"
    },
    ...
};
```

The first example says that an EMR_SETWINDOWEXTEX record can be decoded into a call to SetWindowExtEx with two 32-bit integer values taken from the EMF record. The second example says an EMR_EXTCREATEFONTINDIRECTW call should be decompiled into an assignment statement, which assigns the result from CreateFont to an entry in the EMF object table. The parameters for the CreateFont call are five 32-bit values, eight 8-bit values, and a string. The third example says an EMR_SETPIXELV record should be translated into a SetPixelV call with two 32-bit integers and a color specifier. The last example shows how a constant static array is introduced to hold a variable-size POINT array for the EMR_POLYGON record, which can be translated to a Polygon function call.

More complicated EMF record types can't be captured using these patterns, so they are handled by special code. Bitmaps are written into separate files, which can then be linked-in as resource. An EMF is translated into an OnDraw routine which accepts a device context handle. Extra code is added before and after that to make it a simple yet complete window program, which creates a simple window and handles a WM_PAINT message to draw the decompiled EMF.

Here is some sample output from the decompiler. As you can see, it's smart enough to find duplicated bitmaps and reuse them (the two StretchDIBits call use the same bitmap).

```
void OnDraw(HDC hDC)
{
    HGDIOBJ hObj[5] = { NULL };

    MoveToEx(hDC, 300, 50, NULL);
    LineTo(hDC, 350, 50);
    Rectangle(hDC, 300, 60, 349, 109);
    Ellipse(hDC, 410, 60, 459, 149);
    PatBlt(hDC,300,150,100,100,BLACKNESS);
    hObj[1]=CreateSolidBrush(RGB(0x59,0x97,0x64));
    SelectObject(hDC, hObj[1]);
    PatBlt(hDC,300,300,100,100,PATCOPY);
    SelectObject(hDC, GetStockObject(WHITE_BRUSH));

    static const POINT Points_1[]={ 10, 200, 50, 200, 90, 200, 130, 200 };
    Polyline(hDC, Points_1, 4);

    static KDIB Dib_1; Dib_1.Load(IDB_BITMAP1);// 350x250x8
    Dib_1.StretchDIBits(hDC, 10,10,350,250, 0,0,350,250,
        DIB_RGB_COLORS, SRCCOPY);

    Dib_1.StretchDIBits(hDC, 10,270,350,250, 0,0,350,250,
        DIB_RGB_COLORS, SRCCOPY);

    ...
}
```

Now you may ask, why would anyone want to decompile an EMF into C/C++ code? There are lots of good reasons to do so. As software gets more and more complicated, especially when there are multiple components developed by different teams or even different companies, it's very hard to understand software at a system level. Engineers of other professions have lots of tools to help them understand runtime system behavior; software engineers only have debuggers, API monitoring tools, and trace statements. For graphics drawing, an EMF is a perfect log of the GDI calls made by different components of a system. Decompiling an EMF into a more readable C code can help engineers identify problems and play with code to find possible solutions. The decompiled code can also identify unnecessary GDI calls, inefficient coding practice, or GDI functionality you should try to avoid because it's not handled so well by GDI.

Printing problems could be hard to track down, because the application, GDI, and the printer device driver have to work smoothly together to deliver the final output. Most printer drivers support EMF spooling. By capturing the EMF sent to the printer, decompiling it, and analyzing the output, lots of printing problems can be made clear.

EMF generation can be seen as a program-slicing transformation, as it only records drawing calls to an output device context, while evaluating all other function calls. For example, if you use GetGlyphOutline to retrieve glyph outlines to draw 3D text into an EMF device context, only the final drawing calls are recorded. This can be very useful if an application does not want the lengthy computation process, but only the final processed data for high-performance drawing. The decompiled EMF makes it easy for an application to embed processed data. Some applications are embedding complicated region definitions generated from bitmaps, or vector graphics data, into the application to customize the normal rectangular shaped window. Other drawing API can also use graphics data generated by GDI. For example, DirectDraw does not use the GDI region object directly, but it uses REGIONDATA to define clipping.

Yet another case for decompiling the EMF is optimizing the EMF. We know that the EMF generation process is mostly a recording process. The only known optimizations are objects that are not recorded unless they are selected, and bitmaps may be trimmed in some cases. Objects that are created, selected, queried, and then deselected are recorded into the EMF, although they are not needed in the EMF any more. Extra settings, saves and restorations of device context, are not optimized either. For certain applications where size or performance is critical, manual optimizing decompiled EMF is a way to solve the problem.

Capture an EMF Spool File

Besides exchanging graphics data between applications, an EMF is used to handle almost all print jobs on Win32 systems. Printers are normally slow devices, much slower than modern computers. When an application starts printing, instead of letting the application wait for the printer to finish, the Windows spooler with the help of GDI re directs all drawing requests from the application into the EMF files. This step is called spooling. After the spooling is done, as far as the application is concerned, the print job is finished. So the user can continue his/her job with the application, while the spooler is playing back the EMF spool file to the real printer driver.

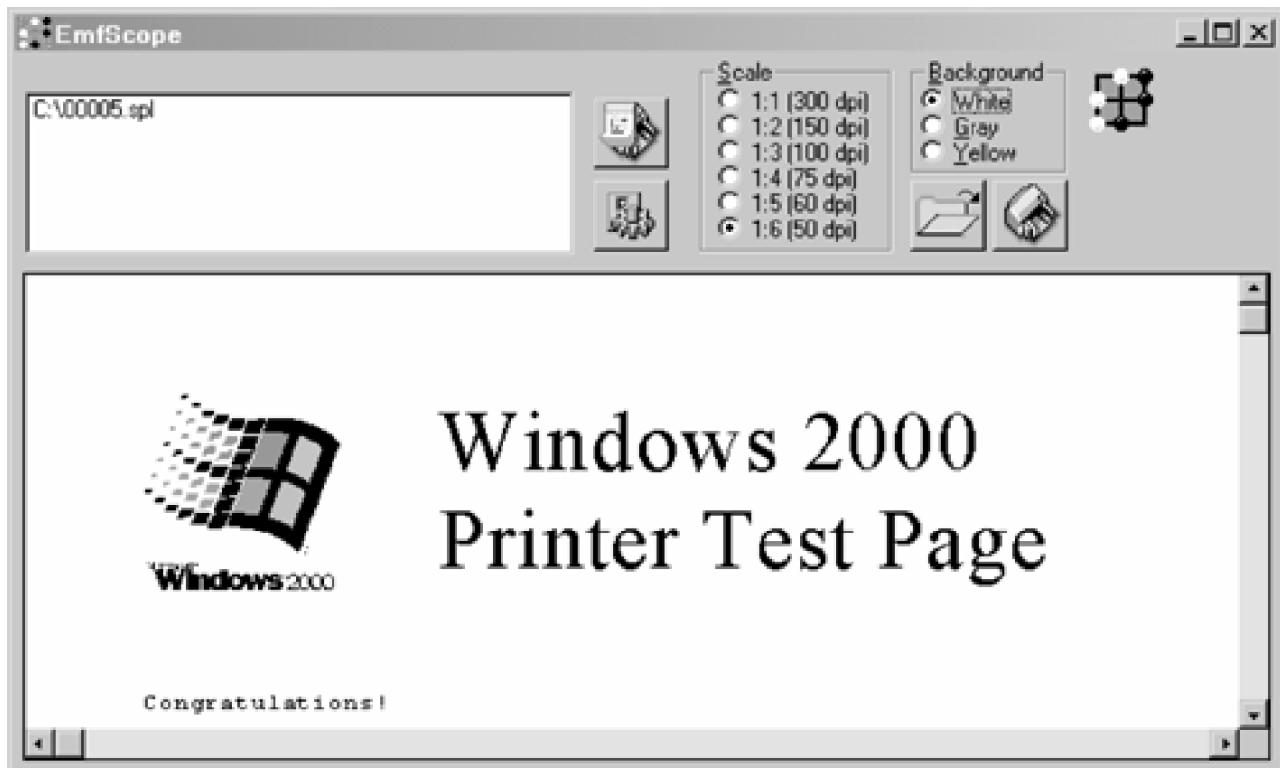
Windows 95/98 uses the standard EMF format for a spool file format. Each page within a printing document is written to a separate EMF spool file. Normally, the spool files are written into the Windows temporary directory, using names like ~emfxxxx.tmp. Once a page of the EMF is generated, the spooler calls a GDI function gdiPlaySpoolStream to play back the EMF spool page to the printer driver. To be more precise, the EMF spool page is sent to the print processor first, which in turn passes the job to the right printer driver after necessary processing. The spooler process is named “Spooler Process” and it creates a window with the class name “SpoolProcessClass”. So it's very easy for an external program to find the hidden window created by the spooler process and install a hook on it, thus injecting a DLL into the spooler process. Once in the spooler process, the injected DLL can hijack the call to gdiPlaySpoolStream. [Chapter 4](#) of this book has extensive coverage of API spying techniques. The gdiPlaySpoolStream function accepts a spool job file, which links all the EMF pages for a printing job together. Although the format for this spool job file is not documented, it's not hard to find file names with a small file. So we have a way to hook into the Windows 95/98 spooler process and get the name of every EMF spool file before the printer driver actually prints it. With the file name, you can make a copy of the EMF file and do whatever you want to do with it.

The Windows NT/2000 spooler is a little bit different from Windows 95/98. It does not create a hidden window; the normal ways of injecting a DLL into it would not work because it's a system process. Even the spool file is different. Now the drawing commands for a whole job are spooled into a single file that is not in the standard EMF format. For each print job there are two spool files generated by the spooler—a file with .SHD extension for settings, and a file with .SPL extension for drawing commands. Luckily, Windows NT/2000 spooler has a nice feature to keep the spool files after a print job is printed, so that they can be reused. So we don't have to hook into the spooler process to get the spool file. The spool files are normally generated in your SystemRoot\system32\spool\printers directory.

The EMF spool file used on Windows NT/2000 can be described as a meta EMF format. An EMF spool file contains a sequence of meta records with a 32-bit type identifier and a 32-bit size, followed by variable-size data. The EMF data for each page in a print job is attached to one of these meta records. This design allows a single spool file to handle a whole print job.

[Figure 16-7](#) shows a spool-file capturing and displaying tool, EmfScope. When running on Windows 95/98, EmfScope can grab the EMF spool files automatically from the spooler and display them in its window as they appear. When running on Windows NT/2000, EmfScope can display the spool file. You can change the display scale or scroll the EMF in the window. [Figure 16-7](#) displays a scaled-down version of a printer test page.

Figure 16-7. EmfScope: Capturing the EMF spool file from the spooler.



As we mentioned earlier, printing problems are usually very hard to identify because the application, GDI, and the printer driver all play important roles in it. Now that you can get the EMF spool file from the spooler, display it on the screen, zoom to different scale, scroll up and down freely, and even decompile it to C code, you can much more easily identify the possible causes for rendering related printing problems. For example, if an object does not appear on the EMF spool file, the printer driver has no way to print it, so most likely it's an application or GDI problem. If a document generates a huge EMF spool file, a reasonable cause is that certain GDI commands are not efficiently encoded in the EMF, so you should check the decompiled code to identify the exact problem and work for a workaround. If an EMF looks perfect on the screen, but does not print right, most likely it's a printer driver defect.

[< BACK](#) [NEXT >](#)

16.5 SUMMARY

This chapter is about a very useful feature in GDI programming, the metafile, which is not adequately covered in Windows programming literature. We discussed basic concepts around the two metafile formats, simple metafile usage, details about metafile internal organization, how to enumerate an EMF to achieve special displaying or debugging goals, how to decompile an EMF into C code, and finally how to capture the EMF spool file from the spooler.

The Windows Enhanced Metafile is mainly designed for an application to exchange graphics data with other applications or other devices so that the data can be displayed with the same dimension and color. As we have seen, an EMF does quite a good job in achieving this device-independent displaying goal. It does the job so well that now it's widely used during printing. But an EMF is not designed to exchange graphics data for other purposes like editing, because what's encoded is GDI drawing commands, instead of high-level object description.

The last part of this chapter about EMF usage in spooling leads us naturally to the next chapter, printing.

Further Reading

Another common graphics metafile format is the CGM (Computer Graphics Meta file) format. It was designed under ISO and ANSI as a common format for platform-independent interchange of bitmap and vector data. An active web site supporting CGM is at www.cgmopen.org.

Microsoft Platform SDK contains a nontrivial EMF sample program, an EMF editor. It's under the Samples\Multimedia\MetaFile\MfEdit directory. You can also get an EMF decoder from MSDN.

Sample Programs

There are two major sample programs and one small sample program for this chapter (see [Table 16-2](#)). More generic functions and classes related to EMF are put into emf.h and emf.cpp.

Table 16-2. Sample Programs for Chapter 16

Directory	Description
Samples\Chapt_16\EMF	Program for testing EMF generating, loading, saving, exchanging with clipboard, decoding, various ways of enumerating an EMF, generating a new EMF from an EMF, and decompiling an EMF.
Samples\Chapt_16\test	Test program to put a decompiled EMF into a stand-alone Windows program.
Samples\Chapt_16\EMFScope	Capturing and displaying an EMF spool file from the spooler.

Chapter 17. Printing

Printing in the Windows operating system is much easier compared with the old-time DOS operating system when every application had to design its own driver framework. But even with the added convenience provided by the Win32 API, the bar for quality printing has been raised so high by commercial applications that substantial design and implementation efforts are needed. The difficulty in providing the printing feature in applications has even been quoted as one of the reasons people are moving to MFC (Microsoft Foundation Class), which provides a much smoother, yet not really professional, wrapping around Win32 printing.

This chapter will discuss the process and components within the Win32 printing system, how to connect to the printer, how to use GDI commands to draw into the printer device context, and printing of simple GDI primitives. We will provide complete sample programs on how to implement truly device-independent WYSIWYG multiple -page source-code printing with syntax highlighting, and JPEG image printing.

17.1 UNDERSTANDING THE SPOOLER

The Windows printing system is a complicated subsystem of the whole graphics system. Although to a normal user or even a programmer, the complexity of the printing system is not so obvious, when there are really hard printing-related problems to solve, you will start to know a long list of components in the printing system and their interactions.

Section 2.4 of this book gives a detailed description of the printing subsystem architecture. There is a long list of components in the printing subsystem. The better-known players in printing are the user application, GDI, Windows graphics engine, printer driver, and the actual printer. The lesser-known components are the spooler process, spooler client DLL, spooler router, print provider, print processor, language monitor, port monitor, and the port IO driver.

Instead of discussing these components in more detail, we will focus in this section on the relationship between normal Windows applications and the printing subsystem.

Printing Process

Printing even a simple job in the Windows operating system is quite an involved process. Over a dozen steps are involved from initially querying for a printer to actually finishing a print job. Here is a summary of the process.

- The application queries the Win32 spooler client DLL, either through spooling functions or through printing-related common dialog boxes, to find information about current printers. Printing common dialog boxes provides an easy way to use the wrapper around the spooler API.
- Win32 spooler client DLL is a user-mode DLL that provides the application and the user-interface portion of a printer driver, which can be used to query about printers, printing jobs, printing settings, etc.
- Once the printing setting is negotiated, the application sends a printing job to the system, using GDI commands. A few special GDI commands are used to signal the start and end of a printing job, and to divide a document in pages.
- GDI accepts the GDI drawing command from the application, writing into the EMF spool file, and passes the spool file to the spooler process.
- The spooler process is a system service that manages spooling and despooling of printing jobs. The spooler client DLL talks to it through remote procedure calls. The spooler passes the job to the spooler router.
- The spooler router is responsible for finding the right print provider to handle a printing job.
- The print provider is responsible for directing the printing job to the right machine physically connected with a printer, which can be a local machine or a remote machine. It also manages the print-job queue, and implements the API that starts, stops, or enumerates printing jobs. The operating system provides the local print provider, Windows network print provider, Novell Netware print provider, and the HTTP print provider. If the printer is not on the local machine, a network print provider is responsible for sending the job over the

network. The job is finally handled by a local print provider, which passes the job to a print processor.

- The print processor is responsible for converting the spooled printing file into a raw data format that can be processed directly by a printer. The most commonly used print processor on Windows 2000 is the EMF print processor that is part of the local print provider.
- The print processor calls GDI to despool the EMF-spool file to a real printer driver device context.
- The windows graphics engine calls the printer driver to implement the GDI drawing command sent to a printer device context, and supports possible rendering required by printer driver.
- The printer driver is responsible for translating the DDI (device driver interface) level drawing primitives to raw data acceptable by the printer. The printer driver writes raw data back to the spooler.
- The spooler passes raw data to a language monitor, which is responsible to provide a full-duplex communication channel between the spooler and printer. A language monitor passes the data to a port monitor.
- A port monitor provides a communication path between the spooler and a kernel-mode port I/O driver, which really has access to the hardware I/O port linked to a printer.
- The port I/O driver sends data from the host machine to the printer. It also receives status information from the printer and passes it back to the port monitor and language monitor.
- The printer firmware running on an embedded CPU in the printer receives data from the printer's I/O port, uncompresses and processes it to a format suitable for driving the printing cartridge, and controls countless mechanical, electrical, and electronics components in the printer to put real dots on a piece of paper. A printer may have a high-powered RISC CPU, tons of memory, a hard disk, and be supported by a real-time multitasking operating system and a sophisticated firmware.

Printer Control Language

The language in which the host machine communicates with a printer is called *printer control language*. The so-called raw printer data, or printer-ready data, is expressed in printer control language. Different printers accept different printer control language, which is supported by firmware in the printer. Some printers may support multiple printer control languages, with special commands to switch between them.

There are three major classes of printer control languages. The choice of the printer control language is a fundamental design decision in designing a printer. It determines the feature set, complexity, and cost of a printer, the complexity of a printer driver, printing speed, and resource requirement on a host machine.

Text-Based Printer Control Language

The lowest-level printer control languages accept plain text with limited formatting commands. The traditional line printers belong to this class. Such printers can print only text documents, not vector or bitmap graphics. Some people may still remember the old days of simulating bitmap graphics on line printers by overstriking carefully selected combination of letters to form different shades of grayscale. These printers are still used even today to print on multiple-sheet pages and special forms or to generate long accounting reports. Normally all printers can be used as line printers. For example, you can still copy a text file in a DOS box to the LPT1 port, which will be sent to the

printer in its original text format.

Raster-Based Printer Control Language

The second class of printer control languages accepts raster bitmaps in certain formats. They may represent most of the printers on the market. Dot-matrix printers, Desk Jet printers or other ink-based printers, and lower-end Laser printers all belong in the raster-based printer category, although there are still significant differences among them.

Data in a raster-based printer control language is usually converted to printer resolution, and in the printer color space. For example, a HP DeskJet printer may accept data in 600-dpi and 1-bit black/2-bit CMY format, which means each square inch contains 600 by 600 pixels, each with 7 bits worth of information. Raster data to the printer is arranged in a planar scan line structure, compressed in a certain format, and divided into a sequence of commands.

A raster-based printer accepts data strictly in top-down order. As enough data to print a swath is received, it's printed on paper, and the printer is ready to accept new data, until a page is finished. A raster-based printer can have a smaller memory to hold a fraction of a page worth of raster data, and a simpler firmware.

For a raster-based printer, a printer driver has to do all the work of converting drawing primitives sent from the application to raster images, on a per-band basis. With the help of GDI, such a printer driver divides a page into bands. The drawing commands for each band are rendered into a raster bitmap, halftoned, converted to the printer color space, and encapsulated in the printer control language. For landscape orientation, a page is divided into vertical instead of horizontal bands, and the raster data after rendering will be rotated 90 degrees.

A printer driver for a raster-based printer can be fairly complicated. Significant amounts of host-machine resource can be used to convert GDI commands to high-resolution, high-color-depth raster data. The amount of data and time needed to transmit it to the printer increases significantly as the printer resolution goes higher.

Here is sample data in the PCL3 printer control language used by HP DeskJet printers.

```
PCL_RESET      "<1B>E"
PJL_ENTER_PCL3GUI    "<1B>@PJL ENTER LANGUAGE=PCL3GUI<0D><0A>"
PCL_RESET      "<1B>E"

CmdStartDoc      "<1B>&u600D<1B>*o5W<0409000000>" 
PCL_US_LETTER:    "<1B>&I2A"
PCL_MEDSOURCE_TRAY1    "<1B>&I1H"
PCL_MEDSOURCE_PRELOAD  "<1B>&I-2H"
PCL_MEDIA_PLAIN     "<1B>&I0M"
PCL_PQ_NORMAL       "<1B>*o0M"
PCL_CRD_K662_C334    "<1B>*g26W<0204025802580002012C012C0004012C>
                      "<012C0004012C012C0004>" 
PCL_ORIENT_PORTRAIT   "<1B>&I0O"

CmdStartPage      "<1B>&I0E<1B>*p0y0X<1B>&I0L<1B>*r1A"

Raster Data      "<1B2A62306D32393779326D313776AC00>" 
                  "<0103C0EA0003FFFC00C0EA000103C031>"
```

"<3776AC000103C0EA0003FFFC00C0EA00>"

...

CmdEndPage "<1B>*rC<0C>"

PJL_EXIT_LANGUAGE "<1B>%%-12345X"

Each PCL command starts with the escape character in the ASCII character set (0x1B in C notation). A printing page in PCL starts with about a dozen initialization commands to inform the printer about language variation, paper size, media source, media type, print quality mode, orientation, etc. These are followed by a body of encoded raster data, and finally some exit commands.

Page Description Language

The third class of printer control languages accepts text and vector graphics data, together with raster bitmap data. Essentially, such a printer control language provides a way to describe a page using a variety of geometrical shapes, texts, color, and rendering operations, much like the use of GDI commands. Such a printer control language is normally referred to as page description language. PostScript and PCL5, PCL6 with vector graphics support belong to this class.

Printers supporting a high-end page description language need to be much more powerful than those supporting a raster-based printer control language. Using generic graphics primitives to describe a page means there is no particular order in which drawing commands appear in a data stream. A printer needs to have enough memory to accept a pageful of drawing primitives, which will be sorted according to the order in which they appear on a page, rendered into raster format, halftoned, and sent to the print head. Essentially, what's inside a raster-based printer driver is moved to the printer. Firmware in the printer must also handle all the text, either using a font cartridge on the printer, or downloading TrueType fonts from the host computer.

A printer driver for a printer supporting a high-end page description language is simpler in some respects, because rendering to raster is normally not needed on that host. You can expect printing to be faster and less demanding on the host computer. But mapping GDI commands to commands in page description language can get very involved sometimes. Supporting the device fonts, font substitution, and font download adds more complexity.

Here is a sample PostScript file generated by an HP Color LaserJet PostScript driver.

```
<1B>
%-12345X@PJL JOB
@PJL ENTER LANGUAGE = POSTSCRIPT
%!PS-Adobe-3.0
%%Title: Document
%%Creator: Pscript.dll Version 5.0
%%Orientation: Portrait
%%PageOrder: Special
%%TargetDevice: (HP Color LaserJet 8500) (3010.104) 1
%%LanguageLevel: 3
%%EndComments

...
```

```
%%IncludeResource: font TimesNewRomanPSMT
F /F0 0 /256 T /TimesNewRomanPSMT mF
/F0S53 F0
[83 0 0 -83 0 0 ] mFS
F0S53 setfont

650 574 moveto
(Printer Control Language)[47 28 23 41 23 37 28 21 55 42 41 23 28
42 23 21 50 37 41 41 41 37 41 0]xshow
showpage
(%[%[Page: 1]%%) =
%%PageTrailer

%%Trailer
%%BoundingBox: 12 12 600 780
%%Pages: 1
(%[%[LastPage]%%) =
%%EOF
<1B>%-12345X@PJL EOJ
<1B>%-12345X
```

After a long PostScript header for printer features, macros, and abbreviation definitions, GDI drawing commands are translated to PostScript commands one by one. The *setfont* operator selects Times New Roman as the current font, the *moveto* operator specifies the drawing position, the *xshow* operator is for text drawing with precise character position, and the *showpage* operator starts printing a page.

Printing Directly to a Port

As mentioned before, raw printer data generated by a printer driver are sent by a port monitor to a port I/O driver, and then to a printer. The port monitor uses normal Win32 file operation to open a handle to an I/O driver and send data to it. An application can do the same to talk to the printer directly if it knows the printer control language.

The following code shows how to dump a file to a printer port.

```
BOOL SendFile(HANDLE hOutput, const TCHAR * filename, bool bPrinter)
{
    HANDLE hFile = CreateFile(filename, GENERIC_READ, FILE_SHARE_READ,
        NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

    if ( hFile==INVALID_HANDLE_VALUE )
        return FALSE;

    char buffer[1024];

    for (int size = GetFileSize(hFile, NULL); size; )
    {
```

```
DWORD dwRead = 0, dwWritten = 0;
if ( ! ReadFile(hFile, buffer, min(size, sizeof(buffer)),
    & dwRead, NULL) )
break;

if ( bPrinter )
    WritePrinter(hOutput, buffer, dwRead, & dwWritten);
else
    WriteFile(hOutput, buffer, dwRead, & dwWritten, NULL);

size -= dwRead;
}

return TRUE;
}

void Demo_WritePort(void)
{
KFileDialog fd;

if ( fd.GetOpenFileName(NULL, "prn", "Raw printer data") )
{
HANDLE hPort = CreateFile("lpt1:", GENERIC_WRITE, FILE_SHARE_READ,
    NULL, OPEN_EXISTING, FILE_ATTRIBUTE_NORMAL, NULL);

if ( hPort!=INVALID_HANDLE_VALUE )
{
    SendFile(hPort, fd.m_TitleName, false);
    CloseHandle(hPort);
}
}
}
```

The SendFile routine opens a handle to an input file and copies data block by block from the file to the output file. The Demo_WritePort routine shows a dialog box for the user to select a file to send to the printer, creates a handle to an I/O port, and calls SendFile.

Writing raw data in printer control language can be useful for applications ported from DOS programs, applications that need only text printing, and applications that think they can do a better job than normal printer drivers. An application could also save raw printer data generated by a printer driver, through printing to a file, and reuse it by sending it directly to the printer port without involving GDI.

On Windows NT/2000, the port LPT1: is not a true hardware port, although WriteFile on it does go through the same system service call as writing to a true hardware port. If you use tools like WINOBJ (www.sysinternals.com), you will find that LPT1 is a symbolic link to "Device\NamedPipe\Spooler\LPT1." After all, it's still controlled by the printing spooler. A device that looks like a real device is NONSPOOLED_LPT1, which is a symbolic link to "\Device\Parallel0." When the spooler is not running—for example, after you shut it down with "net stop spooler"—opening "LPT1:" fails.

Printing through the Spooler

Another way of printing to a printer is using the print spooler API. Win32 provides a rich set of print spooler functions that can be called from an application to query spooler status, control printing job, configure printers, or send data directly to the printer. For most Windows applications, there is no need to call these functions directly, as the commonly used features are available through printing-related common dialog boxes and the printer applet in the control panel. So we will just discuss a few print spooler functions here.

```
BOOL OpenPrinter(LPTSTR pPrinterName, LPHANDLE phPrinter,  
    LPPRINTER_DEFAULTS pDefault);  
DWORD StartDocPrinter(HANDLE hPrinter, DWORD Level, LPBYTE pDocInfo);  
BOOL StartPagePrinter(HANDLE hPrinter);  
BOOL WritePrinter(HANDLE hPrinter, LPVOID pBuf, DWORD cbBuf,  
    LPDWORD pcWritten);  
BOOL EndPagePrinter(HANDLE hPrinter);  
BOOL EndDocPrinter(HANDLE hPrinter);  
BOOL ClosePrinter(HANDLE hPrinter);  
BOOL AbortPrinter(HANDLE hPrinter);
```

Given a printer name and a pointer to a PRINTER_DEFAULTS structure for default settings, OpenPrinter returns a handle to a printer object maintained by the Win32 spooler DLL. Note that a spooler printer handle is not a kernel object handle, nor a GDI object handle, so it can be used only in spooler functions. For current implementation on Windows NT/2000, a spooler printer handle is simply a pointer in the user mode address space.

The StartDocPrinter function informs the print spooler that a new document is to be spooled for printing. The last parameter is a pointer to either DOC_INFO_1 or DOC_INFO_2 structure, which specifies the document name, output file name, and the spool data type for the print job. The spooler DLL creates a new printing job in the spooler using AddJob, and creates a spooler file in the spooler directory. The corresponding closing call is EndDocPrinter, which ends a print job, removes it from the spooler, and frees all allocated resources.

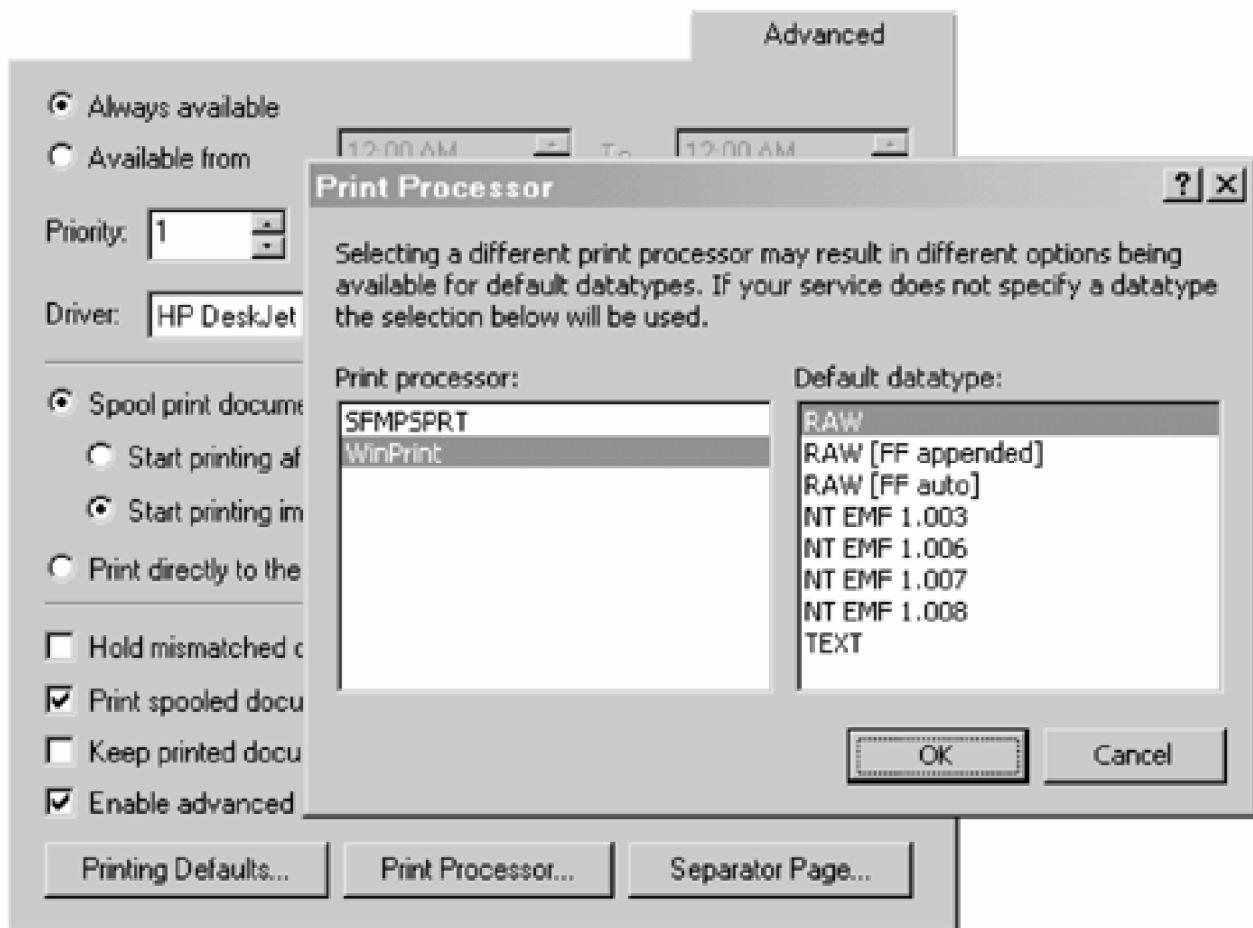
The StartPagePrinter tells the spooler that a new page is going to be generated; after that, an application can use WritePrinter to write data to the spooler. The closing call for StartPagePrinter is EndPagePrinter, after which a new page can be started or a printing job finished.

If an error occurs, function AbortPrinter can be called to abort a print job.

These spooler functions are exported from the Win32 spooler client DLL WINSPOOL.DRV, such that Windows programs can communicate with the spooler. But the actual spooler implementation is in a separate system process (SPOOLSV.EXE). The Spooler client DLL communicates with the spooler using remote procedure calls. For example, StartPagePrinter calls RpcStartPagePrinter, which in turn calls NdrClientCall2.

You can use the spooler functions to send a raw print job to a printer. But with the full printer spooler support, you can do much more. To be more precise, you can send any data as long as it's supported by a print processor, which is actually responsible for handling the data. To check the data format supported by a print processor associated with a printer driver, go to the printer property sheet, and check under the advanced tab. [Figure 17-1](#) shows what's supported by the default Windows 2000 print processor.

Figure 17-1. Data types supported by default Win2000 print processor.



As you can see from [Figure 17-1](#), the default Windows 2000 print processor supports three classes of spool data types: raw data, text, and NT EMF. Although there are four versions of NT EMF listed, there is no documentation on their exact specifications and differences. Another thing to note is that the WinPrint name listed is not a physical DLL name. The default Windows 2000 print processor is part of the local print processor (LOCALSPL.DLL), as can be seen from its exported entry points like `PrintDocumentOnPrintProcessor`.

Without actually trying it, anyone may doubt that the documented spooler API is able to handle an EMF spool file. Here is a routine to test the spooler functions' handling of an EMF spool file.

```
void Demo_WritePrinter(void)
{
    PRINTEFFECT pd;
    memset(&pd, 0, sizeof(PRINTEFFECT));
    pd.lStructSize = sizeof(PRINTEFFECT);

    if ( PrintDlg(&pd)==IDOK )
    {
        HANDLE hPrinter;
```

```
DEVMODE * pDevMode = (DEVMODE *) GlobalLock(pd.hDevMode);

PRINTER_DEFAULTS prn;
prn.pDatatype = "NT EMF 1.008";
prn.pDevMode = pDevMode;
prn.DesiredAccess = PRINTER_ACCESS_USE;

if ( OpenPrinter((char *) pDevMode->dmDeviceName,
    & hPrinter, & prn) )
{
    KFileDialog fd;

    if ( fd.GetOpenFileName(NULL, "spl",
        "Windows 2000 EMF Spool file") )
    {
        DOC_INFO_1 docinfo;

        docinfo.pDocName = "Testing WritePrinter";
        docinfo.pOutputFile = NULL;
        docinfo.pDatatype = "NT EMF 1.008";

        StartDocPrinter(hPrinter, 1, (BYTE *) & docinfo);
        StartPagePrinter(hPrinter);

        SendFile(hPrinter, fd.m_TitleName, true);

        EndPagePrinter(hPrinter);
        EndDocPrinter(hPrinter);
    }

    ClosePrinter(hPrinter);
}

if ( pd.hDevMode ) GlobalFree(pd.hDevMode);
if ( pd.hDevNames ) GlobalFree(pd.hDevNames);
}
```

The Demo_WritePrinter function starts by using the print common dialog box to let the user select a printer to print to. If the OK button is clicked in the print dialog box, the PRINTEDLG structure will be filled with a global handle to a DEVMODE structure, which captures all printing settings. The DEVMODE handle is converted to a pointer and used to fill the PRINTER_DEFAULTS structure required by the Open Printer function. If the OpenPrinter call is successful, the user is queried to choose a Windows NT/2000 EMF spool file. The routine then calls StartDocPrinter, using “NT EMF 1.008” as the spool data type, and sends the contents of the EMF spool file, using the SendFile routine we wrote before. If everything is right, the EMF spool file will be despoiled to the specified printer's printer driver and printed.

The key to successfully using the Demo_WritePrinter routine is getting the right EMF spool file. In [Chapter 16](#) on the

EMF, we mentioned a way to get EMF spool file using the EMFScope tool on Windows 95/98 and using the spooler keeping-spool-file feature on Windows NT/2000. Also a tool is provided to decode the EMF file and the EMF spool file.

The method demonstrated here can be used by an application to send either raw print data or spooled EMF data to the printer without explicitly involving GDI. Combined with the ability to get the EMF spool file from the spooler, the application can reuse the EMF spool file without the original application generating these spool files. This can be very useful to implement some generic document-handling features, which can handle output from different applications in a uniform way. Note that the EMF spool file should be compatible with the printer you're printing to, because the GDI spool file is generated with the original printer driver device context as the reference device context. Also note that in the Demo_WritePrinter routine, we are using the DEVMODE structure from current printing setting. A better way is to extract the original DEVMODE structure from the SHD file generated together with the EMF spool file. Certain fields in the DEVMODE can be changed without regenerating the EMF spool file.

What's more significant about this little demonstration is its helping our understanding of how EMF spooling is implemented. When an application starts a printing job using GDI functions, if GDI decides EMF spooling is allowed, it can use similar methods to create a new EMF spool job in the spooler. For every GDI function call, its EMF record data can be written to the spooler using WritePrinter. GDI's StartDoc call should call StartDocPrinter, StartPage should call StartPagePrinter, etc. The print processor will handle the despooling. Of course, this is just a simple conceptual view.

We will discuss more about the print dialog box and DEVMODE when discussing printing using GDI.

EMF Print Processor

Sending EMF spool data to the spooler using WritePrinter raises another interesting topic: what does an EMF print processor actually do? [Section 2.4](#) of this book provides a quite detailed description of how a print processor works in the context of Windows printing architecture. Here let's take a brief look at what's actually in the Windows 2000 print processor.

The print processor is a nicely envisioned customizable component in the Windows printing architecture. But before Windows 2000 it was not widely used, because there are not many things you can do with it. When a print processor accepts an EMF spool job from the spooler, it can only despool it back to the printer driver using a GDI function, GdiPlayEMF (gdiPlaySpoolStream on Windows 95/98).

Here is the definition for GdiPlayEMF for Windows NT 4.0.

```
BOOL GdiPlayEMF(LPWSTR pwszPrinterName, LPDEVMODEW pDevmode,  
LPWSTR pwszDocName, EMFPLAYPROC pfnEMFPlayFn, HANDLE hPageQuery);
```

As you can see, the print processor gets only the printer name, DEVMODE, and the document name, and nothing else. The only contribution it can make is calling Gdi PlayEMF multiple times to implement the multiple-copy feature. The last two parameters are designed for selectively playing back individual EMF pages, but this is not implemented on Windows NT 4.0.

Windows 2000 expands the capability of a print processor by allowing it to control the playing back of individual EMF pages, with the ability to combine multiple logical pages into one physical page (N-up), reorder pages within a document, or even add transformation to logical pages. With the empowerment from GDI, a Windows 2000 print processor can print multiple pages per physical page (N-up printing), print pages in reverse order, print multiple

copies of each page, print booklets, and print page borders. Such document-handling features can be implemented in a central place, which can be easily extended without changing GDI and printer drivers.

The new print processor power is achieved through a new set of functions exported by GDI. Here are the three key functions:

```
HANDLE GdiGetPageHandle(HANDLE SpoolFileHandle, DWORD Page,
LPDWORD pdwReserved);
BOOL GdiPlayPageEMF(HANDLE SpoolFileHandle, HANDLE hEmf,
RECT * prectDocument, RECT * prectBorder);
HDC GdiGetDC(HANDLE SpoolFileHandle);
```

Function GdiGetPageHandle allows a print processor to query for a handle to a specific page within an EMF spool file. Again, such a handle is not a normal GDI or kernel handle, not even a pointer to EMF bits. The page handle can be used by Gdi PlayEMF to play one page to the printer driver. The prectDocument parameter allows scaling a logical EMF page into one portion of a physical page to implement N-up or book let printing. The optional prectBorder parameter specifies a rectangle for drawing a paper border, useful for marking multiple logical pages on a physical page.

The GdiGetDC function returns an authentic GDI device context handle to the print processor, which can be used by it to apply world transformation before playing back an EMF. With world transformation, a print processor can rotate, or flip the pages around.

Sample EMF print processor source code can be found in Windows NT 4.0 and Windows 2000 DDK, under the src\print\genprint directory. Special GDI functions for the EMF print processor are documented in DDK.

Someone may ask, how can GdiPlayPageEMF actually work? If multiple logical pages are allowed to be mapped to a single physical page for N-up printing, individual EMF pages cannot be played to a device context, especially for printer drivers that need GDI banding support, except when there is another level of EMF generation and playing back. GdiPlayPageEMF does not play EMF to a printer device context; instead it just records a new logical page for a physical page in some internal GDI data structure. Real playing back of multiple logical pages starts only when GdiEndPageEMF is called.

If you step into GdiEndPageEMF under a debugger, you can find interesting dynamics of printing under GDI. To trace into GdiEndPageEMF, attach a debugger to the spooler process from the task manager, by right-mouse clicking on the spooler process and selecting the "Debug" menu item. Once attached to the spooler process, set a breakpoint at location_GdiEndPageEMF@8 in module gdi32.dll. Now you are ready to start a printing job.

You can find that GdiEndPageEMF calls GDI function StartPage, an internal function StartBanding, then goes through a loop of calling functions InternalGdiPlayPageEMF and NextBand. InternalGdiPlayPageEMF sets up world transformation and calls an interesting function, PrintBand. PrintBand calls a system service called Nt GdiGetPerBandInfo, which corresponds to a documented printer driver entry point for GDI to query for band information. PrintBand then calls PlayEnhMetaFile to actually play an EMF to the printer driver device context. Somewhere there must be a loop to handle multiple logical pages on a physical page. Now we should have a much clearer picture of how an EMF plays a central role in printing, especially for Windows 2000.

Enumerating Printers

The Win32 spooler API provides the EnumPrinter function for an application to enumerate printers and query for printer information. EnumPrinter is a complicated function with lots of different options and returns different types of structures. It can be used to enumerate local printers, print providers, domain names, and all printers and print servers in the computer's domain name. It can fill an array of PRINTER_INFO_1 to PRINTER_INFO_5 structures. PRINTER_INFO_2 provides the most complete information about a printer, using 21 fields, including server name, printer name, shared name, driver name, DEVMODE, separate file, print processor, spool data type, security descriptor, etc.

Refer to MSDN for full details on EnumPrinter. Here is a sample code to enumerate all local printers and remote printer connections.

```
void * EnumeratePrinters(DWORD flag, LPTSTR name, DWORD level,
    DWORD & nPrinters)
{
    DWORD cbNeeded;

    nPrinters = 0;
    EnumPrinters(flag, name, level, NULL, 0, & cbNeeded, & nPrinters);

    BYTE * pPrnInfo = new BYTE[cbNeeded];

    if ( pPrnInfo )
        EnumPrinters(flag, name, level, (BYTE *) pPrnInfo, cbNeeded,
            & cbNeeded, & nPrinters);

    return pPrnInfo;
}

void ListPrinters(HWND hWnd)
{
    DWORD nPrinters;

    PRINTER_INFO_5 * pInfo5 = (PRINTER_INFO_5 *)
        EnumeratePrinters(PRINTER_ENUM_LOCAL, NULL, 5, nPrinters);

    if ( pInfo5 )
    {
        for (unsigned i=0; i<nPrinters; i++)
            SendMessage(hWnd, LB_ADDSTRING, 0,
                (LPARAM) pInfo5[i].pPrinterName);

        delete [] (BYTE *) pInfo5;
    }

    PRINTER_INFO_1 * pInfo1 = (PRINTER_INFO_1 *)
        EnumeratePrinters(PRINTER_ENUM_CONNECTIONS, NULL, 1, nPrinters);
```

```
if ( pInfo1 )
{
    for (unsigned i=0; i<nPrinters; i++)
        SendMessage(hWnd, LB_ADDSTRING, 0, (LPARAM) pInfo1[i].pName);
    delete [] (BYTE *) pInfo1;
}
```

The `EnumeratePrinters` function is a simple wrapper around the `EnumPrinters` function, to manage size query and memory allocation. Function `ListPrinters` calls `EnumeratePrinters` to enumerate local printers first and then printer connections. Their names are added to a list box, which can be used for the user to select the printer.

Printer enumeration can be used by applications to provide their own print dialog box. For example, a DirectX game or educational title would not like the normal Windows common dialog box. Some applications need to provide a quick access to the default printer without user selection. The `GetDefaultPrinter` function can be used to get the current default printer name. But it is provided on Windows 2000 only.

```
BOOL GetDefaultPrinter(LPTSTR pszBuffer, LPDWORD pcchBuffer);
```

Querying a Printer

Given a printer handle, the function `GetPrinter` can be used to query lots of detailed information about a printer. `GetPrinter` is able to return `PRINTER_INFO_1` through `PRINTER_INFO_9`.

An application can also call `DeviceCapabilities` to query the device driver user interface module about information like paper bins, collating support, duplex support, private `DEVMODE` size, supported papers, paper names, N-up support, printing speed, etc.

Refer to MSDN about `GetPrinter` and `DeviceCapabilities`.

Set Up a Printer Driver

Various settings for printing are captured by the `DEVMODE` structure. To be more precise, the `DEVMODE` structure is used by all graphics devices for communicating its settings among the application, GDI, and the device driver. The last parameter to `CreateDC` or `CreateIC` functions is a pointer to a `DEVMODE` structure. But for printer devices the `DEVMODE` structure is much more important than for display devices.

Here is the definition for the `DEVMODE` structure.

```
typedef struct _devicemode {
    TCHAR dmDeviceName[CCHDEVICENAME];
    WORD dmSpecVersion;
    WORD dmDriverVersion;
    WORD dmSize;
    WORD dmDriverExtra;
```

```
DWORD dmFields;
union {
    struct {
        short dmOrientation;
        short dmPaperSize;
        short dmPaperLength;
        short dmPaperWidth;
    };
    POINTL dmPosition;
};

short dmScale;
short dmCopies;
short dmDefaultSource;
short dmPrintQuality;
short dmColor;
short dmDuplex;
short dmYResolution;
short dmTTOption;
short dmCollate;
BCHAR dmFormName[CCHFORMNAME];
WORD dmLogPixels;
DWORD dmBitsPerPel;
DWORD dmPelsWidth;
DWORD dmPelsHeight;
union {
    DWORD dmDisplayFlags;
    DWORD dmNup;
}
DWORD dmDisplayFrequency;
DWORD dmICMMETHOD;
DWORD dmICMIntent;
DWORD dmMediaType;
DWORD dmDitherType;
DWORD dmReserved1;
DWORD dmReserved2;
DWORD dmPanningWidth;
DWORD dmPanningHeight;
} DEVMODE;
```

The DEVMODE structure is a very complicated data structure for several reasons. It is a variable-size structure with different sizes for different Window versions, and interpretations for its fields are still evolving. The device driver can attach extra data after the public members of the DEVMODE structure to keep extra settings needed by a driver, so an application should query for the size of a DEVMODE structure and allocate on the heap, instead of assuming a fixed size and allocating on the stack.

The dmDeviceName field specifies the “friendly name” of a printer, as it appears in your control panel printer applet. Note that it's truncated to 32 characters. The dmSpecVersion is a version identifier for the DEVMODE structure being used. WINGDI.H defines DM_SPECVERSION macro to its current version, now at 0x401. The dmDriverVersion is the internal driver version assigned by the driver designer. For example, Windows 2000

UniDriver-based drivers use 0x500.

The dmSize field specifies the byte size of public members in the DEVMODE structure. If you're creating a new DEVMODE structure, you should set it to size of(DEVMODE). But if you get a DEVMODE structure from somewhere else, do not assume dmSize has the same value as your current sizeof(DEVMODE), because you may use the latest Win32 header file to compile your program and run it on an older machine with an old driver supporting a previous version of the DEVMODE specification, or vice versa. The dmDriverExtra field specifies the extra amount of space needed by the device driver for storing private fields after the public DEVMODE fields. If there are no extra fields, assign it to zero. The total space needed by a DEVMODE structure is dmSize + dmDriverExtra.

The dmFields field specifies whether certain fields have been initialized. Different flags are used to identify different fields; for example, DM_ORIENTATION is for the dmOrientation field.

The remaining DEVMODE fields are mostly for device settings. The dmPrintQuality field normally determines the printing quality setting, which is very critical to print quality, printing speed, printer data size, etc. The main way to specify print quality is using the predefined DMRES_HIGH, DMRES_MEDIUM, DMRES_LOW, or DMRES_DRAFT macros. A printer driver normally reports different resolutions to GDI depending on the dmPrintQuality field, to scale the amount of data received from GDI. It also controls how drawing commands are rendered in the printer driver. The values for these macros are from -4 to -1. A printer driver can change dmPrintQuality's value to the actual DPI being used. Refer to MSDN for full details.

Given a printer name and a printer handle, the DocumentProperties function can be used to set up a DEVMODE structure from scratch.

```
LONG DocumentProperties(HDC hWND, HANDLE hPrinter, LPTSTR pDeviceName,  
PDEVMODE pDevModeOutput, PDEVMODE pDevModeInput, DWORD fMode);
```

The last parameter, fMode, controls the operation the function performs. If fMode is 0, the combined size of both public and private DEVMODE fields is returned. If it's DM_OUT_BUFFER, the DEVMODE structure pointed to by pDevModeOutput is filled with the current default DEVMODE setting for a driver. If it's DM_IN_BUFFER, pDevModeInput points to a DEVMODE structure containing input settings. If it's DM_IN_PROMPT, the printer driver's print setup property sheet is displayed to allow the user to change printing settings.

The following GetDEVMODE routine illustrates how to use Document Properties.

```
DEVMODE * GetDEVMODE(TCHAR * PrinterName, int nPrompt)  
{  
    HANDLE hPrinter;  
  
    if ( !OpenPrinter(PrinterName, &hPrinter, NULL) )  
        return NULL;  
  
    // A zero for last param returns the size of buffer needed.  
    int nSize = DocumentProperties(NULL, hPrinter, PrinterName,  
        NULL, NULL, 0);  
  
    DEVMODE * pDevMode = (DEVMODE *) new char[nSize];
```

```
if ( pDevMode==NULL )
    return NULL;

// ask driver to initialize a devmode structure
DocumentProperties(NULL, hPrinter, PrinterName, pDevMode,
    NULL, DM_OUT_BUFFER);

// show property sheet to allow user modification
BOOL rslt = TRUE;
switch ( nPrompt )
{
case 1:
    rslt = AdvancedDocumentProperties(NULL, hPrinter,
        PrinterName, pDevMode, pDevMode) == IDOK;
    break;
case 2:
    rslt = ( DocumentProperties(NULL, hPrinter, PrinterName,
        pDevMode, pDevMode, DM_IN_PROMPT | DM_OUT_BUFFER |
        DM_IN_BUFFER ) == IDOK );
    break;
}

ClosePrinter(hPrinter);

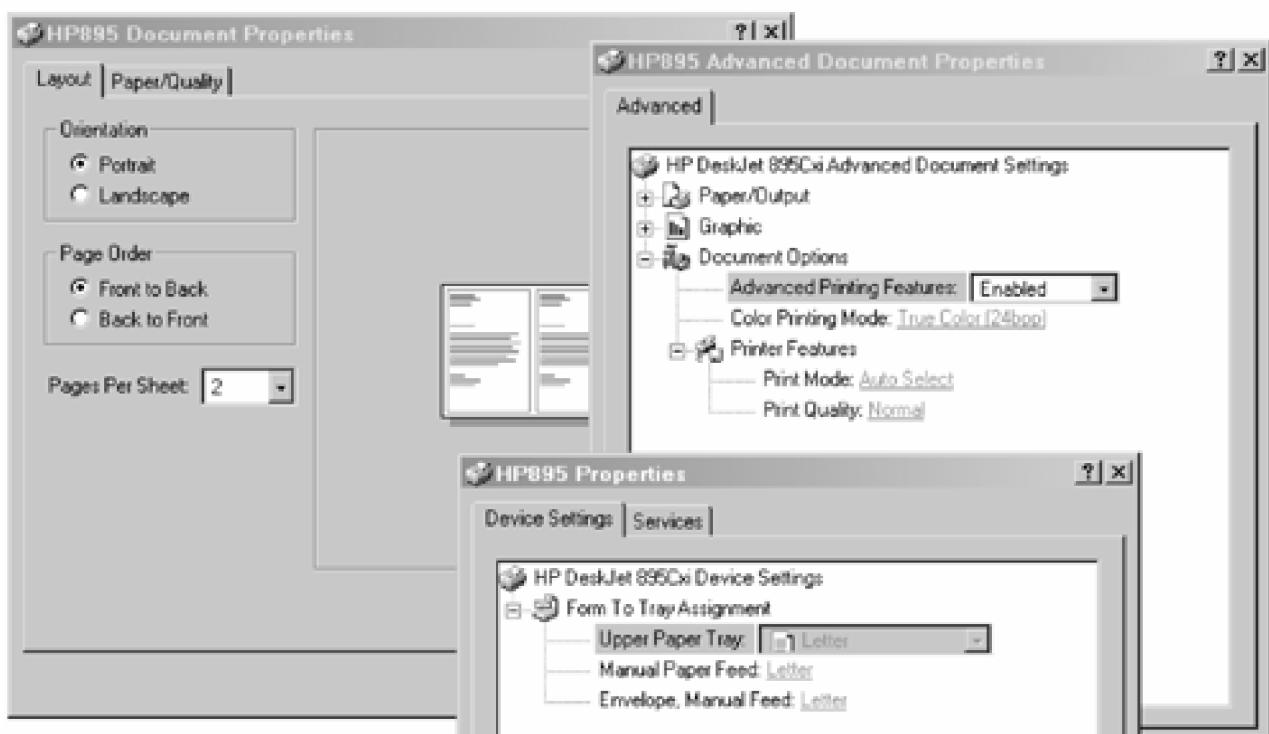
if ( rslt )
    return pDevMode;
else
{
    delete [] (BYTE *) pDevMode;
    return NULL;
}
```

GetDEVMODE starts by calling DocumentProperties to query for the actual DEV MODE structure size. After allocation, it calls DocumentProperties again to get the current default DEVMODE setting as the user specifies in the control panel. The last parameter to GetDEVMODE specifies whether the user should be prompted to change the print setting using the printer driver's setup property sheet, or just its "advanced" property sheet page.

Function AdvancedDocumentProperties is also a function provided by Win32 spooler client DLL. Another related function is PrinterProperties, which displays the printer-properties property sheet for a specified printer.

[Figure 17-2](#) shows sample screen displays for DocumentProperties, AdvancedDocumentProperties, and PrinterProperties.

Figure 17-2. Accessing driver setup property sheets.



All these property sheets shown in [Figure 17-2](#) are implemented by the printer driver user interface DLL. On Windows 95/98, printer drivers are 16-bit DLLs loaded by 16-bit GDI module GDI.EXE. The user interface and core driver can be in the same DLL. For Windows NT 4.0, the core printer driver is a kernel mode DLL, while the user interface portion needs to be a user mode DLL, so they are always separate DLLs. For Windows 2000, although the system allows a user mode printer driver, the user interface DLL is still separate from the core printer driver.

The implementation of these three functions loads the printer user interface DLL into the application process address space to query for driver settings, or display printer-specific property sheets. Any new DLL used by it will also be loaded. Loaded DLLs are normally unloaded before these functions return. Some optimizations seem to be implemented on Windows 2000 for simple queries, but still displaying property sheets could load over a dozen mostly system DLLs.

17.2 BASIC PRINTING USING GDI

The spooler functions described in the last section provide the application with interfaces to the spooler and the printer driver user interface DLL. Normally an application uses printing-related common dialog boxes to set up printers and uses GDI functions to perform printing. Under the hood, the printing common dialog boxes and GDI use spooler functions to communicate with the spooler and the printer driver.

This section will discuss the basic procedure of setting up a printing job using printing common dialog boxes and GDI printing functions.

Printing Common Dialog Boxes

Common dialog boxes are not an essential part of the core Win32 API, as all its features can be implemented using the core Win32 API. But they provide a vital standard user interface for several common tasks in using the Windows operating system. So far we have used choose color, choose font, and file open/save dialog boxes, which all prove to be rather convenient. For printing, Windows provides two common dialog boxes, print dialog box and page setup dialog box.

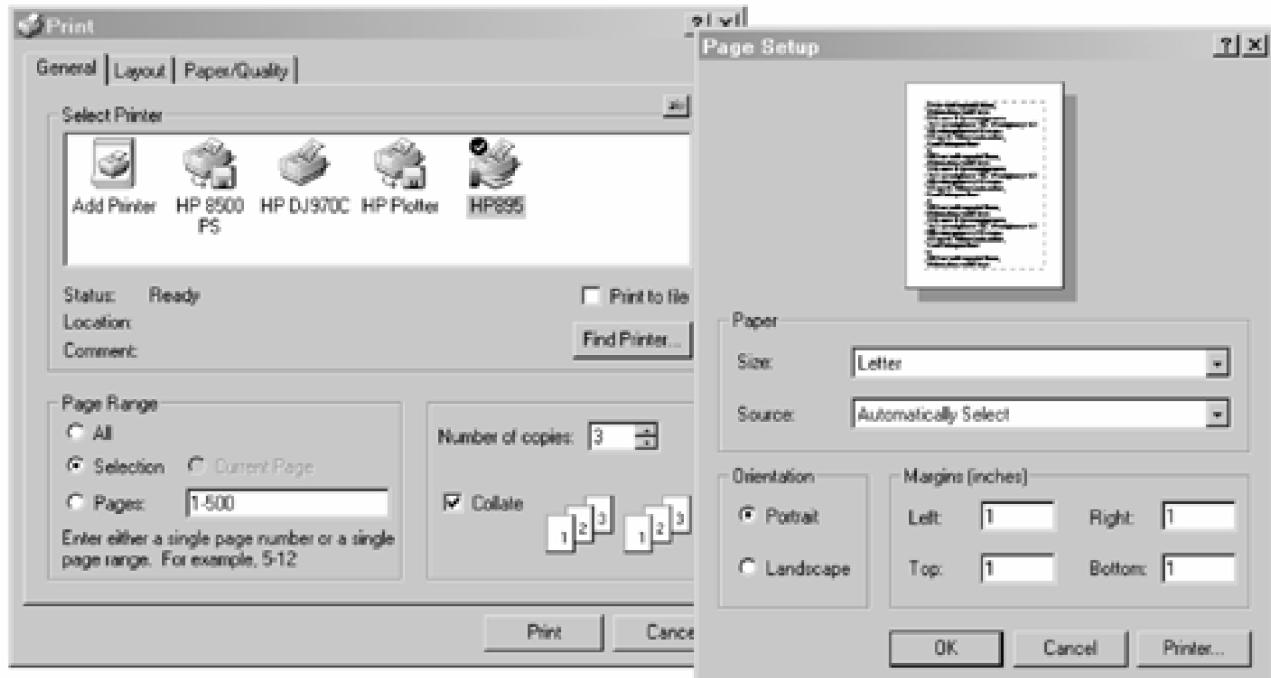
```
typedef struct tagPD {  
    DWORD      IStructSize;  
    HWND       hwndOwner;  
    HGLOBAL    hDevMode;  
    HGLOBAL    hDevNames;  
    HDC        hDC;  
    DWORD      Flags;  
    WORD       nFromPage;  
    WORD       nToPage;  
    WORD       nMinPage;  
    WORD       nMaxPage;  
    WORD       nCopies;  
    HINSTANCE  hInstance;  
    LPARAM     lCustData;  
    LPPRINTHOOKPROC lpfnPrintHook;  
    LPSETUPHOOKPROC lpfnSetupHook;  
    LPCSTR     lpPrintTemplateName;  
    LPCSTR     lpSetupTemplateName;  
    HGLOBAL    hPrintTemplate;  
    HGLOBAL    hSetupTemplate;  
} PRINTDLGA;  
  
BOOL PrintDlg(LPPRINTDLG lppd);  
  
typedef struct tagPSD
```

```
{  
    DWORD      IStructSize;  
    HWND       hwndOwner;  
    HGLOBAL    hDevMode;  
    HGLOBAL    hDevNames;  
    DWORD      Flags;  
    POINT     ptPaperSize;  
    RECT      rtMinMargin;  
    RECT      rtMargin;  
    HINSTANCE  hInstance;  
    LPARAM     lCustData;  
    LPPAGESETUPHOOK lpfnPageSetupHook;  
    LPPAGEPAINTHOOK lpfnPagePaintHook;  
    LPCSTR     lpPageSetupTemplateName;  
    HGLOBAL    hPageSetupTemplate;  
} PAGESETUPDLG;
```

```
BOOL PageSetupDlg(LPPAGESETUPDLG lppsd);
```

Function PrintDlg displays the print common dialog box, which allows the user to select the printer, choose the printing range, copy count, collation, and access to the printer setup property sheets. Function PageSetupDlg displays the page setup common dialog box, which allows the user to select paper size, paper source, orientation, and paper margins. PrintDlg can also return the current default printer setting without displaying a dialog box, if the PD_RETURNDEFAULT flag is passed. [Figure 17-3](#) shows both of them on Windows 2000.

Figure 17-3. Windows 2000 print and page setup common dialog boxes.



Function PrintDlg uses the PRINTDLG structure to accept parameters from the application, return the results to the application, and specify how the print dialog box should be customized. Its hDevMode field holds a global handle to a DEVMODE structure, which holds printing settings. The hDevNames field holds a global handle to a DEVNAMES

structure, which holds the printer driver name, the device name, and the output port name.

Global handles are inherited from the Win16 API, where all tasks running on a system share a common memory address space. Global handles are references to memory blocks allocated from the global heap. These memory blocks can even move in the heap to compact the heap space, making room for big memory blocks. So they should be locked using GlobalLock, which returns a far pointer to the memory block, and unlocked using GlobalUnlock. Win32 API still uses a few global handles, just to make it easier for porting 16-bit applications. In Win32 programs, a global memory handle and a pointer to its data block are still totally different things. GlobalLock must be called to convert a global memory handle to the pointer to its data block. But resource handles have the same values as their corresponding pointers.

The hDC field in PRINTEDLG holds a GDI device context handle or an information context handle, which is created and returned by PrintDlg if PD_RETURNDC or PD_RETURNIC flag is specified. The Flags field controls how input in the structure should be interpreted and how output data should be generated. The next four fields, nFromPage, nToPage, nMinPage, and nMaxPage, control the print page range on the bottom left part of the print dialog box, a nice feature for applications supporting multiple-page documents. Field nCopies specifies the copy count. The rest of PRINTEDLG allows customization of the print dialog box. You can frequently see software with a customized print dialog box. For example, a personal finance software may allow you to select a date range to print from, instead of pages. Refer to MSDN for full details about PRINTEDLG.

Likewise, the function PageSetupDlg uses PAGESETUP structure to accept parameters from the application, return results to the application, and specify how the page setup dialog box should be customized. In a PAGESETUP structure, hDevMode and hDev Names have the same meaning as in the PRINTEDLG structure. Similarly, the Flags field uses dozens of flags to control how PageSetupDlg works. The main value of PAGE SETUP is its ptPageSize, rtMinMargin, and rtMargin fields. The ptPaperSize field specifies the physical dimensions of the current paper size. The rtMinMargin field specifies the minimum margins allowed around the four sides of a piece of paper. Due to the physical limitation of printers, there are nonprintable areas around all sides of the paper. The rtMargin field specifies the current margin selected by the user using the dialog box, which is limited by rtMinMargin. Paper orientation selection is reflected in swapped horizontal and vertical metrics, and also in the DEVMODE structure.

Data returned by PrintDlg and PageSetupDlg is critical for the application to format a document appropriately for printing. So they should share the same DEVMODE and DEVNAMES handles to make the settings consistent. The application should also maintain per-document settings, which could be saved together with the document and recreated when a document is loaded. Some applications will ask the user to select a printer first before a document can be created. Lots of applications query the printer driver to synchronize the printing setting when loading a document.

[Listing 17-1](#) shows the declaration of the KOutputSetup class and part of its implementation, which is a wrapper class for both PrintDlg and PageSetupDlg.

Listing 17-1 KOutputSetup Class for PrintDlg and PageSetupDlg

```
class KOutputSetup
{
    PRINTEDLG m_pd;
    PAGESETUPDLG m_psd;

    void Release(void);

public:
```

```
KOutputSetup(void);
~KOutputSetup(void);

void DeletePrinterDC(void);
void SetDefault(HWND hwndOwner, int minpage, int maxpage);
int PrintDialog(DWORD flag);
BOOL PageSetup(DWORD flag);

void GetPaperSize(POINT & p) const
{
    p = m_psd.ptPaperSize;
}

void GetMargin(RECT & rect) const
{
    rect = m_psd.rtMargin;
}

void GetMinMargin(RECT & rect) const
{
    rect = m_psd.rtMinMargin;
}

HDC GetPrinterDC(void) const
{
    return m_pd.hDC;
}

DEVMODE * GetDevMode(void)
{
    return (DEVMODE *) GlobalLock(m_pd.hDevMode);
}

const TCHAR * GetDriverName(void) const;
const TCHAR * GetDeviceName(void) const;
const TCHAR * GetOutputName(void) const;

HDC CreatePrinterDC(void);
};

KOutputSetup::KOutputSetup(void)
{
    memset (&m_pd, 0, sizeof(PRINTDLG));
    m_pd.lStructSize = sizeof(PRINTDLG);

    memset(& m_psd, 0, sizeof(m_psd));
    m_psd.lStructSize = sizeof(m_psd);
}
```

```
void KOutputSetup::SetDefault(HWND hwndOwner, int minpage, int maxpage)
{
    m_pd(hwndOwner) = hwndOwner;

    PrintDialog(PD_RETURNDEFAULT);

    m_pd.nFromPage = minpage;
    m_pd.nToPage = maxpage;
    m_pd.nMinPage = minpage;
    m_pd.nMaxPage = maxpage;

    m_psd(hwndOwner) = hwndOwner;

    m_psd.rtMargin.left = 1250; // 1.25 inch
    m_psd.rtMargin.right = 1250; // 1.25 inch
    m_psd.rtMargin.top = 1000; // 1.00 inch
    m_psd.rtMargin.bottom= 1000; // 1.00 inch

    HDC hDC = CreatePrinterDC();

    int dpix = GetDeviceCaps(hDC, LOGPIXELSX);
    int dpiy = GetDeviceCaps(hDC, LOGPIXELSY);
    m_psd.ptPaperSize.x = GetDeviceCaps(hDC, PHYSICALWIDTH) * 1000 / dpix;
    m_psd.ptPaperSize.y = GetDeviceCaps(hDC, PHYSICALHEIGHT) * 1000 / dpiy;

    m_psd.rtMinMargin.left= GetDeviceCaps(hDC,PHYSICALOFFSETX)*1000 / dpix;
    m_psd.rtMinMargin.top = GetDeviceCaps(hDC,PHYSICALOFFSETY)*1000 / dpiy;

    m_psd.rtMinMargin.right = m_psd.ptPaperSize.x - m_psd.rtMinMargin.left
        - GetDeviceCaps(hDC, HORZRES) * 1000 / dpix;
    m_psd.rtMinMargin.bottom= m_psd.ptPaperSize.y - m_psd.rtMinMargin.top
        - GetDeviceCaps(hDC, VERTRES) * 1000 / dpiy;

    DeleteObject(hDC);
}

int KOutputSetup::PrintDialog(DWORD flag)
{
    m_pd.Flags = flag;
    return PrintDlg(&m_pd);
}

BOOL KOutputSetup::PageSetup(DWORD flag)
{
    m_psd.hDevMode = m_pd.hDevMode;
    m_psd.hDevNames = m_pd.hDevNames;
    m_psd.Flags = flag | PSD_INTHOUSANDTHSOFINCHES | PSD_MARGINS;

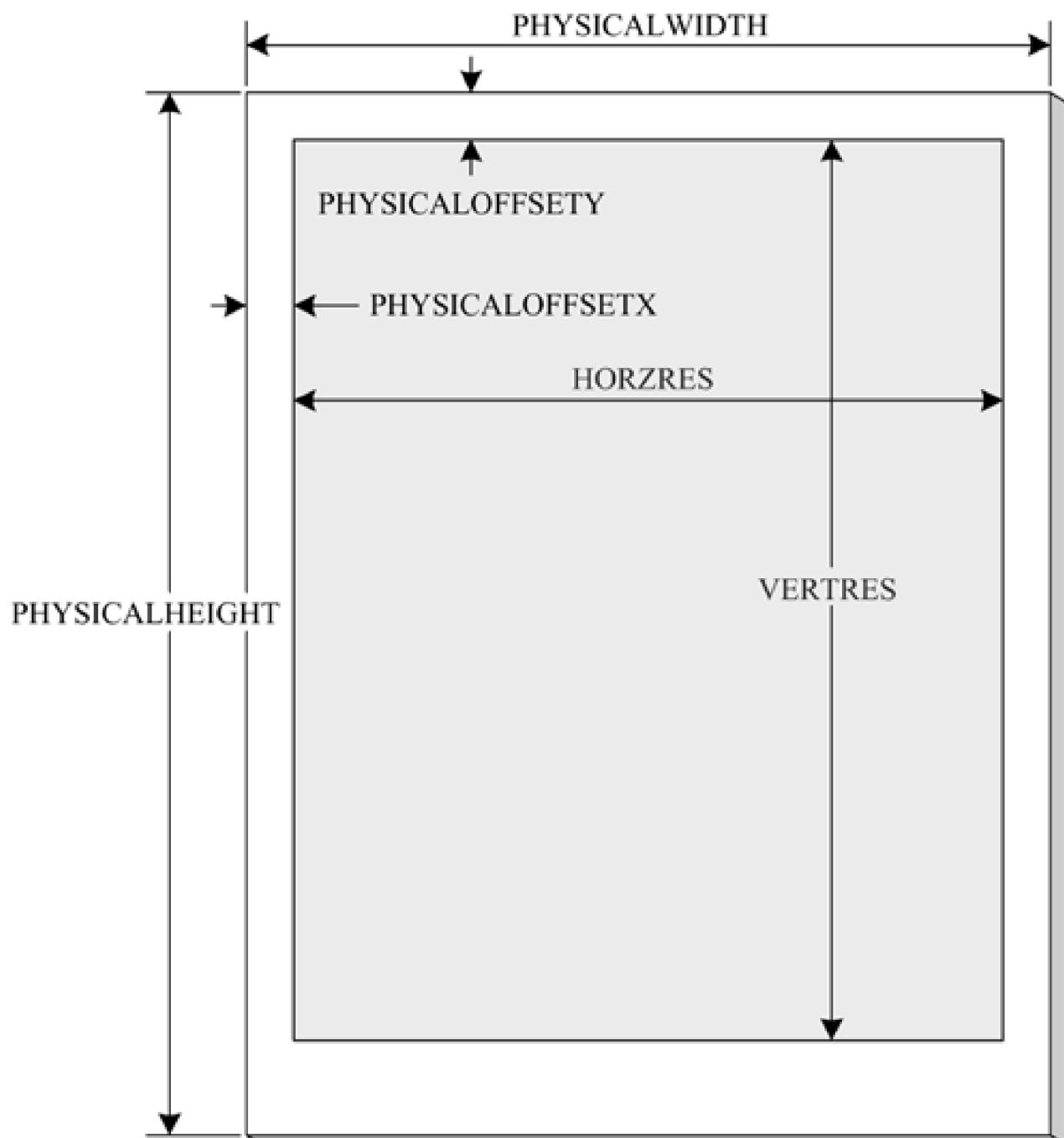
    return PageSetupDlg(& m_psd);
```

}

The KOutputSetup class is similar to MFC's wrapper classes CPrintDialog and C PageSetupDialog combined into one. Its constructor simply initializes m_pd (PRINT DLG structure) and m_psd (PAGESETUPDLG structure) member variables. The Set Default method sets up important fields without showing a dialog box. For m_pd, call PrintDlg with PD_RETURNDEFAULT returns default DEVMODE and DEVNAMES handles. The page selection fields are set from parameters to SetDefault. The default paper margin is set to be 1.25 inch left and right, and 1 inch top and bottom.

The minimum paper margin is calculated according to the physical paper dimensions, printable paper dimensions, and the physical offsets, all returned by GetDevice Caps with the right index. The physical paper dimensions, as returned using PHYSICAL WIDTH PHYSICALHEIGHT, are a measure of the physical paper size. For example, in 300 dpi, 8.5-inch by 11-inch letter-size paper has pixel dimensions of 2550 by 3300. The printable paper dimensions, as returned using HORZRES and VERTRES, are a measure of the printable area in the center of a piece of paper. The physical offsets, as returned using PHYSICALOFFSETX and PHYSICALOFFSETY, are the amount of top and bottom margin. [Figure 17-4](#) illustrates these six values returned by GetDeviceCaps.

Figure 17-4. Paper dimensions returned by GetDeviceCaps.



The PrintDialog method calls the PrintDlg function provided by the common dialog module. The PageSetup method calls the PageSetupDlg function with the same DEVMODE and DEVNAMES handles used by PrintDlg, to make sure their settings are synchronized.

Creating Printer Device Context

The PrintDlg function, and hence the KOutputSetup class, can return a printer device context, which can be used to start a printer job the GDI way. To do that, you just need to specify the PD_RETURNDC flag. Here is a simple example:

```
void Demo_OutputSetup(bool bShowDialog)
{
```

```
KOutputSetup setup;

DWORD flags = PD_RETURNDC;
if ( ! bShowDialog )
    flags |= PD_RETURNDEFAULT;

if ( setup.PrintDialog(flags)==IDOK )
{
    HDC hDC = setup.GetPrinterDC();
    // use printer DC
}
}
```

If the bShowDialog flag is true, the print dialog box will be displayed for the user to set up the printer; otherwise, the current default setting is used. In both cases, a printer device context handle is returned, which can be used. Also returned are the handles to DEVMODE and DEVNAMES structures. All these resources will be freed in KOutputSetup class' destructor.

There is no magic in the print dialog box. A printer device context is still created using the same old function CreateDC. Recall that CreateDC accepts four parameters: a driver name, a device name, an output port or file name, and a pointer to a DEVMODE structure. The prototype for CreateDC is inherited from the Win16 API, where a graphics driver is a loadable 16-bit DLL. For Win32 applications, the graphics device driver is not directly accessible to applications. The PRINTDLG structure provides all the necessary parameters to call CreateDC. We have the DEVMODE structure and DEV NAMES, which contains the driver name, the device name, and the output name.

Here is the KOutputSetup::CreatePrinterDC method, which creates a printer device context based on the DEVMODE and DEVNAMES handles.

```
HDC KOutputSetup::CreatePrinterDC(void)
{
    return CreateDC(GetDriverName(), GetDeviceName(), GetOutputName(),
                    GetDevMode());
}

const TCHAR * KOutputSetup::GetDeviceName(void) const
{
    const DEVNAMES * pDevNames = (DEVNAMES *) GlobalLock(m_pd.hDevNames);
    if ( pDevNames )
        return (const TCHAR *) ( (const char *) pDevNames +
                               pDevNames->wDeviceOffset );
    else
        return NULL;
}
```

The GetDriverName, GetDeviceName, and GetOutputName methods retrieve the driver name, the device name, and the output name from the DEVNAMES structure. Typical values for them are "winspool," a printer-friendly name for the device name, and "lpt1:" for the output name. WINSPOOL.DRV is the Win32 spooler client DLL, supporting all the spooler functions to Win32 applications.

The only essential parameter to create a printer device context is the device name. The driver name is a placeholder; an output name is not needed, as it's passed to GDI through the StartDoc function; and the DEVMODE pointer is optional. If a NULL pointer is passed as a DEVMODE pointer, the printer driver uses the current default printer setting specified in the control panel. To create a device context with a nondefault setting, a valid DEVMODE structure with the right setting must be passed.

The easiest way to create a printer device context on Windows 2000 without using the print common dialog box is to use GetDefaultPrinter and CreateDC. Here is an example to create a printer device context with the current default printer and its default settings:

```
TCHAR PrinterName[64];
DWORD Size = 64;
GetDefaultPrinter(PrinterName, &Size);

HDC hDC = CreateDC(NULL, PrinterName, NULL, NULL);
// use printer DC
DeleteDC(hDC);
```

You can also enumerate printers, query for the default DEVMODE structure, customize it, and create a printer device context with the printer name and settings of your choice.

Querying Printer Device Context

With a printer device context handle, you can use the function GetDeviceCaps to query the device-specific information about a printer device context.

Using the TECHNOLOGY index with GetDeviceCaps, you can check the printer type. The plotters return DT_PLOTTER, and other-raster based printers and even PostScript printers all return DT_RASPRINTER. Note that a new generation of plotters may also be raster-based, instead of being the traditional 8-pen plotters.

The printer device resolution as returned by using LOGPIXELSX and LOG_PIXELSY is critical for the application to set up logical device context. Unlike a display device, which uses logical resolution to make things look bigger on the screen for better viewing, one inch is truly one inch when it's printed on paper. But there are several things to keep in mind about printer resolution.

- The printer resolution changes according to printer quality settings. A printer driver may report 1200 by 1200 dpi, 600 by 600 dpi, or 300 by 300 dpi to GDI, depending on the quality and the media type setting in the DEVMODE structure.
- Paper has margins as illustrated in [Figure 17-4](#). Bear in mind that GetDeviceCaps with HORZRES and VERTRES returns the pixel width and height of the printable area in a piece of paper, not the whole paper.
- The printer driver or printer firmware may scale data received from GDI to a higher resolution for the final printing to achieve better printer quality. For example, a printer driver may report to GDI as 1200 by 1200 dpi, and scale the data to 2400 by 1200 dpi for printing.
- Resolution is an important factor in determining print quality, but there are other important factors like

source data quality, halftoning/dithering algorithm, the number of bits used to represent each color channel, multiple drops of ink, ink drop size, mechanical accuracy, ink chemistry, media-specific color adjustment, etc.

- On Windows 95/98, due to the limitation of 16-bit GDI implementation, as resolution goes higher, the maximum paper size GDI can get smaller. For example, if a driver reports 1200 dpi, the maximum paper dimension is limited to 32,767 pixels, or 27.30 inches. For 2400 dpi, it goes down to 13.65 inches.

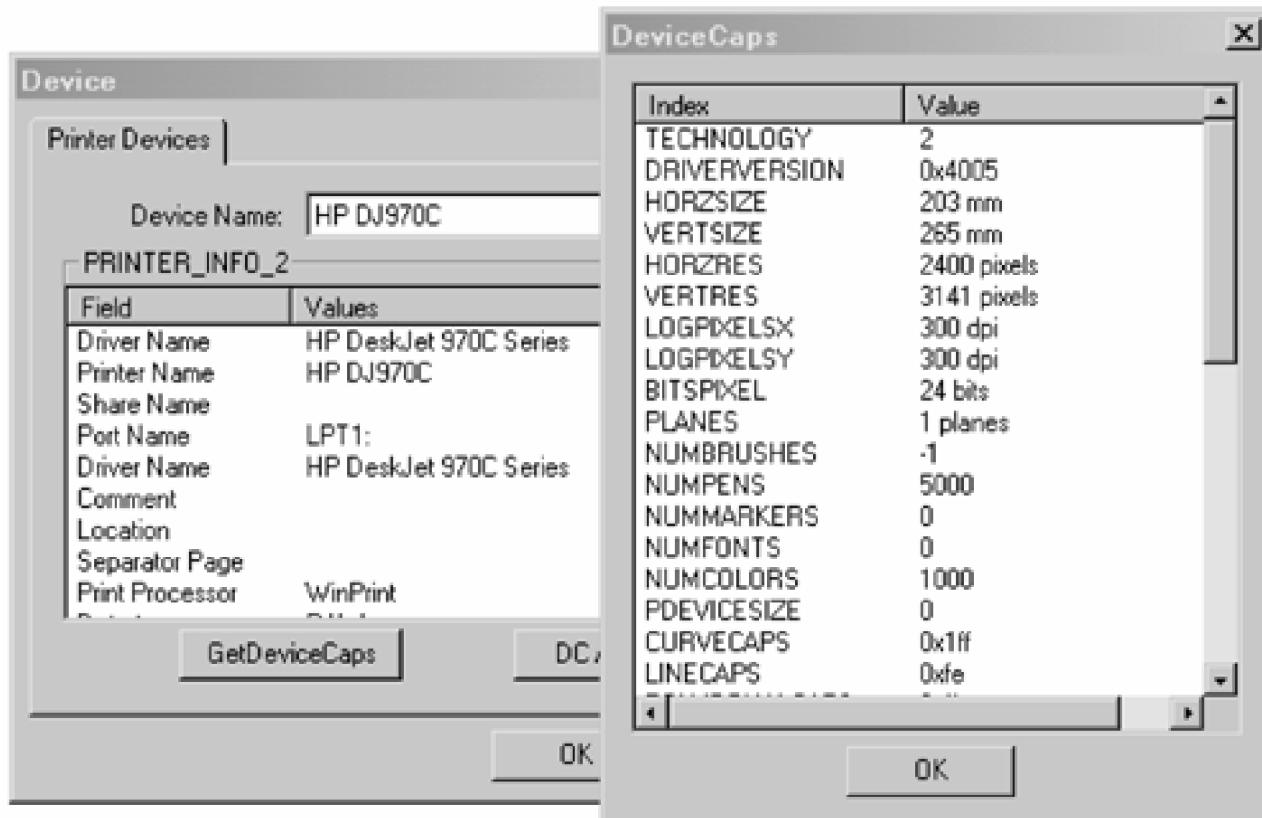
Indexes BITSPIXEL, PLANES, SIZEPALETTE, and NUMCOLORS can be used to determine a device's palette and color pixel format. You can hardly find a printer driver which supports a palette.

Indexes CLIPCAPS, RASTERCAPS, CURVECAPS, LINECAPS, POLYGALCAPS, and TEXTCAPS, which determine the device driver's support for DDI primitives, are less important to applications for NT-based systems. On these systems, the graphics engine does a much better job of supporting rendering to a standard DIB-formatted frame buffer and breaking down GDI commands into primitives. These attributes are more for the graphics engine to understand about the printer driver. But if you do find problems with a particular driver, check these attributes to identify the problems. Windows 98/2000 adds SHADEBLEMDCAPS and COLORMGMCAPS indexes for checking a device's gradient fill, alpha blending, and color management capabilities.

You can also use GDI functions to enumerate the fonts supported by a printer device. High-end printers may support device fonts not available to display devices.

[Figure 17-5](#) shows the results generated by GetDeviceCaps for a printer device context, and the PRINTER_INFO_2 structure generated by GetPrinter. These results are generated by a revised version of the “Device” program originally written for [Chapter 5](#).

Figure 17-5. GetDeviceCaps for a printer device context.



Outline of a Print Job

With a printer device context having the correct settings, we are finally ready to start a print job using GDI. GDI provides several special functions to group the GDI commands into a logical sequence for a print job.

```
typedef struct {
    int cbSize;
    LPCTSTR IpszDocName;
    LPCTSTR IpszOutput;
    LPCTSTR IpszDataType;
    DWORD fwType;
} DOCINFO;

int StartDoc(HDC hDC, CONST DOCINFO * lpdi);
int StartPage(HDC hDC);
int EndPage(HDC hDC);
int EndDoc(HDC hDC);
HDC ResetDC(HDC hDC, const DEVMODE * lpInitData);
int AbortDoc(HDC hDC);
int SetAbortProc(HDC hdc, ABORTPROC IpAbortProc);
```

The function StartDoc tells GDI to start a new print job. The DOCINFO structure passed to StartDoc contains critical information about the job, which is needed by GDI and the spooler. Field IpszDocName is the document name, as it would appear in the printer spooler. But lots of applications embed the application name into it, for example, using the format "<appname> - <docname>." Field IpszOutput is the output device name, where data generated by the printer driver should be sent. If NULL is passed, the default port for the printer receives data. You can change this field to let the printer job be directed to a file, instead of a physical printer port. Field IpszDatatype is the application-suggested spool data type, which may be ignored by GDI and the device driver. For example, EMF spooling is critical for Windows 2000 for special features like N-up printing, so GDI may decide to use EMF spooling even if you asked for raw data spooling.

The last field contains rarely used flags as hints to GDI and the printer driver. DI_APPBANDING says the application is doing banding, while as we have seen, GDI is doing a good job of banding. DI_ROPS_READ_DESTINATION says the application is using a raster operation that reads from a destination surface. For a raster-based printer driver, which does full rendering on the host, there is usually no problem to support any raster operation. But PostScript printers may have difficulty in supporting raster operations that need to read from a destination surface. The DI_ROPS_READ_DESTINATION flag was seen effectively disabling EMF spooling on Windows 95.

The function StartPage signals the starting of a new page in a print job; the function End Page finishes a page. It's essential to divide the GDI drawing in different pages, due to the way the spooler and the printer work. No drawing calls should be put before StartPage, or between an EndPage and another StartPage. Note that with advanced document-handling features implemented in the print processor and the printer driver, StartPage and EndPage define only logical pages, which can be reordered, mapped many-to-one, or one-to-many to physical pages.

Normally, a page can be printed only after EndPage is called. This is due to the nature of spooling and to the fact that GDI commands can draw anywhere on the page, so the printer driver has to see all the drawing commands on a page before it knows what to do first. Spooling and document-handling features can affect this fact. For example,

there is a spooler option to start printing after the last page is spooled. Reverse-order printing, booklet printing, or duplex printing needs to wait for the last page. N-up printing needs to wait for multiple logical pages to be all spooled.

The function EndDoc concludes a print job started with StartDoc. In some cases, the actual printing happens after StartDoc is called, as explained above.

Recall that a printer device context is normally created with a DEVMODE structure with all the settings. These settings can be changed from page to page, using the ResetDC function. ResetDC accepts a device context handle and a pointer to a new, possibly modified DEVMODE structure. An application can use it to switch paper size, paper orientation, or other settings. For example, Microsoft Word allows each page to have a different paper size, orientation, etc.

The function AbortDoc can be used to abnormally terminate a print job, and cancel the not-yet-performed part of the current print job.

Function SetAbortProc sets up a callback routine, which will be used by GDI periodically to check if the print job should be aborted. It's normally used to implement a canceling-printing feature in applications.

[Listing 17-2](#) shows a simple, yet self-contained sample to illustrate the outline of a GDI print job.

Listing 17-2 Outline of a Print Job using GDI

```
int nCall_AbortProc;

BOOL CALLBACK SimpleAbortProc(HDC hDC, int iError)
{
    nCall_AbortProc++;
    return TRUE;
}

void SimplePrint(int nPages)
{
    TCHAR temp[MAX_PATH];

    DWORD size = MAX_PATH;
    GetDefaultPrinter(temp, &size); // default printer name

    HDC hDC = CreateDC(NULL, temp, NULL, NULL); // DC with default setting

    if ( hDC )
    {
        nCall_AbortProc = 0;
        SetAbortProc(hDC, SimpleAbortProc);

        DOCINFO docinfo;
        docinfo.cbSize      = sizeof(docinfo);
        docinfo.lpszDocName = _T("SimplePrint");
        docinfo.lpszOutput  = NULL;
```

```
docinfo.lpszDatatype = _T("EMF");
docinfo.fwType      = 0;

if ( StartDoc(hDC, & docinfo) > 0 )
{
    for (int p=0; p<nPages; p++) // one page at a time
        if ( StartPage(hDC) <= 0 )
            break;
        else
        {
            int width = GetDeviceCaps(hDC, HORZRES);
            int height = GetDeviceCaps(hDC, VERTRES);
            int dpix  = GetDeviceCaps(hDC, LOGPIXELSX);
            int dpiy  = GetDeviceCaps(hDC, LOGPIXELSY);

            wsprintf(temp, _T("Page %d of %d"), p+1, nPages);
            SetTextAlign(hDC, TA_TOP | TA_RIGHT );
            TextOut(hDC, width, 0, temp, _tcslen(temp));
            Rectangle(hDC, 0, 0, dpix, dpiy);
            Rectangle(hDC, width, height, width-dpix, height-dpiy);

            if ( EndPage(hDC)<0 )
                break;
        }
    EndDoc(hDC);
}

DeleteDC(hDC);
}

wsprintf(temp, "AbortProc called %d times", nCall_AbortProc);
MessageBox(NULL, temp, "SimlePrint", MB_OK);
}
```

The SimplePrint routine implements a simple multipage printing loop. It starts by querying for the default printer name, creating the printer device context, setting the abort procedure, and then starting printing using StartDoc. If everything is still under control, SimplePrint goes through a loop to print each page. The GDI drawing for each page is enclosed by StartPage and EndPage calls. For each page, the routine queries printable area size and resolution, draws a one-inch square in the top-left and bottom-right corner of the page, and prints a page number on the top-right corner of the page.

Note that for a printer device context, point (0, 0) in the device coordinate system, which is the same as point (0, 0) in the logical coordinate system if the mapping mode is MM_TEXT, is aligned with the first printable pixel location on the paper, not the very first pixel location on the paper. So its distance from the actual top-left corner of the paper is determined by a non-printable margin, as returned by GetDeviceCaps (hDC, PHYSICALOFFSETX) and GetDeviceCaps(hDC, PHYSICALOFFSETY). In other words, the exact location of the output from [Listing 17-2](#) is device dependent, and so is the font used for the text drawing, as it does not select a custom logical font.

You can use the output generated by SimplePrint to measure the actual nonprintable areas, and verify whether an

inch is really an inch. If multiple pages are printed, the actual page order also reflects other print settings like reverse printing.

SimplePrint ends by displaying a dialog box showing the number of times the abort procedure is called. You may be surprised to know that sometimes it's never called, and sometimes it's called only once per page. The abort procedure is of less importance for newer versions of the operating systems because of EMF spooling, and for Win32 applications because of better multitasking and multithreading support.

[< BACK](#) [NEXT >](#)

17.3 DESIGN FOR PRINTING

Using the spooler and GDI printing functions described in the last two sections, you will be able to find the printer, set up the printer, start and finish a print job. Anything that gets drawn on the paper really depends on your skillful use of the basic GDI drawing primitives. So it sounds like we should wrap up this chapter and move on.

But, in real-world applications, printing in Windows is hard because there is very little discussion on how printing features should be implemented beyond the basics. The best source of information may be MFC's printing functionality implemented as part of its document view architecture.

There are several classes of basic problems you may face when adding printing to your program. They may affect the way the program is designed in the first place. In this section, we will develop several classes for adding generic printing functionality to your program, which support a uniform logical coordinate system, display-ratio change, on-screen paper simulation, paper margin, multiple-page document, multiple-column display, and multipage printing. The next two sections will continue with more complete sample programs, which can do syntax-highlighting programs listing printing and photo printing.

Uniform Logical Coordinate Space

The programs we have written so far use the MM_TEXT mapping mode, which uses almost an identity transformation from the logical coordinate system to the device coordinate system, except for some possible translation. Drawing routines using MM_TEXT can't be shared easily for both screen display and printing, as printing uses high resolution, which is device dependent and even printing-setting dependent.

A better solution is to set up a logical coordinate system that is expressed in real-world units like inches or millimeters. GDI has provided several predefined mapping modes—for example, MM_LOENGLISH, MM_LOMETRIC, or MM_TWIPS. Professional applications also allow the user to change the display zoom ratio. For example, Microsoft Word allows you to change the zoom ratio from 500% to 10% relative to screen logical resolution, plus page width, full page, and dual page fit. Print preview implementation also requires your drawing routine to be able to scale to fit into a window. So you can't used the predefined mapping modes.

The only way out is defining your own mapping mode using the most generic MM_ANIOTROPIC mapping mode. Here are a few requirements for such a mapping mode.

- Uniform logical coordinate system. The number of units to represent a physical measurement unit in a logical coordinate system should be fixed. For example, one inch is always 300 units in a logical coordinate system, independent of a display scale and device. This will ensure you have a single piece of drawing code with no reference to GetDeviceCaps(hDC, LOGPIXELSX).
- Support display zoom between 500% and 10%.
- Support commonly used media size, even when running on Windows 95/98. To be more specific, the magic cut-off dimension is 17 inches, or 43 cm, which supports up to Tabloid, Ledger, 11 by 17 inch, or A3.
- Support common display logical resolutions without any rounding error.

With these restrictions, it's easy to figure out what you can do. The commonly used display logical resolutions are 96 dpi for small font, 120 dpi for large font, 360 dpi and 600 dpi for printers. The lowest common multiplier (LCM) of 96, 120, 360, and 600 is 7200. Recall that the LCM of a bunch of numbers is the smallest number that is a multiplier of all the numbers under consideration. Multiplying 7200 by 17-inch gives 122,400, which is too far away from the maximum device surface size (32,767) you can use on Windows 95/98. We can only use a number smaller than 1927 (32,767/17). LCM of 96 and 120 is 480, which is within the limit. LCM of 96, 120, and 360 is 1440, which is within the limit, too. LCM of 96, 120, and 300 is 2400, which is beyond the limit. So a reasonable number to choose is 1440 dpi, the same number used by the MM_TWIPS mapping mode.

The highest display resolution is 120 dpi in the large-font mode. Multiplying 120 dpi by 500% gives 600 dpi, which is only one-third of 1927. To conclude, if we use 1440 dpi as the uniform logical coordinate space resolution, we can cover an area of 22.75 by 22.75 inches in the logical coordinate space, which can be zoomed 1500% on a 120-dpi display, without breaking the 16-bit GDI limitation on Windows 95/98.

What's so cool about 1440 dpi is that an application can precisely address any pixels in the device coordinate space for 96-, 120-, and 360-dpi graphics devices without any rounding error. For example, 15 units in the logical space equals one pixel on 96-dpi display, and 12 units in the logical space equals one pixel on 120-dpi display. For a 600-dpi printer, one pixel in the printer surface is mapped to 2.4 logical units, or 12 logical units represent 5 pixels. Another advantage of 1440 dpi is that 1440 is a multiple of 72, so one point used in font metrics is 20 units.

If you're lucky enough to write applications solely for NT-based systems, you should consider using 7200-dpi logical coordinate space, which can precisely address every pixel in the commonly seen 96-, 120-, 300-, 360-, 600-, 720-, 1200-, 1440-, and 2400-dpi graphics devices.

Setting up this uniform logical coordinate space is rather easy, as shown by the following SetupULCS routine.

```
#ifdef NT_ONLY
#define BASE_DPI 9600
#else
#define BASE_DPI 1440
#endif

int gcd(int m, int n)
{
    if ( m==0 )
        return n;
    else
        return gcd(n % m, m);
}

void SetupULCS(HDC hDC, int zoom)
{
    SetMapMode(hDC, MM_ANISOTROPIC);

    int mul = BASE_DPI          * 100;
    int div = GetDeviceCaps(hDC, LOGPIXELSX) * zoom;
    int fac = gcd(mul, div);
```

```
mul /= fac;  
div /= fac;  
  
    SetWindowExtEx(hDC, mul, mul, NULL);  
    SetViewportExtEx(hDC, div, div, NULL);  
}
```

Macro BASE_DPI is the number of units in the uniform logical coordinate system that corresponds to one inch. It's defined as 1440 normally, unless you tell the compiler the program is meant only for NT-based platforms by defining the NT_ONLY macro.

Routine SetupULCS accepts a device context handle and a display zoom ratio. The device context handle can refer to any graphics device. The display zoom ratio is 400 for 400% zoom, and 10 for 10% zoom. Two internal variables are calculated based on the display zoom ratio and the device context's logical resolution, and then scaled down by removing their greatest common divisor (GCD). SetWindowExtEx and Set ViewportExtEx are then called to set up the mapping from the logical coordinate space to the device coordinate space. Note that BASE_DPI is used as both horizontal and vertical extent in calling SetWindowExtEx, to ensure that the logical coordinate space is always BASE_DPI dpi. That is to say, moving BASED_DPI units in the logical coordinate space is guaranteed to be one inch.

Table 17-1. Supporting Multiple Devices and Zoom Ratio

Display Resolution	Device Zoom Ratio	Window Extent	Viewport Extent
96-dpi display	500%	(3, 3)	(1, 1)
96-dpi display	100%	(15, 15)	(1, 1)
96-dpi display	10%	(150, 150)	(1, 1)
120-dpi display	50%	(24, 24)	(1, 1)
120-dpi display	10%	(120, 120)	(1, 1)
360-dpi printer	100%	(4, 4)	(1, 1)
600-dpi printer	100%	(12, 12)	(5, 5)
1200-dpi printer	100%	(6, 6)	(5, 5)

[Table 17-1](#) shows sample logical-coordinate-space to device-coordinate-space mapping set up by the SetupULCS routine.

Paper Simulation

Simulating paper dimensions on the screen is a common technique used by professional graphics applications and word processors to give the user a real sense of how the document is going to be printed on paper. Such a display mode is now called a page layout view. It is also widely used in a print preview display. Some applications even use page layout view as the primary user interface.

There are several elements in implementing a page layout on the screen. First, the client area of a window is normally painted with a dark color to form a background. Pages are displayed using white as the foreground color, a black border, with a simple drop shadow. There are margins and gaps between pages, and between pages and the client area boundary. The print preview normally displays a dotted-line frame to mark nonprintable areas, so that any

overflow is visible on the screen. Some applications also provide some visual clue on the paper margin set up by the user through the page setup dialog box.

Here is our implementation for paper simulation on the screen. The DrawPaper routine is a method of the KSurface class we're able to discuss. The m_Paper member in KSurface class specifies paper dimensions, m_MinMargin is for minimum margin, and m.Margin is for margin. These variables are obtained from the page setup dialog box. The DrawPaper routine calls a help function DrawFrame three times, first to draw paper frame with a shadow, second to mark the minimum margin, and finally to mark the margin. The px and py functions are for coordinate scaling.

```
void KSurface::DrawPaper(HDC hDC, const RECT * rcPaint, int col, int row)
{
    // paper frame
    DrawFrame(hDC, px(0, col), py(0, row),
              px(m_Paper.cx, col), py(m_Paper.cy, row),
              RGB(0, 0, 0), RGB(0xE0, 0xE0, 0xE0), true);

    // minimum margin : printable margin
    DrawFrame(hDC, px(m_MinMargin.left, col), py(m_MinMargin.top, row),
              px(m_Paper.cx - m_MinMargin.right, col),
              py(m_Paper.cy - m_MinMargin.bottom, row),
              RGB(0xF0, 0xF0, 0xF0), RGB(0xF0, 0xF0, 0xF0), false);

    // margin
    DrawFrame(hDC, px(m.Margin.left, col), py(m.Margin.top, row),
              px(m_Paper.cx - m.Margin.right, col),
              py(m_Paper.cy - m.Margin.bottom, row),
              RGB(0xFF, 0xFF, 0xFF), RGB(0xFF, 0xFF, 0xFF), false);
}
```

Multipage, Multicolumn Display

A generic document may be divided into multiple pages. When the display zoom ratio is small enough, multiple pages may be fitted into a single row to make the screen more readable and give the user a sense of the whole document.

For an application supporting multipage document and multicolumn display, the logic for displaying multiple pages in multiple columns should be put in a central place, instead of being implemented over and over again. Here is the UponDraw routine that controls display for the KSurface class:

```
// multipage display in multicolumn, with paper simulation, zoom, scroll
void KSurface::UponDraw(HDC hDC, const RECT * rcPaint)
{
    int nPage = GetPageCount();
    int nCol = GetColumnCount();
    int nRow = (nPage + nCol - 1) / nCol;
```

```
for (int p=0; p<nPage; p++)
{
    SaveDC(hDC);

    int col = p % nCol;
    int row = p / nCol;
    DrawPaper(hDC, rcPaint, col, row);
    SetupULCS(hDC, m_nZoom);

    OffsetViewportOrgEx(hDC, px(m_Margin.left, col),
        py(m_Margin.top, row), NULL);

    UponDrawPage(hDC, rcPaint,
        GetDrawableWidth(), GetDrawableHeight(), p);

    RestoreDC(hDC, -1);
}
}
```

The UponDraw method, named to avoid confusion with OnDraw, uses the virtual function GetPageCount to figure out the total number of pages for the document, and Get ColumnCount to figure out the number of columns. It then goes through a loop to display each page at different rows and columns. For each page, it saves the device context setting, calls the paper simulation routine, and sets up the uniform logical coordinate space. Before calling the UponDrawPage virtual method to draw the contents of each page, it calls OffsetViewportOrgEx to move the origin in the logical coordinate space to the position determined by the paper margin for the current page. UponDrawPage is passed the width and height of the effective drawable area (without the margin removed), and the sequence page number. So it's freed from worrying about the page's position on the screen. After a page is drawn, the device context setting is restored to handle possible other pages.

Multipage Printing

We have already discussed multipage printing in the last section. The challenge we are facing here is how to reuse the same UponDrawPage page drawing virtual method in printing. This problem is basically how to set up the proper logical coordinate space for printing.

Here is the printing implementation for the KSurface class.

```
// multipage printing, 100% scale
bool KSurface::UponPrint(HDC hDC, const TCHAR * pDocName)
{
    int scale = GetDeviceCaps(hDC, LOGPIXELSX);

    SetMapMode(hDC, MM_ANISOTROPIC);

    SetWindowExtEx(hDC, BASE_DPI, BASE_DPI, NULL);
    SetViewportExtEx(hDC, scale, scale, NULL);
    OffsetViewportOrgEx(hDC,
```

```
(m_Margin.left - m_MinMargin.left) * scale / BASE_DPI,
(m_Margin.top - m_MinMargin.top ) * scale / BASE_DPI, NULL);

DOCINFO docinfo;

docinfo.cbSize      = sizeof(docinfo);
docinfo.lpszDocName = pDocName;
docinfo.lpszOutput  = NULL;
docinfo.lpszDatatype = "EMF";
docinfo.fwType      = 0;

if ( StartDoc(hDC, & docinfo) <= 0)
    return false;

int nFrom = 0;
int nTo   = GetPageCount();

for (int pageno=nFrom; pageno<nTo; pageno++)
{
    if ( StartPage(hDC) < 0 )
    {
        AbortDoc(hDC);
        return FALSE;
    }

    UponDrawPage(hDC, NULL, GetDrawableWidth(),
        GetDrawableHeight(), pageno);
    EndPage(hDC);
}

EndDoc(hDC);

return true;
}
```

Setting up the logical coordinate space is easier for printing, because it's always printed at 100% zoom ratio. There is no paper simulation for printing, but to use the same UponDrawPage routine we have to move the logical coordinate space origin to the position specified by the top-left margin. The OffsetViewportOrgExt function is called for that purpose. Note, the amount of offset is only the difference between the margin and the nonprintable area, because the origin in the device coordinate space is aligned at the first printable location.

Generic Printing Class

Now it's time to introduce a generic class for GDI printing, the KSurface class. [Listing 17-3](#) shows its class declaration and rest of its implementation.

Listing 17-3 KSurface class for generic multipage display and printing.

```
// 1440 = LCM(72, 96, 120, 360)      good for 22.75 inch on Win95/98
// 7200 = LCM(72, 96, 120, 360, 600) perfect for all major devices
#ifndef NT_ONLY
typedef enum { ONEINCH = 7200 };
#else
typedef enum { ONEINCH = 1440 };
#endif

class KSurface
{
public:
    typedef enum { BASE_DPI = ONEINCH,
        MARGIN_X = 16,
        MARGIN_Y = 16
    };

    KOutputSetup m_OutputSetup;
    int m_nSurfaceWidth; // total surface width in pixel
    int m_nSurfaceHeight; // total surface height in pixel
    SIZE m_Paper; // in BASE_DPI
    RECT m_Margin; // in BASE_DPI
    RECT m_MinMargin; // in BASE_DPI
    int m_nZoom; // 100 for 1:1
    int m_nScale; // GetDeviceCaps(hDC, LOGPIXELSX) * zoom / 100

    int px(int x, int col) // from base_dpi to screen
    {
        return (x + m_Paper.cx * col) * m_nScale / BASE_DPI +
            MARGIN_X * (col+1);
    }

    int py(int y, int row) // from base_dpi to screen
    {
        return (y + m_Paper.cy * row) * m_nScale / BASE_DPI +
            MARGIN_Y * (row+1);
    }

public:
    int GetDrawableWidth(void) const
    {
        return m_Paper.cx - m_Margin.left - m_Margin.right;
    }

    int GetDrawableHeight(void) const
    {
        return m_Paper.cy - m_Margin.top - m_Margin.bottom;
    }
```

```
}

virtual int GetColumnCount(void)
{
    return 1;
}

virtual int GetPageCount(void)
{
    return 1; // single page
}

virtual const TCHAR * GetDocumentName(void)
{
    return _T("KSurface - Document");
}

virtual void DrawPaper(HDC hDC, const RECT * rcPaint,
                      int col, int row);
virtual void CalculateSize(void);
virtual void SetPaper(void);
virtual void RefreshPaper(void);

virtual void UponDrawPage(HDC hDC, const RECT * rcPaint, int width,
                        int height, int pageno=1);
virtual bool UponSetZoom(int zoom);
virtual void UponInitialize(HWND hWnd);
virtual void UponDraw(HDC hDC, const RECT * rcPaint);
virtual bool UponPrint(HDC hDC, const TCHAR * pDocName);
virtual bool UponFilePrint(void);
virtual bool UponFilePageSetup(void);
};

// convert from <fr>1/1000 inch page setup to BASE_DPI
void KSurface::SetPaper(void)
{
    POINT paper;
    RECT margin, margin;
    m_OutputSetup.GetPaperSize(paper);
    m_OutputSetup.GetMargin(margin);
    m_OutputSetup.GetMinMargin(minmargin);

    m_Paper.cx = paper.x * BASE_DPI / 1000;
    m_Paper.cy = paper.y * BASE_DPI / 1000;

    m_Margin.left = margin.left * BASE_DPI / 1000;
    m_Margin.right = margin.right * BASE_DPI / 1000;
    m_Margin.top = margin.top * BASE_DPI / 1000;
    m_Margin.bottom = margin.bottom * BASE_DPI / 1000;
```

```
m_MinMargin.left = minmargin.left * BASE_DPI / 1000;
m_MinMargin.right = minmargin.right * BASE_DPI / 1000;
m_MinMargin.top = minmargin.top * BASE_DPI / 1000;
m_MinMargin.bottom= minmargin.bottom * BASE_DPI / 1000;
}

// calculate overall surface size needed to fit nPage in nCol columns
void KSurface::CalculateSize(void)
{
    int nPage = GetPageCount();
    int nCol = GetColumnCount();
    int nRow = (nPage + nCol - 1) / nCol;

    m_nSurfaceWidth = px(m_Paper.cx, 0) * nCol + MARGIN_X;
    m_nSurfaceHeight = py(m_Paper.cy, 0) * nRow + MARGIN_Y;
}

bool KSurface::UponSetZoom(int zoom)
{
    if (zoom == m_nZoom)
        return false;

    m_nZoom = zoom;
    HDC hDC = GetDC(NULL);
    m_nScale = zoom * GetDeviceCaps(hDC, LOGPIXELSX) / 100;
    DeleteDC(hDC);
    CalculateSize();
    return true;
}

void KSurface::RefreshPaper(void)
{
    int zoom = m_nZoom;
    m_nZoom = 0;
    SetPaper();
    UponSetZoom(zoom);
}

void KSurface::UponInitialize(HWND hWnd)
{
    m_OutputSetup.SetDefault(hWnd, 1, GetPageCount());
    m_nZoom = 100;
    RefreshPaper();
}

void KSurface::UponDrawPage(HDC hDC, const RECT * rcPaint, int width,
                           int height, int pageno)
{
    for (int h=0; h<=(height-BASE_DPI); h += BASE_DPI)
```

```
for (int w=0; w<=(width-BASE_DPI); w += BASE_DPI)
    Rectangle(hDC, w, h, w+BASE_DPI, h+BASE_DPI);
}

bool KSurface::UponFilePrint(void)
{
    int rslt = m_OutputSetup.PrintDialog(PD_RETURNDC | PD_SELECTION);

    if ( rslt==IDOK )
        UponPrint(m_OutputSetup.GetPrinterDC(), GetDocumentName());

    m_OutputSetup.DeletePrinterDC();
    return false;
}

bool KSurface::UponFilePageSetup(void)
{
    if ( m_OutputSetup.PageSetup(PSD_MARGINS) )
    {
        RefreshPaper();
        return true;
    }
    return false;
}
```

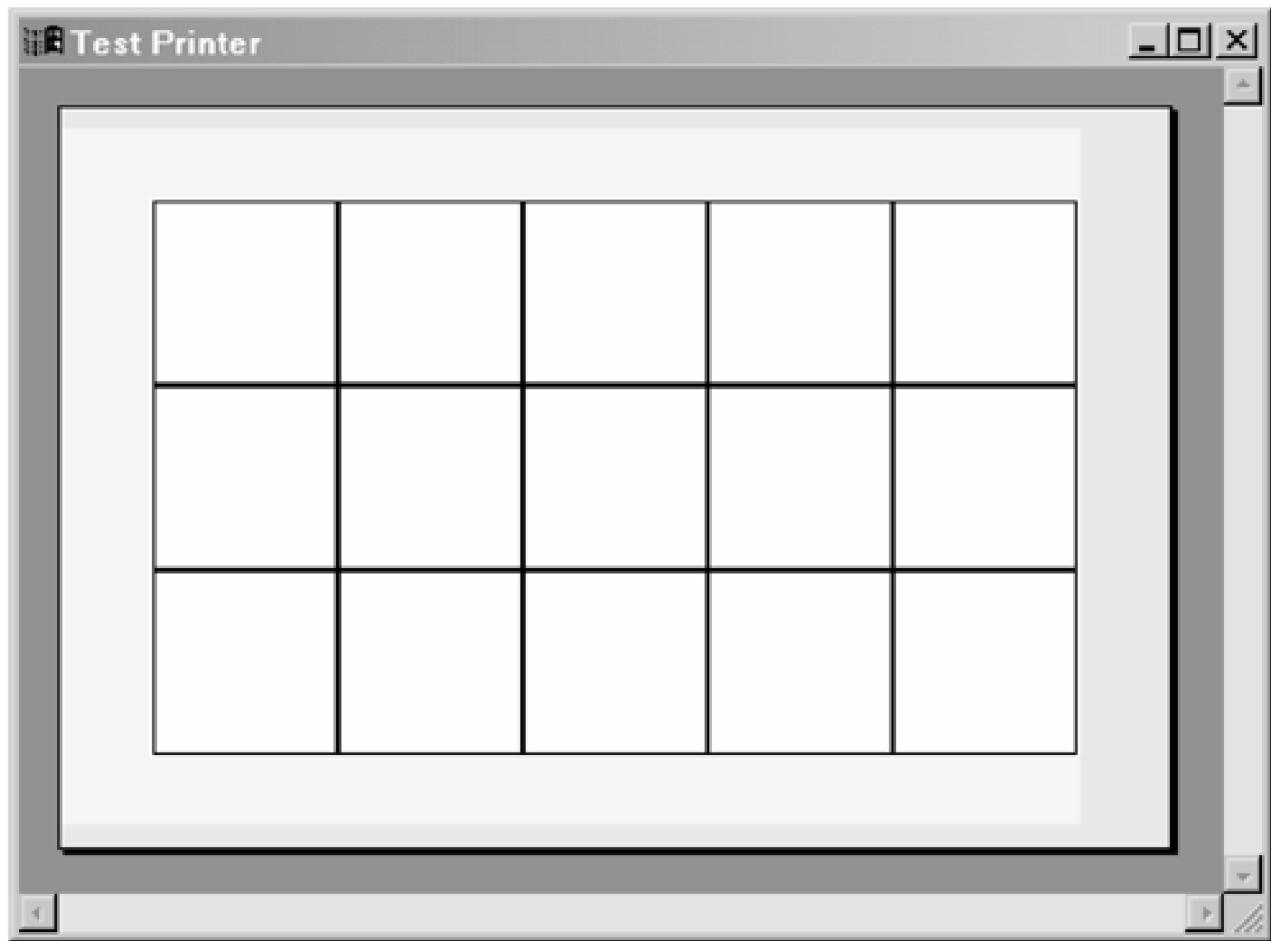
The KSurface class is a generic class for multipage displaying and printing, which supports different display zoom ratios. It's so generic that it is not even attached to any window, you just need to pass a device context handle to use it. So it can be used in an SDI window; MDI window, dialog box, property sheet, even ActiveX control, an EMF, or a memory device context.

The m_OutputSetup member variable is an instance of the KOutputSetup class, which manages print setup and page setup. Member variables m_nSurfaceWidth and m_nSurfaceHeight specify total overall pixel height and width, which can be used to scroll the display in a limited-size window. The drawing code fully supports scrolling. The next three member variables are taken from the page setup dialog box, but converted to the uniform logical coordinate space by the SetPaper method. The m_nZoom and m_nScale member variables are for managing the display zoom ratio, which is used by the px and py in-line conversion functions.

The virtual and nonvirtual methods are quite obvious in what they do. Upon DrawPage is the key method to be overridden for customizing drawing. Its default implement fills one-inch square blocks on the page. The UponSetZoom method should be linked with the menu or toolbar to control the display zoom ratio. The UponInitialize method should be called to initialize class member variables. The UponDraw method should be linked with WM_PAINT message handling, if it's used in a window. The UponFilePrint method should be linked with the print menu item. And finally, the Upon FilePageSetup method should be linked with the page setup menu item.

On the CD accompanying this book there is a KPageCanvas class, which derives from the KScrollCanvas class for scrollable MDI child window, and from the KSurface class for multipage drawing and printing. [Figure 17-6](#) shows KSurface's default Upon Draw Page routine drawing one-inch blocks on a 4-inch by 6-inch index card in landscape orientation, with margins all set to be $\frac{1}{2}$ inch, and displayed at 75% zoom ratio.

Figure 17-6. Sample usage of KSurface and KPageCanvas class.



[< BACK](#) [NEXT >](#)

17.4 DRAWING ON PRINTER DEVICE CONTEXT

GDI is supposed to be device independent, so drawing to a screen device context and to a printer device context should not differ greatly. But there are some issues to consider when dealing with a printer device context, especially when the printouts are not what you want. Some of these problems are not really printer device context related, as they are related more to how to design your drawing algorithm to be totally scalable with different logical coordinate system settings.

Coordinate Space Units

When dealing with screen displaying and printing in the same piece of code, you need to switch your mind-set from the MM_TEXT mode or the device coordinate space to the logical coordinate space. One unit in the logical coordinate space is not one unit in the device coordinate space any more.

For example, the KSurface class developed in the last section uses 1440-dpi logical coordinate space uniformly, both for printing and screen display at different zoom ratios. Most graphics devices do not have a resolution as high as 1440 dpi, so normally multiple logical units are mapped into a single pixel on the device surface. But in the near future, it's possible to have printers reporting 2400 dpi to GDI, which is higher than 1440 dpi. For such devices, a single logical space unit may correspond to several pixels on the device surface.

To be more specific, there are several things to pay attention to regarding coordinate space units, if you want to write generic device-independent drawing code.

- For non-single-pixel pens, the pen width is expressed in the logical coordinate space. When writing generic code, you have to use pen width that is fixed in real-world units, instead of pixel units. For example, when using the KSurface class, which uses 1440 as one inch, a one-point-wide pen should use 20 as the pen width.
- The GDI functions LPtoDP and DPtoLP should be used to map coordinates between the logical coordinate system and the device coordinate system, because you can't assume a fixed relationship between them.
- Some drawing code may increment a coordinate to move to the next coordinate to form a big drawing. For example, you may have a sequence of lines next to each other to fill an area. Such algorithms should be reconsidered for possible gaps, overlapping, and inconsistency.
- BitBlt is often used to display a bitmap, but it's good only for a screen display or an MM_TEXT mapping mode. For a generic drawing code, the more generic StretchBlt should replace BitBlt.
- The exact size of patterns used in GDI hatch brushes is device dependent. If you use hatch brushes in scalable drawing code, or for printing, there is no way to tell the final size of the patterns. Implement your own device-independent hatch brush solution, as we demonstrated in [Chapter 9](#).
- Pattern brushes use a bitmap in the device coordinate space without scaling. So when a bitmap pattern brush is drawn to a high-resolution printer device context, it gets repeated to fill an area without any scaling. So pattern brushes should be avoided unless you scale the pattern bitmap to the right size before creating the pattern brush. Bitmap tiling should be used as a replacement.

Text

Text metrics and drawing functions provided by GDI do not provide a good solution to implement truly scalable text drawing. The main problem is that GDI uses integer-based text metrics scaled down to font size. When dozens of characters are put into a single line, their errors in widths can add up to a large amount to influence text formatting. Text height also has errors that may accumulate rapidly.

If you use normal GDI functions like TextOut, or USER functions like DrawText, with the KSurface class, and KPageCanvas class, you can experiment with different device resolutions and display zoom scales and easily notice their error accumulation.

This issue has been explored in great detail in [Chapter 15](#) on text. The solution is to use a reference font whose size is the same as the em-square size a TrueType font is designed in. The class KTextFormtor was developed to solve this problem.

To experiment with device-independent text formatting and multipage printing, a sample program CodePrint is provided with this chapter. CodePrint implements a simple multipage source code viewer, with syntax highlighting and printing. Its text formatting uses the KTextFormator class to make sure the number of lines on a page and character position on a line are always fixed, regardless of the device resolution and display zoom ratio.

Syntax highlighting is implemented by a simple C/C++ lexical analyzer, which knows about C/C++ keyword, number, character literal, string literal, and comments. Based on the lexical analyzer, each character in a string is assigned a color. A sequence of characters with the same color is drawn as a single text drawing call with the right text foreground color.

The source-code drawing logic is implemented in the KProgramPageCanvas class, which is derived from the KPageCanvas class discussed in the last section. Here are its two important methods.

```
void KProgramPageCanvas::SyntaxHighlight(HDC hDC, int x, int y,
                                         const TCHAR * mess)
{
    BYTE flag[MAX_PATH];

    int len = _tcslen(mess);
    assert(len < MAX_PATH-50);
    memset(flag, 0, MAX_PATH);
    ColorText(mess, flag);

    float width = 0.0;

    for (int k=0; k<len; )
    {
        int next = 1;

        while ( (k+next<len) && (flag[k]==flag[k+next]) )
            next++;

        if (next>1)
        {
```

```
SetTextColor(hDC, crText[flag[k]]);

m_formatter.TextOut(hDC, (int)(x + width + 0.5), y, mess+k, next);

float w, h;
m_formatter.GetTextExtent(hDC, mess+k, next, w, h);
width += w;
k += next;
}

void KProgramPageCanvas::UponDrawPage(HDC hDC, const RECT * rcPaint,
    int width, int height, int pageno)
{
    if ( rcPaint ) // skip if current page does not intersect with rcPaint
    {
        RECT rect = { 0, 0, width, height };

        LPtodP(hDC, (POINT *) & rect, 2);

        if ( !IntersectRect(& rect, rcPaint, & rect) )
            return;
    }

    HGDIOBJ hOld = SelectObject(hDC, m_hFont);
    SetBkMode(hDC, TRANSPARENT);
    SetTextAlign(hDC, TA_LEFT | TA_TOP);

    KGetline parser(m_pBuffer, m_nSize);
    int skip = pageno * m_nLinePerPage;

    for (int i=0; i<skip; i++)
        parser.Nextline();

    for (i=0; i<m_nLinePerPage; i++)
        if ( parser.Nextline() )
    {
        SyntaxHighlight(hDC, 0,
            (int)(m_formatter.GetLINESPACE() * i + 0.5), parser.m_line);
    }
    else
        break;

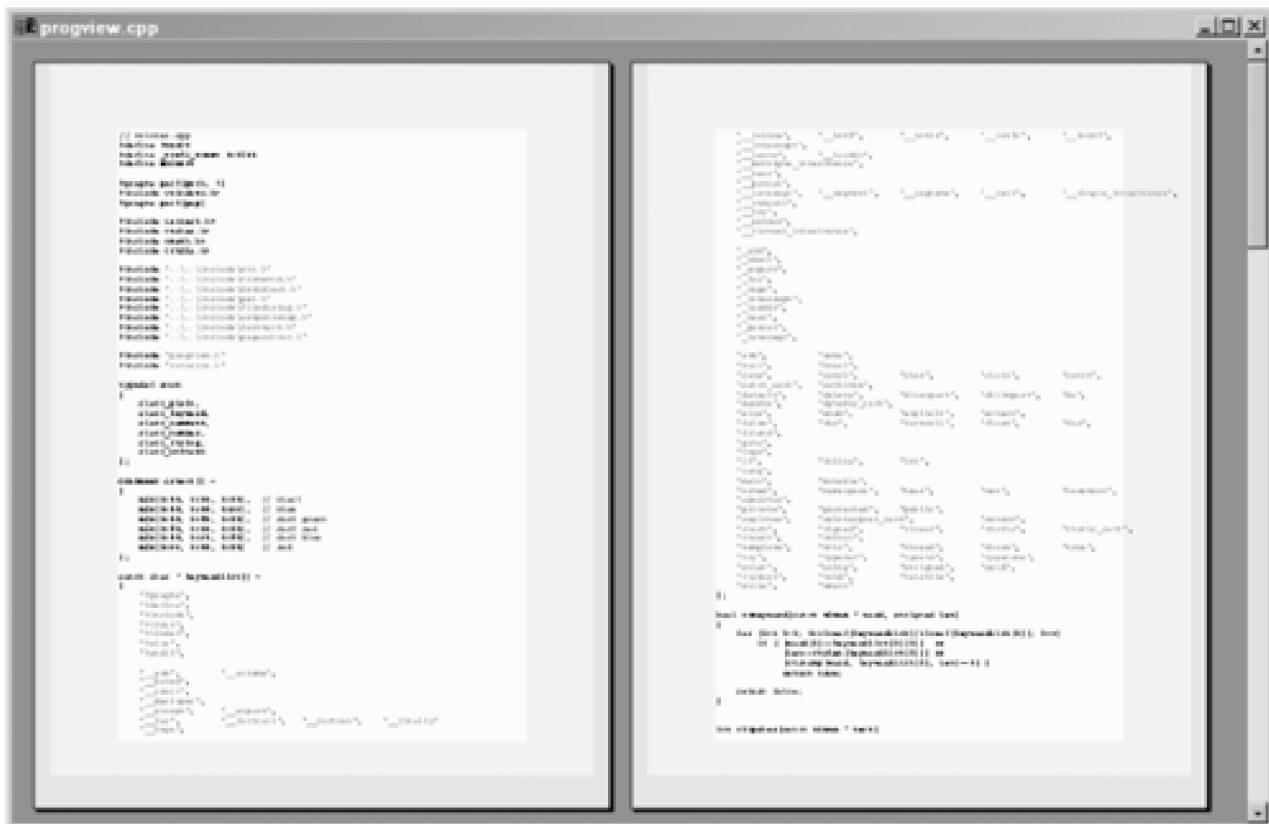
    SelectObject(hDC, hOld);
}
```

The KProgramPageCanvas class overrides the KSurface::GetPageCount method to provide an accurate page count, which is based on counting the number of lines in a source code and the precise text height. The UponDrawPage method handles drawing of a single page in a multipage, multicolumn setting. It starts by converting

a page rectangle from the logical coordinate space to a device coordinate, so that it can check whether it overlaps with the current repainting rectangle. Nonvisible pages will be totally skipped. It then uses the KGetline class to sequentially get text lines in the source code, skip the lines belonging to previous pages, and draw only the current pages. For better performance, we should have built an index. The SyntaxHighlight routine handles the drawing of a single line. It calls the lexical analyzer ColorText to assign color to each character based on C/C++ lexical rules, and uses methods from the KText Format to accurately position the text drawing.

Now we have a source-code printing program that prints in color. [Figure 17-7](#) shows a sample screen of the CodePrint program, displaying its own source code.

Figure 17-7. CodePrint: accurate text formatting, multipage viewing and printing.



Bitmap

Device-dependent bitmaps and DIB sections are always connected with a memory device context. If a memory device context is created based on the target device context, device-dependent bitmaps created to be compatible with the screen most likely will not be compatible with the printer device context. For example, if you use LoadBitmap to load a bitmap resource as DDB, and create a memory device context based on the printer device context, you can't normally select the bitmap into the memory device context, because they may not be compatible. You should create the memory device context based on a display device context instead.

As a general rule of thumb, device-dependent bitmaps should be avoided for printing, especially when running in 256-color display modes, as printers do not support the hardware palette.

Some applications divide big bitmaps into small chunks to optimize the bitmap display on screen, or to avoid the old

64-KB bitmap size limitation. This is especially important on Windows 95/98, as a GDI drawing call blocks other threads from using GDI until it's finished to solve the reentrance problem of 16-bit GDI code. But such a divide-and-conquer strategy may cause big problems for printing. First, we have seen that the EMF generation is not smart enough to remove the unused source bitmap portion from the EMF, so the EMF generated could be huge, causing a big performance problem. Second, sending lots of small bitmaps to the printer driver makes it harder for the printer driver to process them. Third, when dividing bitmaps into smaller bitmaps, you have to make sure there is no gap between them, even when they are scaled up or down.

For EMF-related bitmap printing problems, the EMF viewing and decoding tools developed in the last chapter could offer valuable help.

JPEG Image Printing

With the advancement of digital image devices like digital cameras, scanners, and color printers, we have more and more good-quality image sources, and professional photo grade printing output. Printing high-quality photos on your personal computers has never been so much fun. The problem with Win32 bitmap formats is that there is no compression, not to mention good-quality compression, for high color and true color images. Most photos you can get are not in the Windows native BMP format.

The most common photo-image format used today is JPEG, which is developed by the Joint Photographic Experts Group. Up to now, GDI still does not support JPEG handling, although it provides a limited back door for sending BMP-wrapped JPEG to printer drivers. On Windows 98/2000, to send a JPEG or PNG image to a printer driver, an application should call ExtEscape with QUERYESCSUPPORT and CHECKJPEGFORMAT as parameters. If the printer driver gives the green light, an application can send a JPEG or PNG image to the printer driver using SetDIBitsToDevice or StretchDIBits. But there is no guarantee that the printer you're using supports JPEG/PNG decompression, so the application has to support JPEG/PNG decompression anyway. If by any chance a printer driver supports it, printing performance will be improved.

The following routine shows how to send a JPEG image to a device.

```
BOOL StretchJPEG(HDC hDC, int x, int y, int w, int h,  
    void * pJPEGImage, unsigned nJPEGSize, int width, int height)  
{  
    DWORD esc = CHECKJPEGFORMAT;  
    if ( ExtEscape(hDC,QUERYESCSUPPORT,sizeof(esc),(char *)&esc,0,0) <=0 )  
        return FALSE;  
  
    DWORD rslt = 0;  
    if ( ExtEscape(hDC, CHECKJPEGFORMAT, nJPEGSize, (char *) pJPEGImage,  
        sizeof(rslt), (char *) &rslt) <=0 )  
        return FALSE;  
    if ( rslt!=1 )  
        return FALSE;  
  
    BITMAPINFO bmi;  
  
    memset(&bmi, 0, sizeof(bmi));  
    bmi.bmiHeader.biSize      = sizeof(BITMAPINFOHEADER);
```

```
bmi.bmiHeader.biWidth      = width;
bmi.bmiHeader.biHeight     = - height; // top-down image
bmi.bmiHeader.biPlanes     = 1;
bmi.bmiHeader.biBitCount   = 0;
bmi.bmiHeader.biCompression = BI_JPEG;
bmi.bmiHeader.biSizeImage  = nJPEGSize;

return GDI_ERROR != StretchDIBits(hDC, x, y, w, h,
    0, 0, width, height, pJPEGImage,
    & bmi, DIB_RGB_COLORS, SRCCOPY);
}
```

The StretchJPEG routine calls the ExtEscape function twice, first to check if CHECKJPEGFORMAT escape is supported, second to check if the JPEG image we want to send is acceptable. If both checks pass, the JPEG image is wrapped as a DIB and passed to StretchDIBits. Note that, if StretchJPEG fails, the caller is responsible for decoding JPEG and sending the decoded JPEG to the device instead.

Decoding JPEG itself is very hard, due to the complexity of JPEG compression. But the Independent JPEG Group (IJG) has done a very good job of implementing JPEG compression and decompression for multiple platforms, and it has made the source code free. The Independent JPEG Group's JPEG library can be downloaded from www.ijg.org. Although IJG's JPEG library is written in plain C, not C++, it has a very good object-oriented design, which implements information hiding and inheritance through clever use of C language features.

On the CD accompanying this book, IJG's JPEG library has been morphed a little bit to be more like C++ code. Moving to C++ makes it easier to customize the library without using C function pointers. For example, the original IJG code supports decoding only from the file stream. Now it's very easy to extend it to support decoding from a memory buffer.

```
class const_source_mgr : public jpeg_source_mgr
{
public :
    void Reset(const unsigned char * buffer, int len )
    {
        bytes_in_buffer = len;
        next_input_byte = buffer;
    }

    void init_source(j_decompress_ptr cinfo)
    {

    }

    virtual void term_source(j_decompress_ptr cinfo)
    {
        if (cinfo->src)
        {
            delete (const_source_mgr *) cinfo->src;
            cinfo->src = NULL;
        }
    }
}
```

```
}

};

GLOBAL(void) jpeg_const_src (j_decompress_ptr cinfo,
    const unsigned char * buffer, int len)
{
    const_source_mgr * src;

    if (cinfo->src == NULL) // first time for this JPEG object?
        cinfo->src = new const_source_mgr;

    src = (const_source_mgr *) cinfo->src;
    src->Reset(buffer, len);
}
```

[Listing 17-4](#) shows a class for decoding a JPEG image to a Windows DIB format, and generating a JPEG file from a DIB image.

Listing 17-4 Class KPicture for JPEG image encoding/decoding

```
class KPicture
{
    void Release(void);
    int Allocate(int width, int height, int channels, bool bBits=true);

public:
    BITMAPINFO * m_pBMI;
    BYTE     * m_pBits;
    BYTE     * m_pJPEG;
    int      m_nJPEGSize;

    KPicture();
    ~KPicture();

    int GetWidth(void) const
    {
        return m_pBMI->bmiHeader.biWidth;
    }

    int GetHeight(void) const
    {
        return m_pBMI->bmiHeader.biHeight;
    }

    BOOL DecodeJPEG(const void * jpegimage, int jpegsize);
    BOOL QueryJPEG(const void * jpegimage, int jpegsize);

    BOOL LoadJPEGFile(const TCHAR * filename);
```

```
BOOL SaveJPEGFile(const TCHAR * fileName, int quality);
};

BOOL KPicture::DecodeJPEG(const void * jpegimage, int jpegsize)
{
try
{
struct jpeg_decompress_struct dinfo;

jpeg_error_mgr jerr;
dinfo.err = & jerr;
dinfo.jpeg_create_decompress();
jpeg_const_src(&dinfo, (const BYTE *) jpegimage, jpegsize);
dinfo.jpeg_read_header(TRUE);
dinfo.jpeg_start_decompress();

int bps = Allocate(dinfo.image_width, dinfo.image_height,
dinfo.out_color_components, true);

if ( m_pBits==NULL )
return FALSE;

for (int h=dinfo.image_height-1; h>=0; h--) // bottom-up
{
BYTE * addr = m_pBits + bps * h;
dinfo.jpeg_read_scanlines(&addr, 1);
}

dinfo.jpeg_finish_decompress();
dinfo.jpeg_destroy_decompress();

m_pJPEG = (BYTE *) jpegimage;
m_nJPEGSize = jpegsize;
}
catch (...)
{
return FALSE;
}

return TRUE;
}
```

What's shown in [Listing 17-4](#) is KPicture class's declaration and its decoding routine DecodeJPEG. Method DecodeJPEG decodes a memory buffer containing a JPEG image straight into a Windows bottom-up DIB format. The Allocate method manages memory allocation and setting up a BITMAPINFO structure. Both a 24-bit color JPEG image and an 8-bit gray scale image are supported. After an image is decoded, the pointer and the size of the original JPEG image are still kept in the KPicture class, just in case a printer driver is welling to accept it. Note that we have changed the JPEG library to return a decoded image in blue, green, red byte order to be compatible with the 24-bpp DIB format.

To experiment with JPEG image decoding, displaying, and printing, there is another program on the CD named "ImagePrint." In ImagePrint, the JPEG image is supported by the KImageCanvas class, which derives from the KPageCanvas class. Kimage Canvas's main drawing routine supports displaying and printing a decoded image, and printing the original JPEG image to a printer. Here is its UponDrawPage method.

```
void KImageCanvas::UponDrawPage(HDC hDC, const RECT * rcPaint,
                                int width, int height, int pageno)
{
    if ( (m_pPicture==NULL) && (m_pPicture->m_pBMI==NULL) )
        return;

    int sw = m_pPicture->GetWidth();
    int sh = m_pPicture->GetHeight();
    int dpix = sw * ONEINCH / width;
    int dpiy = sh * ONEINCH / height;

    int dpi = max(dpix, dpiy);

    int dispwidth = sw * ONEINCH / dpi;
    int dispheight = sh * ONEINCH / dpi;

    SetStretchBltMode(hDC, STRETCH_DELETESCANS);

    int x = ( width- dispwidth)/2;
    int y = (height-dispheight)/2;

    if ( StretchJPEG(hDC, x, y, dispwidth, dispheight,
                      m_pPicture->m_pJPEG, m_pPicture->m_nJPEGSize, sw, sh ) )
        return;

    StretchDIBits(hDC, x, y, dispwidth, dispheight, 0, 0, sw, sh,
                  m_pPicture->m_pBits, m_pPicture->m_pBMI, DIB_RGB_COLORS, SRCCOPY);
}
```

The ImagePrint program is designed to be a mini photo printing program, so the KimageCanvas::UponDrawPage tries to maximize use of what may be a piece of expensive photo medium. It calculates the maximum size it can fit in a paper according to its paper margin setting, without changing the photo proportion. With the paper and margin simulation on screen, you can easily adjust the paper margin or switch between portrait and landscape orientation. It calls the function StretchJPEG first, trying to send the smaller-size JPEG image. Failing this gives it the perfect excuse to send a huge BMP image.

By the way, there is at least one printer driver that accepts JPEG images, the driver for the HP 8500 Color Postscript printer. If you print to a file, you can find the JPEG image encoded as binary data in the PostScript file, which really reduces the file size.

17.5 SUMMARY

In this chapter the topics we have discussed so far in this book—device context, lines and curves, area fill, bitmap, font, text, and EMF—all come together, because now we want printing. We've discussed spooler architecture, printing process, spooler API to query about printers and set up printers, GDI printing support functions, and—most importantly—how to use Win32 API support to implement professional printing features in your application. In [Section 17.3](#), a generic displaying and printing class KSurface was developed. It supports truly WYSIWYG presentation of graphics data using a uniform logical coordinate system. [Section 17.4](#) gives two nontrivial sample programs which use the KSurface class and KPageCanvas class to solve real-world problems: source-code printing and JPEG image printing.

With printing fully taken care of, we're finishing the traditional GDI part of the Windows graphics programming. But, like any technology, GDI is still developing, and clearly it's moving to a more colorful, high-performance world with hardware acceleration. To get ready for the future of GDI, the next chapter is about DirectDraw programming.

Further Reading

If you want to know more about how printing and spooler work, grab Microsoft DDK and read its very detailed description of DDI interface, spooler architecture, and how to write a minidriver using the UniDriver architecture. DDK also comes with a sample driver, minidriver, EMF print processor, and a port monitor source code. The older version of Windows NT DDK even has the full source code for a PostScript printer driver.

Examining EMF spool files generated by the spooler can identify quite a few printing problems. Refer back to [Chapter 16](#) for discussions and tools on understanding an EMF.

Sample Programs

[Chapter 17](#) comes with a number of sample programs (see [Table 17-2](#)).

Table 17-2. Sample Programs for [Chapter 17](#)

Directory	Description
Samples\Chapt_17\PrinterDevice	Program for querying printer spooler related information and printer device context information. Based on an earlier program for display device.

Samples\Chapt_17\Printer	Test program for sending raw data to spooler, sending EMF to spooler, print dialog box, page setup dialog box, simple print loop, KSurface class, KPageCanvas class, drawing lines and curves, area fills, bitmaps and texts device independently.
Samples\Chapt_17\CodePrint	Syntax-highlighting source-code viewing and printing, WYSIWGY text formating.
Samples\Chapt_17\ImagePrint	JPEG image viewing and printing program, supporting sending JPEG to printer, uses the JPEG library in the Samples\include\jlib directory.

[< BACK](#) [NEXT >](#)

Chapter 18. DirectDraw and Direct3D Immediate Mode

GDI as a Windows graphics programming API has been around for a long time. So much has changed in the personal computing world that it's time for GDI to make fundamental changes, although we have seen that it keeps improving little by little with every new operating-system releases.

The future of GDI is code-named GDI+, which will be the next-generation GDI from Microsoft. According to published Microsoft documentation (www.microsoft.com/hwdev/video/-GDIplus.htm), GDI+ will create the infrastructure for desktop user interface innovations. It will permit easy integration of 2D and 3D, will bring digital imaging to the desktop, and will raise the bar on desktop graphics and performance. GDI+ will offer enhanced graphics capabilities such as alpha blending, antialiasing, texturing, advanced typography, imaging, hardware acceleration, window layering, double buffering, front-end or back-end composition, gamma control, displayed subpixel rendering, 3D user interface, etc.

You can feel that the common theme of GDI+ is the integration of Microsoft's game-programming API DirectDraw and Direct3D with the traditional GDI. The integration of 2D and 3D starts from the API level and goes well into the DDI (device driver interface) level. On the DDI level, GDI+ will use a mixture of 2D and 3D commands for all hardware-accelerated rendering. GDI+ will define new tokens for primitives that are not already described by existing DirectDraw and Direct3D tokens.

What we are hearing here is that DirectDraw and Direct3D will not be for games and education software only; they will become a core portion of GDI+. In other words, GDI+ is GDI + DirectDraw + Direct3D + more. Now we have every reason to jump-start DirectDraw and Direct3D.

DirectDraw is a rather complicated 2D programming API which may need 200 pages for adequate coverage. Direct3D Immediate Mode is so complicated that it takes lots of 3D computer graphics knowledge to merely understand it, not to say use it effectively. This short chapter is only a brief introduction to DirectDraw and Direct3D. The focuses of our presentation are:

- Introducing the basic concepts, interfaces, and methods to GDI programmers.
- Developing C++ classes for easy DirectDraw and Direct3D programming.
- Demonstrating how GDI features can be used in DirectDraw and Direct3D programming.
- Demonstrating how DirectDraw and Direct3D can be used in “traditional” Windows programs.

18.1 COMPONENT OBJECT MODEL (COM)

The GDI portion of the Win32 API consists of roughly 500 functions which form a complicated function web without clear grouping. For DirectX, Microsoft borrows the COM model to define interface between applications and the operating system. Understanding the basic ideas behind COM is vital to writing correct DirectX programs.

COM Interface

In COM, a set of semantically related abstract methods are grouped together to form an abstract base class, called an *interface* in COM.

As an abstract base class, a COM interface defines syntactically only the prototypes for each method in the interface and the order of these methods. The semantics of these methods are defined vaguely using natural language, instead of a formal specification language that is machine verifiable, because there is no good candidate for such a language.

All COM interfaces are derived from the same root interface, **IUnknown**, which is defined as:

```
class IUnknown
{
public:
    virtual HRESULT __stdcall QueryInterface(REFIID riid,
                                             void ** ppvObject) = 0;
    virtual ULONG __stdcall AddRef(void) = 0;
    virtual ULONG __stdcall Release(void) = 0;
};
```

Every COM interface has a 128-bit reference identifier, which normally contains more information than a book's ISBN, a car's license plate number, or your driver license number. Such identifiers are designed to be globally unique, so naturally they are called GUID (global unique ID). For example, the GUID for the **IUnknown** interface is named **IID_IUnknown** and is defined as:

```
DEFINE_GUID(IID_IUnknown, 0x00000000, 0x0000, 0x0000,
            0xC0, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x46);
```

COM Class

A COM interface is only a specification; it needs to be implemented by a COM class to be really useful. A COM class implementing a COM interface needs to be derived from the COM interface and implement all its methods. For example, here is a sample implementation for the **IUnknown** interface:

```
class KUnknown : public IUnknown
{
    ULONG m_nRef;      // reference count

public:
    KIUnknown()
    {
        m_nRef = 0;      // reference count starts with zero
    }

    ULONG AddRef(void)
    {
        return ++ m_nRef; // AddRef increments reference count
    }

    ULONG Release(void) // Release decrements reference count
    {
        if ( -- m_nRef==0 ) // if reference count reaches zero
        {
            delete this; // delete current object
            return 0;
        }
        return m_nRef;
    }

    HRESULT QueryInterface(REFIID id, void ** ppvObj)
    {
        if ( iid==IID_IUnknown ) // only support IUnknown interface
        {
            * ppvObj = this; // return a pointer to current object
            AddRef(); // increase reference count
            return S_OK;
        }
        return E_NOINTERFACE;
    }
};
```

A COM object normally manages its creation, sharing, and deletion using a reference count; the only exception is singleton COM object that is allocated as global variable so it does not need deletion. So a COM object normally needs at least one member variable, its reference count. The reference count starts with zero when a COM object is first created. The AddRef method increments the reference count, which should be called when a new pointer to a COM object is created. The Release method decrements the reference count, which should be called when a pointer to a COM object is no longer needed. When the reference count reaches zero, the corresponding COM object should be deleted, unless it's a singleton object. The KUnknown class assumes its instances are allocated from heap using the new operator, so the delete operator should be invoked to delete them. Matching AddRef with Release is critical for COM programs to work. Extra AddRef leaves undeleted COM objects when they should be deleted, which results in memory/resource leak. Extra Release discards COM objects too early, which most likely

will lead to access violations.

A COM object must implement every COM interface it derives from. So it would implement the IUnknown interface and possibly other interfaces. The QueryInterface method is designed for a client of a COM class to query for its support for COM interfaces. QueryInterface accepts a reference to a GUID, returning a pointer to a COM object cast to a certain COM interface and a success or failure result. For the KUnknown class, which only implements the IUnknown interface, QueryInterface checks if the GUID parameter equals IID_IUnknown; if so, the 'this' pointer is passed back, the reference count is incremented, and S_OK is returned; otherwise, the error code E_NOINTERFACE is returned.

A pointer returned by QueryInterface is called an *object interface pointer*, or simply an *interface pointer*. Strictly speaking, an interface pointer is not a pointer to a COM interface, because a COM interface is only a specification, a contract, but not a runtime object. An interface pointer is a pointer to somewhere in a COM object whose first double word contains a pointer to a table of method pointers, which implements the virtual methods defined in a COM interface. In short, an interface pointer is a pointer to a pointer to an array of virtual method implementations. For our simple sample, which only implements a single interface, the interface pointer is the object pointer.

Each COM class also has a GUID, which uniquely identifies it. COM class GUID normally uses a separate type CLSID, which has exactly the same format as a GUID.

Creating a COM Object

We have a COM interface and a COM class. How can another component use it? The advantage of COM comes mainly from its strict separation of interface and implementation, which means the client-side components of a COM class cannot see its class declaration. When class declaration is not available, you cannot call the new operator to create an instance of the class, call the delete operator to delete an object, allocate a COM object on the stack.

To allow client components to create a COM object, COM defines a generic COM interface IClassFactory, which is responsible for creating a COM object. Interface IClassFactory's main method is CreateInstance, which accepts an interface GUID, creates a new COM object, and returns an interface pointer. A class factory class, whose sole purpose is to create instances of the formal class, normally accompanies each COM class. COM classes are normally implemented in a DLL, whose main exported function is DllGetClassObject, which is defined as:

```
STDAPI DllGetClassObject(REFCLSID rclsid, REFIID riid,  
                         LPVOID * ppv);
```

The job of a COM DLL's DllGetClassObject function is to check for any class GUID implemented by the current DLL. If a match is found, it finds the right class factory, and returns a pointer to its class factory object, which can be used to create one or many instances of the COM class.

COM DLLs are required to register with the operating system once they are installed on a system, so the operating system knows exactly where they are and what COM classes they implement. The generic way to create a COM object is by using a single generic function provided by the Win32 API, CoCreateInstance.

CoCreateInstance accepts COM class CLSID and a COM interface IID, searches the registry to find the right COM component, loads it into application's process address space, locates its DllGetClassObject, and calls it to create a COM object.

HRESULT

Most COM interface methods return a 32-bit signed value of the type HRESULT; Add Ref and Release are two exceptions. An HRESULT is composed of three fields, which tell if a method call was successful, the facility in which it fails, and a status code.

The highest bit (bit 31) of HRESULT indicates whether a call is successful, 1 for failure and 0 for success. Bits 25 through 16 form an 11-bit facility code. Bits 15 through bit 0 form a 16-bit status code.

The first bit of an HRESULT is most important, because it tells whether the call is successful. A good way to tell if a call is successful is using the SUCCEEDED macro, which checks if an HRESULT is nonnegative. The opposite of the SUCCEEDED macro is the FAILED macro, which checks whether an HRESULT is negative. COM methods usually return S_OK (0) for success, but it's not a good practice to compare an HRESULT with S_OK. Likewise, DirectDraw methods normally return DD_OK (0) for success.

The facility code is normally only put into an HRESULT when a call has failed so that the failure location can be identified to some extent. DirectX uses 0x876 and 0x878 as its facility codes. Here is how a DirectDraw/Direct3D error HRESULT is formed.

```
#define _FACDD          0x876
#define MAKE_DDHRESULT(code) MAKE_HRESULT(1, _FACDD, code)
```

For example, the HRESULT for creating a DirectDraw surface with an invalid pixel format, DDERR_INVALIDPIXELFORMAT, is defined as MAKE_DDH_RESULT(145).

DirectX defines around 200 different error HRESULT values, since it is critical for the application to find possible reasons when a call fails.

DirectX and COM

DirectX uses dozens of COM interfaces and COM classes. But it does not follow the COM way strictly. Most notably, a DirectX COM object is either created using a special exported function, or created from an existing COM object, not using the generic `CoCreateInstance` function.

The center of DirectDraw and Direct3D Immediate Mode is the IDirectDraw interface series. Once a COM interface is published, which means it's formally documented and communicated to other parties and may be in use, it cannot be changed. When new functionality is needed, a new interface must be created. For example, after the initial IDirectDraw interface came the IDirectDraw2 interface, the IDirectDraw4 interface, and the latest IDirectDraw7 interface used in DirectX 7.0.

There is an exported special DirectDraw function to create a DirectDraw object that supports the IDirectDraw7 interface:

```
HRESULT WINAPI DirectDrawCreateEx(GUID * lpGUID, LPVOID * lpDD,  
        REFIID iid, IUnknown * pUnkOuter);
```

The first parameter is a pointer to a GUID identifying a graphics device supporting DirectDraw/Direct3D, which can be a hardware implementation, a hardware implementation on a second monitor, or a software simulation. If NULL is passed, the active display device is assumed. You can use DDCREATE_EMULATIONONLY to select software enumeration, or DDCREATE_HARDWAREONLY to favor hardware accelerated implementation. This feature is particularly useful for software testing and workaround problems faced in one implementation. Another function, DirectDraw EnumerateEx, is provided to enumerate current DirectDraw/Direct3D implementations on your system. Your program can use it to find the right implementation to meet your requirements. Conceivably, someone could write a DirectDraw/Direct3D implementation, which would implement metafile-like features for DirectX, or a high-resolution large-size/high-resolution DirectX device for printing on printers.

The second parameter is a pointer to a variable, which receives an interface pointer when DirectDrawCreateEx successfully creates a DirectDraw object. The third parameter is only allowed to be IID_IDirectDraw7, the GUID for the IDirectDraw7 interface. The last parameter is reserved for compatibility with COM aggregation, and must be NULL for the moment.

Here is an example of how to use DirectDrawCreateEx to initialize the DirectDraw environment.

```
void Demo_DirectDrawCreateEx(KLogWindow * pLog)
{
    IDirectDraw7 * pDD = NULL;

    HRESULT hr = DirectDrawCreateEx(NULL, (void **) & pDD,
                                   IID_IDirectDraw7, NULL);

    if ( FAILED(hr) )
    {
        pLog->Log("DirectDrawCreateEx failed (%x)", hr);
        return;
    }

    CheckInterface(pLog, pDD, IID_IDirectDraw7, "IDirectDraw7");
    CheckInterface(pLog, pDD, IID_IDirectDraw4, "IDirectDraw4");
    CheckInterface(pLog, pDD, IID_IDirectDraw2, "IDirectDraw2");
    CheckInterface(pLog, pDD, IID_IDirectDraw, "IDirectDraw" );
    CheckInterface(pLog, pDD, IID_IUnknown, "IUnknown" );

    CheckInterface(pLog, pDD, IID_IDDVideoPortContainer,
                  "IDDVideoPortContainer" );
    CheckInterface(pLog, pDD, IID_IDirectDrawKernel,
                  "IDirectDrawKernel" );
    CheckInterface(pLog, pDD, IID_IDirect3D7, "IDirect3D7");
    CheckInterface(pLog, pDD, IID_IDirect3D3, "IDirect3D3");

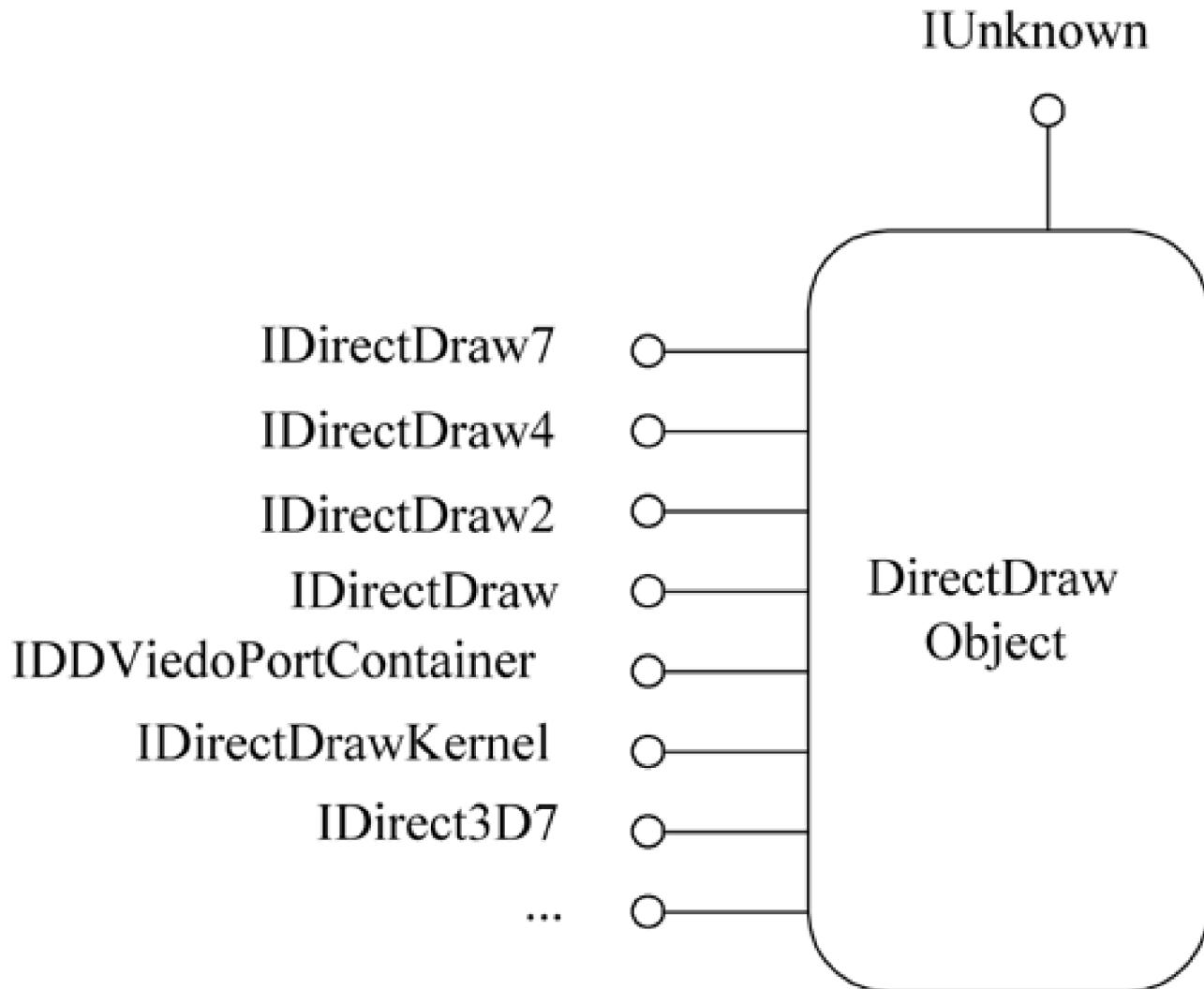
    pDD->Release();
}
```

The Demo_DirectDrawCreateEx routine asks the system to create a DirectDraw object and returns its IDrectDraw7

interface pointer. If the system has DirectX 7.0 installed, the routine checks for the object's support for other COM interfaces, using a subroutine CheckInterface. CheckInterface uses QueryInterface to query for a new interface pointer, dumps some data to a logging window, and releases it. Finally, Demo_DirectDrawCreateEx calls the Release method to release the DirectDraw object.

You will find that the DirectDraw object created by DirectDrawCreateEx supports all the interfaces listed in the routine, except the IDirect3D3 interface. [Figure 18-1](#) shows the COM interfaces supported by a DirectDraw object, using the traditional COM diagram.

Figure 18-1. COM interfaces supported by a DirectDraw object.



A COM object supporting all the interfaces shown in [Figure 18-1](#) would be very complicated, especially where there is a mix of DirectDraw and Direct3D interfaces. From the pointers returned by QueryInterface, you could notice that a DirectDraw object is not created as a whole when it is first created. The system is smart enough to allocate and initialize the right part of the object only when it's needed.

As we mentioned, an object interface pointer is a pointer to a pointer to a virtual function table. If C++ compiler is used to create a COM object, the compiler will assemble a virtual function table in program's constant area and generate code to put the virtual function table pointer into a newly created object. DirectDraw object's virtual function table is assembled in a global read/writeable data segment during runtime. Thus, it's very easy to substitute different implementations, depending on current system setting, or even to hijack DirectX method calls at runtime as a

powerful debugging aid. [Section 4.3](#) of this book discusses techniques for spying on DirectX COM interfaces.

18.2 DIRECTDRAW BASICS

Although COM interfaces are based on the C++ abstract base class, using COM interfaces is much harder than using C++ classes. COM interfaces are designed to make components work with each other easily at the binary level, not to make the programmer's life easy at the source-code level. The usual practice in using COM interfaces is wrapping them again using C++ classes.

DirectX COM interfaces are no better than other COM interfaces. They normally have a limited number of methods, with complicated parameters defined using huge structures, and dozens of flags. For example, to draw a bitmap onto a surface, GDI provides several functions: PatBlt, BitBlt, StretchBlt, PlgBlt, MaskBlt, TransparentBlt, and AlphaBlend; DirectDraw only has two drawing methods: FastBlt and Blt.

Given the complexity of fully describing DirectDraw interfaces, we will not describe the full details of each method. Instead we will discuss them in the context of their use in C++ wrapping class and actual drawing code. You can easily refer back to MSDN for full details of a COM interface or a method within it.

IDirectDraw7 Interface

[Listing 18-1](#) shows the KDirectDraw class, which is a basic wrapper class for the IDirectDraw7 interface.

Listing 18-1 Wrapper Class for IDirectDraw7 interface

```
#define SAFE_RELEASE(inf) { if (inf) { inf->Release(); inf = NULL; } }

// Wrapper for IDirectDraw7 interface, supporting primary surface
class KDirectDraw
{
protected:
    IDirectDraw7 * m_pDD;
    RECT      m_rcDest; // destination rectangle
    KDDSurface m_primary;

    virtual HRESULT Discharge(void);

public:
    KDirectDraw(void);

    virtual ~KDirectDraw(void)
    {
        Discharge();
    }
}
```

```
void SetClientRect(HWND hWnd);

virtual HRESULT SetupDirectDraw(HWND hTop, HWND hWnd,
    int nBufferCount=0, bool bFullScreen=false,
    int width=0, int height=0, int bpp=0);
};

KDirectDraw::KDirectDraw(void)
{
    m_pDD = NULL;
}

HRESULT KDirectDraw::Discharge(void)
{
    m_primary.Release();
    SAFE_RELEASE(m_pDD);
    return S_OK;
}

void KDirectDraw::SetClientRect(HWND hWnd)
{
    GetClientRect(hWnd, &m_rcDest);
    ClientToScreen(hWnd, (POINT*)&m_rcDest.left);
    ClientToScreen(hWnd, (POINT*)&m_rcDest.right);
}

HRESULT KDirectDraw::SetupDirectDraw(HWND hTop, HWND hWnd,
    int nBufferCount, bool bFullScreen,
    int width, int height, int bpp)
{
    HRESULT hr = DirectDrawCreateEx(NULL, (void **) &m_pDD,
        IID_IDirectDraw7, NULL);
    if ( FAILED(hr) )
        return hr;

    if ( bFullScreen )
        hr = m_pDD->SetCooperativeLevel(hTop,
            DDSCL_FULLSCREEN | DDSCL_EXCLUSIVE);
    else
        hr = m_pDD->SetCooperativeLevel(hTop, DDSCL_NORMAL);

    if ( FAILED(hr) )
        return hr;

    if ( bFullScreen )
    {
        hr = m_pDD->SetDisplayMode(width, height, bpp, 0, 0);
        if ( FAILED(hr) )
            return hr;
    }
}
```

```
    SetRect(& m_rcDest, 0, 0, width, height);
}
else
    SetClientRect(hWnd);

hr = m_primary.CreatePrimarySurface(m_pDD, nBufferCount);

if ( FAILED(hr) )
    return hr;

if ( ! bFullScreen )
    hr = m_primary.SetClipper(m_pDD, hWnd);

return hr;
}
```

The KDirectDraw class defines three member variables: an IDirectDraw7 interface pointer m_pDD, a destination rectangle m_rcDest, and a primary drawing surface m_primary. The drawing surface uses another yet-to-be-defined class KDDSurface. Its constructor sets m_pDD to NULL pointer, the Discharge method frees allocated resources, and the destructor calls the Discharge method to make sure everything is freed.

The SetupDirectDraw method creates a DirectDraw object and prepares for basic drawing using DirectDraw. It accepts seven parameters: a top-level window handle, a child window handle which will use DirectDraw, a count of number of backup buffers, a Boolean flag for full-screen mode, and three integers to specify the screen format for full-screen mode. The SetupDirectDraw method calls the function DirectDrawCreateEx to create a DirectDraw object, which returns an IDirectDraw7 interface pointer in m_pDD. If the call is successful, it calls IDirectDraw7::SetCooperative Level to tell the system main window handle whether the full-screen mode or a windowed mode is required.

Full-screen mode DirectX programs are normally for games or educational titles, which claim exclusive ownership of DirectX supporting resources. DirectX also supports rendering in windowed mode and even multiple instances of DirectDraw in multiple windows. A full-screen DirectX program normally changes the display resolution and color format to best match its design. For example, programs using palette animation need to switch the screen to a 256-color display mode; programs worried about performance may switch screen to a low-resolution display mode to reduce video memory usage and the amount of data transfer. The SetupDirectDraw method calls the IDirectDraw7::SetDisplayMode method to switch the display mode. A full-screen program should use the IDirectDraw7::EnumerateDisplayModes method to enumerate supported display modes; otherwise the call may fail. For example, quite a few display cards support 32-bpp display modes but not 24-bpp display modes.

The SetupDirectDraw method also calculates the rectangle defining the intended drawing surface in member variable m_rcDest. When running in full-screen mode, the destination rectangle is the whole display screen; otherwise it's set to be the client area of a window specified by the hWnd parameter. Note that two window handles are passed to SetupDirectDraw, so that we're not restricted to use DirectDraw only in the main window.

The SetupDirectDraw method ends with creating the primary drawing surface and setting up clipping on it, for which we need the KDDSurface class.

IDirectDrawSurface7 Interface

To draw anything with DirectDraw, you need a surface, which is similar to a device context in GDI. A DirectDraw surface could represent the current display monitor or an off-screen drawing surface backed up by a memory buffer. The former is similar to a display device context, and the latter is close to a memory device context backed up by a DIB section.

The interface for a DirectDraw surface is currently the IDirectDrawSurface7 interface, which has around four dozen methods. Compared with the number of GDI functions accepting device context handle as parameter, you would wish for more methods in the IDirectDrawSurface7 to make programming easier.

We will describe some of the basic IDirectDrawSurface7 methods as we use them in our wrapping class KDDSurface. Besides being a simple wrapper, the KDDSurface class also provides quite a few methods to make DirectDraw surface easy to use. The class declaration for KDDSurface is shown in [Listing 18-2](#), while its implementation will be shown as needed.

Listing 18-2 Wrapper Class for IDirectDrawSurface7 interface

```
class KDDSurface
{
protected:
    IDirectDrawSurface7 * m_pSurface;
    DDSURFACEDESC2      m_ddsd;
    HDC                 m_hDC;

public:
    KDDSurface();
    virtual void Discharge(void); // release before destructor

    virtual ~KDDSurface() // make sure everything is released
    {
        Discharge();
    }

    operator IDirectDrawSurface7 * & ()
    {
        return m_pSurface;
    }

    operator HDC ()
    {
        return m_hDC;
    }

    int GetWidth(void) const
    {
        return m_ddsd.dwWidth;
    }
```

```
int GetHeight(void) const
{
    return m_ddsd.dwHeight;
}

HRESULT CreatePrimarySurface(IDirectDraw7 * pDD, int nBackBuffer);

const DDSURFACEDESC2 * GetSurfaceDesc(void);
virtual HRESULT RestoreSurface(void); // restore surface if lost

// DirectDraw Blting
HRESULT SetClipper(IDirectDraw7 * pDD, HWND hWnd);

HRESULT Blt(LPRECT prDest, IDirectDrawSurface7 * pSrc,
            LPRECT prSrc, DWORD dwFlags, LPDDBLTFX pDDBltFx=NULL)
{
    return m_pSurface->Blt(prDest, pSrc, prSrc, dwFlags, pDDBltFx);
}

DWORD ColorMatch(BYTE red, BYTE green, BYTE blue);
HRESULT FillColor(int x0, int y0, int x1, int y1, DWORD fillcolor);
HRESULT BitBlt(int x, int y, int w, int h,
               IDirectDrawSurface7 * pSrc, DWORD flag=0);

HRESULT BitBlt(int x, int y, KDDSurface & src, DWORD flag=0)
{
    return BitBlt(x, y, src.GetWidth(), src.GetHeight(), src, flag);
}

HRESULT SetSourceColorKey(DWORD color);

// Drawing using GDI device context
HRESULT GetDC(void);      // Get DC device context handle
HRESULT ReleaseDC(void);

HRESULT DrawBitmap(const BITMAPINFO * pDIB, int dx, int dy,
                  int dw, int dh);

// Direct Pixel Access
BYTE * LockSurface(RECT * pRect=NULL);
HRESULT Unlock(RECT * pRect=NULL);

int GetPitch(void) const
{
    return m_ddsd.lPitch;
}
```

The KDDSurface class has three member variables: a pointer to the IDirectDrawSurface7 interface, a surface description structure, and a GDI device context handle. The IDirectDrawSurface7 interface pointer is returned by the

system when a DirectDraw surface is created, and it acts as the sole interface for the surface to the system. The surface description structure, which is of type DDSURFACEDESC2, is used to describe the surface format. It holds a surface's critical information, such as type, width, height, pitch, memory address, pixel format, etc. Each DirectDraw surface can be associated with a GDI DC handle, which can be used to use GDI to draw onto a DirectDraw surface.

Although DirectDraw provides a single method, IDirectDraw7::CreateSurface, to create a surface, there are many different ways a surface can be created. The KDDSurface class provides several methods to make surface creation easy. Here is KDD-Surface's constructor and the method for creating a primary drawing surface, which is used by the KDirectDraw class.

```
KDDSurface::KDDSurface()
{
    m_pSurface = NULL;
    m_hDC     = NULL;
    m_nDCRef  = 0;

    memset(& m_ddsd, 0, sizeof(m_ddsd));
    m_ddsd.dwSize = sizeof(m_ddsd);
}

HRESULT KDDSurface::CreatePrimarySurface(IDirectDraw7 * pDD,
                                         int nBufferCount)
{
    if ( nBufferCount==0 )
    {
        m_ddsd.dwFlags      = DDSD_CAPS;
        m_ddsd.ddsCaps.dwCaps  = DDSCAPS_PRIMARYSURFACE;
    }
    else
    {
        m_ddsd.dwFlags      = DDSD_CAPS | DDSD_BACKBUFFERCOUNT;
        m_ddsd.ddsCaps.dwCaps  = DDSCAPS_PRIMARYSURFACE | DDSCAPS_FLIP |
                               DDSCAPS_COMPLEX | DDSCAPS_VIDEOMEMORY;
        m_ddsd.dwBackBufferCount = nBufferCount;
    }

    return pDD->CreateSurface(& m_ddsd, & m_pSurface, NULL);
}
```

For a full-screen DirectX program, display hardware supports a simple surface, double-buffered surface, or triple-buffered surface. A simple surface has only a single drawing buffer where all drawing happens and video display signal is generated. A double-buffered surface has two buffers, one being displayed and one being drawn to. DirectX provides the IDirectDrawSurface7::Flip method to flip back and front buffers. Triple buffering uses three buffers: one being displayed, one waiting to be flipped to the front, and one being drawn to. Double or triple buffering is the key to smooth, flickerless drawing. But it is only available in full-screen, exclusive mode, because hardware buffer-flipping can only flip the whole display area, not just a window. For flicker less drawing in a windowed DirectDraw program, you have to use an off-screen drawing surface and explicitly copy its contents to the primary surface.

The CreatePrimarySurface method accepts an IDirectDraw7 interface pointer and a back buffer count. If the back buffer count is zero, it sets up two flags in the DDSURFACEDESC2 structure to create a simple primary surface; otherwise extra flags and the back buffer count need to be set. Note that the m_ddsd member variable, which is of type DDSURFACEDESC2, is partially initialized in the class constructor.

Drawing on a DirectDraw Surface

Given a DirectDraw surface, you can draw on it. There are three ways to draw on a DirectDraw surface: using hardware-accelerated IDirectDrawSurface7 methods, using GDI, or directly manipulating the pixels in the surface's frame buffer.

Hardware-Accelerated Drawing

IDirectDrawSurface7 only provides three drawing methods: Blt, BltFast, and Blt Batch. However, the last method is not implemented. Because only Blt and BltFast can be hardware accelerated, they should be used whenever possible to achieve good performance. Here is the declaration for the Blt method.

```
HRESULT Blt(LPRECT lpDestRect, LPDIRECTDRAWINTERFACE7 lpDDSrcSurface,  
           LPRECT lpSrcRect, DWORD dwFlags, LPDBLTFX lpDDBltFx);
```

The Blt method is similar to GDI's StretchBlt function; it transfers a rectangle region in the source surface to a rectangle region on the destination surface. The destination surface is referenced by the current IDirectDrawSurface7 pointer, and the destination rectangle is specified by the lpDestRect parameter. The source surface is specified by the lpDDSrcSurface parameter, and the source rectangle is specified by the lpSrcRect parameter. The dwFlags parameter contains flags controlling how bitblting is performed; the last parameter is a pointer to a DBLTFX structure with more fields to control the pixel transfer.

The simplest use of Blt is filling a destination with a solid color, similar to what PatBlt does. Here is the KDDSurface::FillColor that wraps solid color fill.

```
HRESULT KDDSurface::FillColor(int x0, int y0, int x1, int y1,  
                               DWORD fillcolor)  
{  
    DBLTFX fx;  
    fx.dwSize = sizeof(fx);  
    fx.dwFillColor = fillcolor;  
  
    RECT rc = { x0, y0, x1, y1 };  
  
    return m_pSurface->Blt(&rc, NULL, NULL, DDBLT_COLORFILL, &fx);  
}
```

The FillColor method fills a RECT structure from four parameters. The source surface and source rectangle are not needed. The dwFlags parameter is DDBLT_COLORFILL, and the DBLTFX is mainly the fill color.

GDI Drawing

Although DirectDraw is designed to let game programmers get away from GDI, they are not allowed to go too far away, because from time to time GDI's help is needed. Although DirectDraw provides drawing with hardware acceleration, its drawing functions are very limited. In DirectX, GDI still plays a big role in drawing text and initializing the surface with bitmaps. To use GDI with a DirectDraw surface, use the IDirectDrawSurface::GetDC method to retrieve a GDI device context handle, which can later be freed using the ReleaseDC method.

Here are wrapper methods for GetDC, ReleaseDC, and a method to draw a DIB on a DirectDraw surface using GDI.

```
HRESULT KDDSurface::GetDC(void)
{
    return m_pSurface->GetDC(&m_hDC);
}

HRESULT KDDSurface::ReleaseDC(void)
{
    if ( m_hDC==NULL )
        return S_OK;
    HRESULT hr = m_pSurface->ReleaseDC(m_hDC);
    m_hDC = NULL;
    return hr;
}

HRESULT KDDSurface::DrawBitmap(const BITMAPINFO * pDIB, int x, int y,
                               int w, int h)
{
    if ( SUCCEEDED(GetDC()) )
    {
        StretchDIBits(m_hDC, x, y, w, h,
                      0, 0, pDIB->bmiHeader.biWidth, pDIB->bmiHeader.biHeight,
                      &pDIB->bmiColors[GetDIBColorCount(pDIB)], pDIB,
                      DIB_RGB_COLORS, SRCCOPY);

        return ReleaseDC();
    }
    else
        return E_FAIL;
}
```

The DrawBitmap method draws a packed device-independent bitmap onto a DirectDraw surface using StretchDIBits, which is perfect for loading a bitmap onto a DirectDraw surface. If performance is critical, DrawBitmap should only be used to load bitmap to an off-screen surface or a texture surface, which is then drawn to the primary surface repetitively using the hardware-accelerated Blt method. You will find most DirectX literatures use DDB or DIB section with the memory device context to load a bitmap, which requires creating two GDI objects. Using DIB to load a bitmap is the preferable method, because it does not change color as DDB does, and it does not need extra

GDI resource.

The device context handle returned by the IDirectDrawSurface7::GetDC method is classified as a memory DC handle. If you can GetObjectType on it, GDI returns OBJ_MEMDC. But it is by no means a normal memory DC handle, as it is not created with CreateCompatibleDC or even CreateDC. It's created with a special system service call named NtGdiDdGetDC. Given the DC handle, you can use Get CurrentObject(m_hDC, OBJ_BITMAP) to query the bitmap selected in it; a DIB section handle will be returned. If you use GetObject to query for its DIBSection structure, a valid DIBSection structure will be filled. The only thing that appears unique is that the pointer to the bits is a kernel address space pointer, which can't be accessed in user mode. Nevertheless, the DIB section is not a normal DIB section, because the Direct Draw surface can have strange pixel formats not commonly classified as standard DIB formats. For example, some display drivers may support 8-bpp 2-3-2 RGB surface, or 16-bpp 4-4-4-4 RGB surface.

Direct Pixel Access

The combination of Blt, BltFast, and GDI functions is still not powerful enough in some cases. For example, to draw a simple pixel in a DirectDraw surface, calling either Blt or GDI's SetPixel will be slow. DirectDraw allows direct access to a surface's frame buffer through surface locking. The IDirectDrawSurface7::Lock method maps a surface's frame buffer to a user-mode addressable memory block. It handles off-screen surfaces and the primary surface in the same way. Through the surface frame buffer pointer, a locked surface can be accessed in the same way a DIB/DIB section's pixel array can be accessed; that is to say, endless interesting algorithms can be implemented. Surface locking is a perfect match for old game programs running under DOS, which use direct access to video memory to achieve high performance unobtainable with GDI.

Here are the wrapper methods for surface locking and unlocking.

```
BYTE * KDDSurface::LockSurface(RECT * pRect)
{
    if ( FAILED(m_pSurface->Lock(pRect, & m_ddsd,
        DDLOCK_SURFACEMEMORYPTR | DDLOCK_WAIT, NULL)) )
        return NULL;
    else
        return (BYTE *) m_ddsd.lpSurface;
}

HRESULT KDDSurface::Unlock(RECT * pRect)
{
    m_ddsd.lpSurface = NULL; // make it unavailable

    return m_pSurface->Unlock(pRect);
}
```

The LockSurface method calls the IDirectDrawSurface7::Lock method to lock a rectangle region on a surface, which fills a DDSURFACEDESC2 structure. The most important fields filled are the surface pixel format, width, height, the scan line pitch, and a pointer to the frame buffer. If a valid rectangle is given to the Lock method, the lpSurface pointer points to the address of the top-left pixel in the rectangle; otherwise it points to the first pixel of the surface. The Unlock method reverses the action of surface locking.

The pointer returned by the Lock method can be used to directly access the contents of a surface. But extreme care should be taken, because there is no clipping involved. Accessing out-of-boundary pixels can either cause an access violation or step on other people's toes if running under windowed mode. The application is responsible for implementing the necessary clipping.

Here is a sample routine that does a little more than a plain solid color fill.

```
case 4:  
    for (i=0; i<width; i++)  
    {  
        * (unsigned *) pS = color;  
        pS += sizeof(unsigned);  
    }  
    break;  
  
default:  
    return FALSE;  
}  
}  
  
surface.Unlock();  
return TRUE;  
}  
else  
    return FALSE;  
}
```

The PixelFillRect routine fills a rectangle region with horizontal lines of different colors. It uses LockSurface to get a pointer to a surface, calculates the pixel address according to its pixel format, and then draws line by line, pixel by pixel, by directly copying pixel data to the surface frame buffer.

Using the Blt/BltFast methods, GDI drawing and direct pixel access with surface locking are mutually exclusive. When a GDI DC handle is open, surface locking will fail; when the surface is locked, GDI drawing will fail. On Windows 95/98, surface locking normally grabs a system mutex, which blocks other threads from using the 16-bit GDI for thread reentrance reasons. So the surface should be locked only when needed and unlocked when it's not needed.

Color Matching

For the KDDSurface::FillColor method, the fill color parameter is typed as DWORD, not COLORREF as we are familiar with in GDI. It is in the physical color format as required by the particular surface, not a generic RGB value. DirectDraw is really a thin layer above the hardware. It's so thin that there is no easy way to specify a color in Direct Draw using RGB color channels. Physical colors acceptable to DirectDraw all depend on the surface pixel format.

Here is a generic color-matching method implemented for the KDDSurface class.

```
const DDSURFACEDESC2 * KDDSurface::GetSurfaceDesc(void)  
{  
    if ( SUCCEEDED(m_pSurface->GetSurfaceDesc(& m_ddsd) )  
        return & m_ddsd;  
    else  
        return NULL;  
}
```

```
DWORD KDDSurface::ColorMatch(BYTE red, BYTE green, BYTE blue)
{
    if ( m_ddsd.ddpfPixelFormat.dwSize==0 ) // not initialized
        GetSurfaceDesc();           // get surface description

    const DDPIXELFORMAT & pf = m_ddsd.ddpfPixelFormat;

    if ( pf.dwFlags & DDPF_RGB )
    {
        // x-5-5-5
        if ( (pf.dwRBitMask == 0x7C00) && (pf.dwGBitMask == 0x03E0) &&
            (pf.dwBBitMask==0x001F) )
            return ((red>>3)<<10) | ((green>>3)<<5) | (blue>>3);

        // 0-5-6-5
        if ( (pf.dwRBitMask == 0xF800) && (pf.dwGBitMask == 0x07E0) &&
            (pf.dwBBitMask==0x001F) )
            return ((red>>3)<<11) | ((green>>2)<<5) | (blue>>3);

        // x-8-8-8
        if ( (pf.dwRBitMask == 0xFF0000) && (pf.dwGBitMask == 0xFF00) &&
            (pf.dwBBitMask==0xFF) )
            return (red<<16) | (green<<8) | blue;
    }

    DWORD rslt = 0;

    if ( SUCCEEDED(GetDC()) ) // get GDI DC
    {
        COLORREF old = GetPixel(m_hDC, 0, 0); // save original pixel
        SetPixel(m_hDC, 0, 0, RGB(red, green, blue)); // put RGB pixel
        ReleaseDC();
        const DWORD * pSurface = (DWORD *) LockSurface(); // lock surface

        if ( pSurface )
        {
            rslt = * pSurface;           // read first DWORD
            if ( pf.dwRGBBitCount < 32 )
                rslt &= (1 << pf.dwRGBBitCount) - 1; // truncate for bpp
            Unlock();                  // unlock surface
        }
        else
            assert(false);

        GetDC();
        SetPixel(m_hDC, 0, 0, old); // put original pixel back
        ReleaseDC();              // release GDI DC
    }
    else

```

```
    assert(false);

    return rsIt;
}
```

The ColorMatch method converts a color specified using its red, green, and blue channels to a physical color, a DWORD ready to be put into a frame buffer. It makes sure a correct DDPIXELFORMAT structure is set up; if not, GetSurfaceDesc is called to query the current surface description structure. It then tries to determine if the surface is of the common 15-bpp, 16-bpp, 24-bpp, and 32-bpp RGB formats by comparing its channel bit masks. If the masks match, ColorMatch uses bit operation to combine RGB channels to the right physical color.

If this quick path fails, GDI's help is asked. The code retrieves a GDI device handle for the surface, uses GetPixel to save the pixel at (0, 0), SetPixel to set pixel (0, 0) to an RGB value, and then reads its physical color using a locked surface. Before returning, the code restores the original pixel (0, 0) using another SetPixel call. Note that, because the GDI surface handle and surface locking are exclusive, the code has to release DC handle before the locking surface, and acquire it again to restore the changed pixel. That is too much work just to convert an RGB value to a physical color. So results from the ColorMatch method should be reused when possible. The KDDSurface::Fill Color method is designed to accept a physical color, instead of a logical color, to allow caching of physical colors.

IDirectDrawClipper Interface

The primary surface DirectDraw creates always covers the whole screen display. With it, you can draw anywhere on the screen, just like a DC returned by GetDC(NULL).

To restrict the area of drawing, DirectDraw supports clipping using a clipping object, which is abstracted to be the IDirectDrawClipper interface. A clipper object is normally created and then attached to a DirectDraw surface. The clipping region represented by a clipper object is specified by a so-called clip list, which is nothing other than the RGNDATA structure used by GDI to access its region object. The easiest way to initialize a clipper object with the right clip list is to associate it with a window. The operating system automatically manages its clip list when the window moves or resizes, including managing visibility.

Here is the KDDSurface::SetClipper method, which creates a clipper object, associates it with a window, and attaches it to a surface. It's called by the KDirectDraw ::SetupDirectDraw method after a primary surface is created.

```
HRESULT KDDSurface::SetClipper(IDirectDraw7 * pDD, HWND hWnd)
{
    IDirectDrawClipper * pClipper = NULL;

    HRESULT hr = pDD->CreateClipper(0, &pClipper, NULL);
    if (FAILED(hr))
        return hr;

    pClipper->SetHWND(0, hWnd);
    m_pSurface->SetClipper(pClipper);

    return pClipper->Release();
}
```

Note that, after calling IDirectDrawSurface7::SetClipper, the surface has a pointer to the clipper object, so its reference count is incremented. Then the IDirectDrawClipper::Release method can be called to release the function local variable reference to the object.

Simple DirectDraw Window

Now we have all the right classes and methods to construct a simple DirectDraw window. [Listing 18-3](#) shows a simple yet complete window class that supports DirectDraw.

Listing 18-3 Simple DirectDraw Window Class

```
class KDDWin : public KWindow, public KDirectDraw
{
    void OnNCPaint(void)
    {
        RECT rect;
        GetWindowRect(m_hWnd, &rect);

        DWORD dwColor[18];

        for (int i=0; i<18; i++)
            dwColor[i] = m_primary.ColorMatch(0, 0, 0x80 + abs(i-9)*12);

        FillRect(m_primary, rect.left+24, rect.top+4, rect.right -
                 88 - rect.left, 18, dwColor, 18);

        BYTE * pSurface = m_primary.LockSurface(NULL); // just for address
        m_primary.Unlock(NULL);

        if ( SUCCEEDED(m_primary.GetDC()) )
        {
            TCHAR temp[MAX_PATH];
            const DDSURFACEDESC2 * pDesc = m_primary.GetSurfaceDesc();

            if ( pDesc )
                wsprintf(temp, "%dx%d %d-bpp, pitch=%d, lpSurface=0x%x",
                        pDesc->dwWidth, pDesc->dwHeight,
                        pDesc->ddpfPixelFormat.dwRGBBitCount,
                        pDesc->lPitch, pSurface);
            else
                strcpy(temp, "LockSurface failed");

            SetBkMode(m_primary, TRANSPARENT);
            SetTextColor(m_primary, RGB(0xFF, 0xFF, 0));
            TextOut(m_primary, rect.left+24, rect.top+4, temp, _tcslen(temp));
        }
    }
}
```

```
m_primary.ReleaseDC();
}

}

void OnDraw(void)
{
SetClientRect(m_hWnd);
int n = min(m_rcDest.right-m_rcDest.left,
m_rcDest.bottom-m_rcDest.top)/2;

for (int i=0; i<n; i++)
{
DWORD color = m_primary.ColorMatch( 0xFF*(n-1-i)/(n-1),
0xFF*(n-1-i)/(n-1), 0xFF*i/(n-1) );

m_primary.FillColor(m_rcDest.left+i, m_rcDest.top+i,
m_rcDest.right-i, m_rcDest.bottom-i, color);
}
}

LRESULT WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
switch( uMsg )
{
case WM_CREATE:
m_hWnd = hWnd;
if (FAILED(SetupDirectDraw(GetParent(hWnd), hWnd, false)))
{
MessageBox(NULL, _T("Unable to Initialize DirectDraw"),
_T("KDDWin"), MB_OK);
CloseWindow(hWnd);
}
return 0;

case WM_PAINT:
OnDraw();
ValidateRect(hWnd, NULL);
return 0;

case WM_NCPAINT:
DefWindowProc(hWnd, uMsg, wParam, lParam);
OnNCPaint();
return 0;

case WM_DESTROY:
PostQuitMessage(0);
return 0;

default:
```

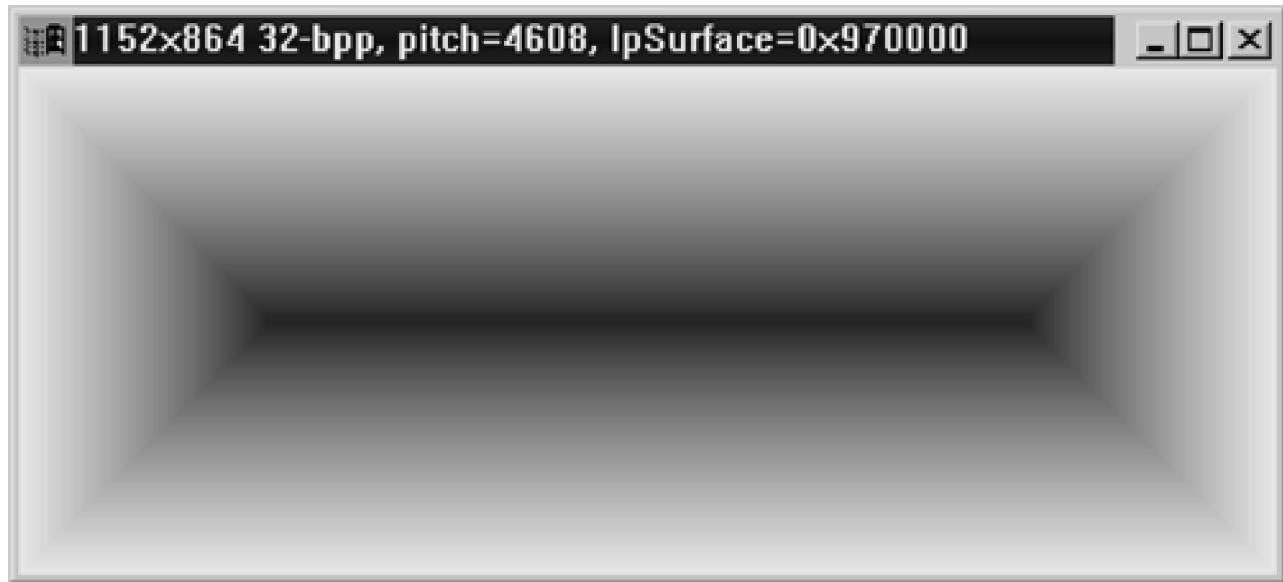
```
        return DefWindowProc(hWnd, uMsg, wParam, lParam);
    }
}

void GetWndClassEx(WNDCLASSEX & wc)
{
    KWindow::GetWndClassEx(wc);
    wc.style |= (CS_HREDRAW | CS_VREDRAW);
}
};
```

The KDDWin class is derived from the KWindow class and the KDirectDraw class. It initializes DirectDraw support in the WM_CREATE message handle by calling the KDirectDraw::SetupDirectDraw method. Its WM_PAINT message handling uses KDDSurface::FillColor to fill the window client area with rectangles with colors gradually changing from yellow to blue. This illustrates hardware-accelerated drawing using the IDirectDrawSurface7::Blt method. Its WM_NCPAINT message handling uses the PixelFillRect routine to draw the title bar background, and then uses the GDI text function to display the primary surface format as a text string on the title bar. Overall, this simple class shows how easy it is to use the KDirectDraw and KDDSurface class to add DirectDraw support to your normal window program, and the three ways of drawing on a DirectDraw surface.

[Figure 18-2](#) shows a screen from the DDBASIC sample program on the CD, which uses the KDDWin class to create a main window.

Figure 18-2. Simple DirectDraw drawing in windowed mode.



[< BACK](#) [NEXT >](#)

18.3 BUILDING A DIRECTDRAW GRAPHICS LIBRARY

As we have seen in the last section, DirectDraw provides only two hardware-accelerated drawing methods, Blt and BltFast, which are able to handle solid color fill and copying between DirectDraw surfaces. For more complicated drawing requests, you have to either break them down to Blt and BltFast, use direct pixel access on a locked surface, or use GDI.

DirectDraw is well suited for porting DOS game programs to Windows, which normally have extensive graphics libraries that need nothing much more than direct pixel access. For anyone not having a graphics library to port from, you have to build your own library or go back to GDI.

In this section, we will discuss how to build a basic DirectDraw graphics library that can handle pixel, line, area fill, text, bitmap, and transparent bitmaps.

Pixel Drawing

When a DirectDraw surface is locked, a pointer to its frame buffer is returned. To draw a pixel is simply to find the right position in the frame buffer and copy a number of bytes. Locking and unlocking DirectDraw surface are expensive system calls we cannot afford to do for each single-pixel drawing. So the graphics library design should allow the application to lock the surface once and draw lots of pixels before unlocking it.

Here is a generic class that handles DirectDraw surface locking and pixel access.

```
class KLockedSurface
{
public:
    BYTE * pSurface;
    int   pitch;
    int   bpp;

    bool Initialize(KDDSurface & surface)
    {
        pSurface = surface.LockSurface(NULL);
        pitch   = surface.GetPitch();
        bpp     = surface.GetSurfaceDesc()->ddpfPixelFormat.dwRGBBitCount;

        return pSurface!=NULL;
    }

    BYTE & ByteAt(int x, int y)
    {
        BYTE * pPixel = (BYTE *) (pSurface + pitch * y);
```

```
    return pPixel[x];
}
WORD & WordAt(int x, int y)
{
    WORD * pPixel = (WORD *) (pSurface + pitch * y);

    return pPixel[x];
}

RGBTRIPLE & RGBTripleAt(int x, int y)
{
    RGBTRIPLE * pPixel = (RGBTRIPLE *) (pSurface + pitch * y);

    return pPixel[x];
}

DWORD & DWordAt(int x, int y)
{
    DWORD * pPixel = (DWORD *) (pSurface + pitch * y);

    return pPixel[x];
}

BOOL SetPixel(int x, int y, DWORD color)
{
    switch ( bpp )
    {
        case 8:   ByteAt(x, y) = (BYTE) color; break;
        case 15:
        case 16:  WordAt(x, y) = (WORD) color; break;
        case 24:  RGBTripleAt(x, y) = * (RGBTRIPLE *) & color; break;
        case 32:  DWordAt(x, y) = (DWORD) color; break;
        default: return FALSE;
    }

    return TRUE;
}

DWORD GetPixel(int x, int y); // omitted
void Line(int x0, int y0, int x1, int y1, DWORD color);
};
```

The KLockedSurface class uses three member variables to represent a locked surface: a pointer to the frame buffer, pitch, and pixel depth. The Initialize method locks a DirectDraw surface to set these member variables to the proper value. The four in-line methods, ByteAt, WordAt, RGBTripleAt, and DWordAt, turn the frame buffer into a randomly accessible two-dimensional array. They can be used either to read or to write an 8-bpp, 16-bpp, 24-bpp, or 32-bpp pixel on the surface. The KLockedSurface ::SetPixel method is a generic method for drawing a pixel on the surface, similar to GDI's function by the same name. Likewise, the KLockedSurface::GetPixel is a generic method for reading a pixel.

Here is an example of using the KLockedSurface class to implement the SetPixel method in the KDDSurface class.

```
BOOL KDDSurface::SetPixel(int x, int y, DWORD color)
{
    KLockedSurface frame;

    if ( frame.Initialize(* this) )
    {
        frame.SetPixel(x, y, color);
        Unlock();
        return TRUE;
    }
    else
        return FALSE;
}
```

To achieve high performance, multiple-pixel drawing should be merged to use a single surface locking. The KLockedSurface class can also be used to implement logical or non-logical raster operations when drawing a pixel. Here are few examples:

```
ByteAt(x, y) |= (BYTE) color; // R2.MergePen
WordAt(x, y) ^= (DWORD) color; // R2.XorPen
DWordAt(x, y) = 0;           // R2.Black
ByteAt(x, y) = ((ByteAt(x-1, y) + ByteAt(x, y-1),
                 ByteAt(x+1, y) + ByteAt(x, y+1)) / 4; // Blurring
```

Note that there is no clipping or boundary checking in the KLockedSurface class. The application is assumed to send preclipped coordinates. Here is a random pixel drawing routine.

```
void PixelDemo(void)
{
    KLockedSurface frame;

    if ( !frame.Initialize(m_primary) )
        return;
    for ( int i=0; i<4096; i++ )
        frame.SetPixel(
            m_rcDest.left + rand() % ( m_rcDest.right - m_rcDest.left ),
            m_rcDest.top + rand() % ( m_rcDest.bottom - m_rcDest.top ),
            m_primary.ColorMatch(rand()%256, rand()%256, rand()%256));

    m_primary.Unlock();
}
```

Line Drawing

All curves can be broken down into lines, which should be supported as basic primitives in any graphics library. A widely used algorithm for digital line drawing is the Bresenham algorithm, published in 1965. The Bresenham algorithm treats approximating a line using pixels on a finite grid as an iterative process controlled by an error value. The error value determines whether both x- and y-coordinates, or just one of them, should be updated when moving to the next pixel. When moving to the next pixel, the error value is updated to represent the difference between the real line and the pixel approximation. The nice feature about the Bresenham algorithm is that it involves no expensive multiplication and division, if you do not count doubling a value as multiplication.

Here is an implementation of the Bresenham algorithm for the KLockedSurface class.

```
void KLockedSurface::Line(int x0, int y0, int x1, int y1, DWORD color)
{
    int bps      = (bpp+7) / 8;           // bytes-per-pixel
    BYTE * pPixel = pSurface + pitch * y0 + bps * x0; // first pixel

    int error;                // error
    int d_pixel_pos, d_error_pos; // adjustments when error>=0
    int d_pixel_neg, d_error_neg; // adjustments when error<0
    int dots;                 // number of dots to draw

    {
        int dx, dy, inc_x, inc_y;

        if ( x1 > x0 )
        {   dx = x1 - x0; inc_x = bps; }
        else
        {   dx = x0 - x1; inc_x = -bps; }
        if ( y1 > y0 )
        {   dy = y1 - y0; inc_y = pitch; }
        else
        {   dy = y0 - y1; inc_y = -pitch; }

        d_pixel_pos = inc_x + inc_y; // move x and y
        d_error_pos = (dy - dx) * 2;

        if ( d_error_pos < 0 ) // x dominant
        {
            dots      = dx;
            error     = dy*2 - dx;
            d_pixel_neg = inc_x; // move x only
            d_error_neg = dy * 2;
        }
        else
        {
```

```
dots      = dy;
error     = dx*2 - dy;
d_error_pos = - d_error_pos;
d_pixel_neg = inc_y;      // move y only
d_error_neg = dx * 2;
}

}

switch ( bps )
{
    case 1: // 8-bpp pixel loop omitted, refer to CD
    case 2: // 16-bpp pixel loop omitted, refer to CD
    case 3: // 24-bpp pixel loop omitted, refer to CD
        break;

    case 4:
        for (; dots>=0; dots--)      // 32-bpp pixel loop
        {
            * (DWORD *) pPixel = color; // draw 32-bpp pixel

            if ( error>=0 )
                { pPixel += d_pixel_pos; error += d_error_pos; }
            else
                { pPixel += d_pixel_neg; error += d_error_neg; }
        }
        break;
}
}
```

The KLockedSurface::Line method is divided into two sections, an initial setup phase and a pixel drawing loop. The initial setup phase sets up first pixel address, number of pixels to draw, initial error value, pixel address adjustment values, and error value adjustment values. The pixel drawing loop supports all common surface pixel formats. For each pixel format, it loops through setting the pixel value and moving to the next pixel under the control of the error value. Note that pixel address calculation is in-lined and updated incrementally for better performance.

Almost all early graphics libraries for DOS game programs were written in assembly code for better performance. There is enough literature out there still tempting you to use assembly code for graphics drawing primitives. Unless you have examined assembly code generated by a modern compiler and are still pretty sure you can do a better job, do not use unoptimized assembly code to slow down your program.

If you want to see what the compiler can do, ask it to generate listing files with both C/C++ statements and assembly code. Here is how the VC 6.0 compiler can do the 32-bpp pixel drawing loop:

```
//      eax : color
//      ebx : dots
//      ecx : error
//      edx : pPixel
//      esi : d_error_pos
```

```
//      edi : d_error_neg
//      ebp : d_pixel_neg

        test  ebx, ebx      if ( dots<0 ) goto _finish;
        jl    _finish
        mov   eax, color     eax = color
        inc   ebx            dots ++;

_repeat: test  ecx, ecx
        mov   [edx], eax     * (DWORD *) pPixel = color;
        jl    _elsepart      if ( error < 0 ) goto _elsepart
        add   edx, d_pixel_pos  pPixel += d_pixel_pos
        add   ecx, esi       error += d_error_pos
        jmp   _next          goto _next
_elsepart: add   edx, ebp      pPixel += d_pixel_neg
        add   ecx, edi       error += d_error_neg
_next:   dec   ebx            dots --;
        jne   _repeat        if ( dots!=0 ) goto _repeat
_FINISH:
```

For an Intel CPU running in 32-bit mode, there are seven general-purpose registers the compiler can use. For this tight pixel loop, which is critical to line drawing performance, the compiler is smart enough to use all seven registers. The only important value not stored in the register is d_pixel_pos. For the pixel drawing loop, the only nonregister access is for d_pixel_pos and writing pixel to the frame buffer. The compiler separates one testing instruction and its corresponding jump instruction to make sure both instruction pipelines in the CPU can be utilized.

The KLockedSurface::Line method can be extended to draw styled lines, lines with raster operation, or even alpha blended lines. But thicker lines should be converted to area fills. It assumes the coordinates are preclipped, too.

Here is an example of how to use the line drawing method.

```
void LineDemo(KDDSurface & surface, int x, int y, int Radius)
{
    const int N      = 19;
    const double theta = 3.1415926 * 2 / N;

    const COLORREF color[10] = {
        RGB(0, 0, 0), RGB(255,0,0), RGB(0,255,0), RGB(0,0, 255),
        RGB(255,255,0), RGB(0, 255, 255), RGB(255, 255, 0),
        RGB(127, 255, 0), RGB(0, 127, 255), RGB(255, 0, 127)
    };

    DWORD dwColor[10];
    for (int i=0; i<10; i++)
        dwColor[i] = surface.ColorMatch(GetRValue(color[i]),
                                       GetGValue(color[i]), GetBValue(color[i]));

    KLockedSurface frame;
```

```
if ( frame.Initialize(m_primary) )
{
    for (int p=0; p<N; p++)
        for (int q=0; q<p; q++)
            frame.Line( (int)(x + Radius * sin(p * theta)),
                       (int)(y + Radius * cos(p * theta)),
                       (int)(x + Radius * sin(q * theta)),
                       (int)(y + Radius * cos(q * theta)),
                       dwColor[min(p-q, N-p+q)]);

    m_primary.Unlock();
}
}
```

Area Fill

DirectDraw supports filling a rectangular area with solid color. A nonrectangular area can either be broken down to a sequence of rectangular areas or converted to a clipping region.

Here is the implementation for the KDDSurface::FillRgn method, which fills an arbitrary region with a solid color.

```
RGNDATA * GetClipRegionData(HRGN hRgn)
{
    DWORD dwSize = GetRegionData(hRgn, 0, NULL);

    RGNDATA * pRgnData = (RGNDATA *) new BYTE[dwSize];

    if ( pRgnData )
        GetRegionData(hRgn, dwSize, pRgnData);

    return pRgnData;
}

BOOL KDDSurface::FillRgn(HRGN hRgn, DWORD color)
{
    RGNDATA * pRegion = GetClipRegionData(hRgn);

    if ( pRegion==NULL )
        return FALSE;

    const RECT * pRect = (const RECT *) pRegion->Buffer;

    for (unsigned i=0; i<pRegion->rdh.nCount; i++)
    {
        FillColor(pRect->left, pRect->top, pRect->right,
                  pRect->bottom, color);
        pRect++;
    }
}
```

```
}

delete [] (BYTE *) pRegion;

return TRUE;
}
```

GDI provides a powerful set of region functions that can handle simple geometric shapes, combinations of shapes, closed paths, and even text outlines. It makes sense for DirectDraw programs to take advantage of GDI's region support. If performance is an issue, region data can be precalculated and cached.

The KDDSurface::FillRgn method accepts a GDI region object handle; it uses the GDI function GetRegionData to decompose a region into a sequence of rectangles (RGNDATA structure) and then calls IDirectDrawSurface7::Blt method to perform a solid color fill on each of them. The current DirectDraw clipper on the surface is still effective.

An alternative implementation could convert the current DirectDraw clipper's clip list to a GDI region, combine with the region to draw, create a new clip list, and use the Blt method once to draw the whole thing. The trouble with the second approach is that you have to create a second DirectDraw clipper and manage clipper switching on a surface.

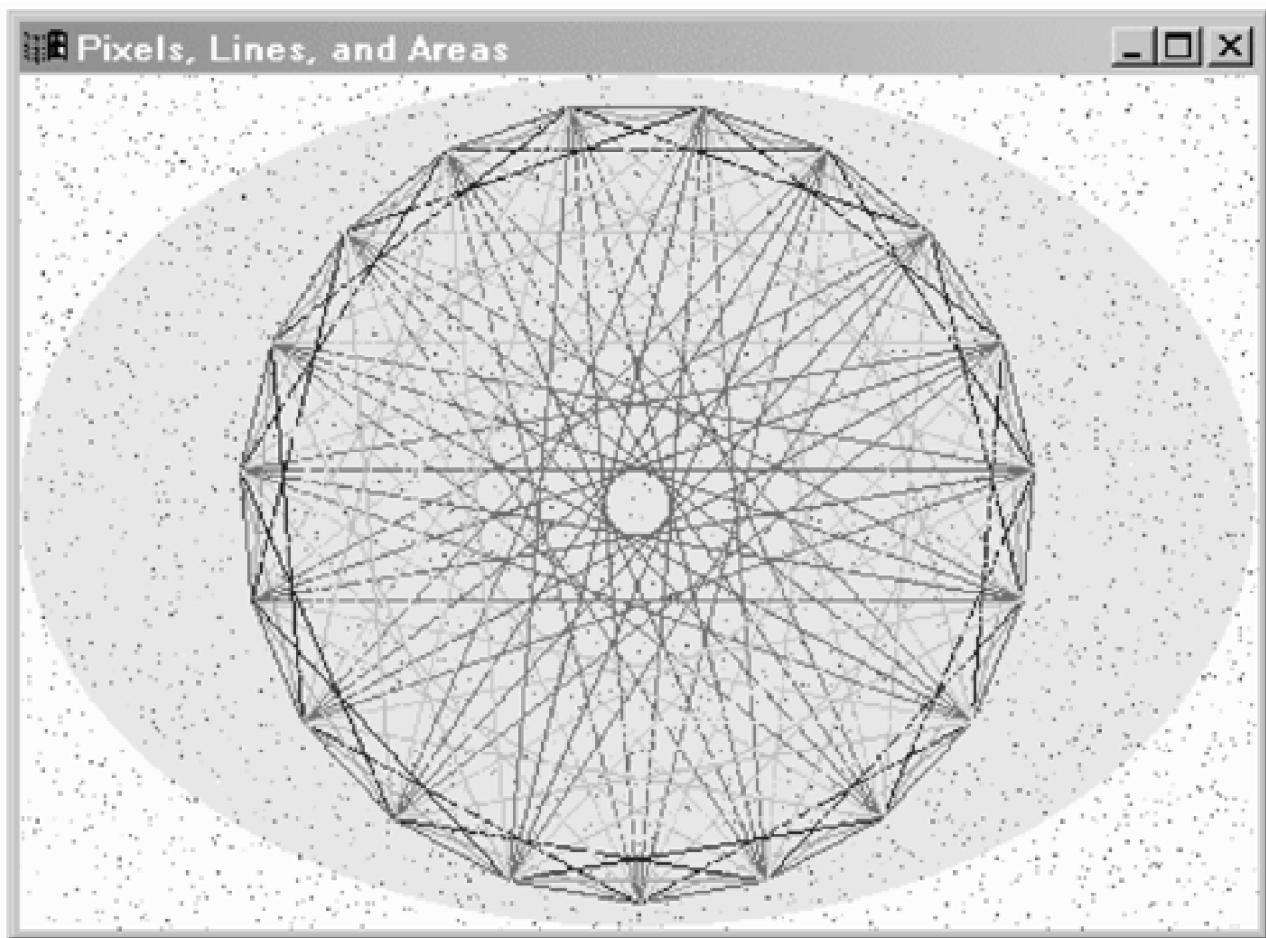
Here is an example of how to draw a solid ellipse on a DirectDraw surface.

```
void RegionDemo(void)
{
    HRGN hRgn = CreateEllipticRgnIndirect(& m_rcDest);

    if ( hRgn )
    {
        m_primary.FillRgn(hRgn, m_primary.ColorMatch(0xFF, 0xFF, 0));
        DeleteObject(hRgn);
    }
}
```

[Figure 18-3](#) shows a DirectDraw MDI child window with pixels, lines, and a solid-filled ellipse.

Figure 18-3. Pixels, lines, and area fills on a DirectDraw surface.



Clipping

DirectDraw surface supports clipping through DirectDraw clipper objects, which are created using the `IDirectDraw7::CreateClipper` method. There are two classes of DirectDraw clipper objects: those associated with a window and those created with a clip list.

When a clipper is associated with a window using the `IDirectDrawClipper::SetHWnd` method, some magic in the operating system makes sure the clipper object is always synchronized with that particular window's update region. So drawing on a DirectDraw surface with the clipper attached could confine itself to the visible client area. So far, we have seen how such clippers can help DirectDraw drawing using the `Blt` method behave properly in windowed mode.

An application can also explicitly manage a DirectDraw clipper object by setting its clip list, which is none other than GDI's `RGNDATA` structure. DirectX documentation may assume that DirectX programmers are well versed in GDI programming, so there is not much explanation on how to use a clip list properly.

Here are a routine and a class, linking the GDI region object with the DirectDraw clipper.

```
BOOL SetClipRegion(IDirectDrawClipper * pClipper, HRGN hRgn)
{
    RGNDATA * pRgnData = GetClipRegionData(hRgn);

    if ( pRgnData==NULL )
```

```
return FALSE;

HRESULT hr = pClipper->SetClipList(pRgnData, 0);

delete (BYTE *) pRgnData;

return SUCCEEDED(hr);
}

class KRgnClipper
{
    IDirectDrawClipper * m_pNew;
    IDirectDrawClipper * m_pOld;
    IDirectDrawSurface7 * m_pSrf;

public:
    KRgnClipper(IDirectDraw7 * pDD, IDirectDrawSurface7 * pSrf, HRGN hRgn)
    {
        pDD->CreateClipper(0, & m_pNew, NULL); // create new clipper

        SetClipRegion(m_pNew, hRgn);           // set clip list from region

        m_pSrf = pSrf;
        pSrf->GetClipper(& m_pOld);         // get old clipper
        pSrf->SetClipper(m_pNew);           // replace with new clipper
    }

    ~KRgnClipper()
    {
        m_pSrf->SetClipper(m_pOld);         // restore old clipper
        m_pOld->Release();                 // release old clipper
        m_pNew->Release();                 // release new clipper
    }
};
```

The SetClipper routine sets the clip list of a DirectDraw clipper from a GDI region object. It uses the GDI function GetRegionData to retrieve the clip list from a region handle. Recall that because region data can be of arbitrary size, you have to query for its size first, allocate the space, and then retrieve the real data.

The KRgnClipper class manages switching the clipper object attached to a DirectDraw surface to a new clipper created from a GDI region object. Its constructor creates a new clipper, sets its clip list from a GDI region handle, and replaces the current clipper object associated with a surface. Any drawing after that will use the new clipper. The destructor restores the original clipper and releases the resource.

Here is a routine that illustrates how to use KRgnClipper class for filling an area.

```
void ClipDemo(void)
{
    HRGN hUpdate = CreateRectRgn(0, 0, 1, 1);
```

```
GetUpdateRgn(m_hWnd, hUpdate, FALSE);           // update region
OffsetRgn(hUpdate, m_rcDest.left, m_rcDest.top); // screen coordinate
HRGN hEllipse = CreateEllipticRgn(m_rcDest.left-20, m_rcDest.top-20,
                                  m_rcDest.right+20, m_rcDest.bottom+20);
CombineRgn(hEllipse, hEllipse, hUpdate, RGN_AND);
DeleteObject(hUpdate);

KRgnClipper clipper(m_pDD, m_primary, hEllipse);

DeleteObject(hEllipse);

m_primary.FillColor(m_rcDest.left-20, m_rcDest.top-20,
                     m_rcDest.right+20, m_rcDest.bottom+20,
                     m_primary.ColorMatch(0, 0, 0xFF));
}
```

The ClipDemo routine first queries the current window's update region; then it translates the update region from the client area coordinate to the screen coordinate, as required by DirectDraw. The routine then creates an elliptic region larger than the window's client area, which is then combined with the update region and used to create a new clipper. The KDDSurface::FillColor routine is used to fill an area larger than the window's client area. But because of the clipping, the drawing is clipped to both the ellipse and the update region.

Off-Screen Surface

The easy way to draw a bitmap on a DirectDraw surface is by using GDI functions, as shown in the KDDSurface::DrawBitmap method. Although you can copy the bitmap bits to a locked DirectDraw surface, handling decompression, different bitmap formats, stretching, palette, and pixel format conversion can be quite some code to write. But the performance and feature benefits of DirectDraw are lost in doing that.

The proper way of drawing bitmaps in DirectDraw is to take advantage of both GDI and DirectDraw by loading a bitmap to an off-screen surface first, using GDI, and then drawing to the main drawing surface using DirectDraw.

Here is the KOffScreenSurface class, which derives from the KDDSurface class to handle off-screen surface and bitmap loading to an off-screen surface.

```
typedef enum
{
    mem_default,
    mem_system,
    mem_nonlocalvideo,
    mem_localvideo
};

class KOffScreenSurface : public KDDSurface
{
public:
    HRESULT CreateOffScreenSurface(IDirectDraw7 * pDD, int width,
```

```
    int height, int mem=mem_default);
HRESULT CreateOffScreenSurfaceBpp(IDirectDraw7 * pDD, int width,
    int height, int bpp, int mem=mem_default);

HRESULT CreateBitmapSurface(IDirectDraw7 * pDD,
    const BITMAPINFO * pDIB, int mem=mem_default);
HRESULT CreateBitmapSurface(IDirectDraw7 * pDD,
    const TCHAR * pFileName, int mem=mem_default);
}

const DWORD MEMFLAGS[] =
{
    0,
    DDSCAPS_SYSTEMMEMORY,
    DDSCAPS_NONLOCALVIDMEM | DDSCAPS_VIDEOMEMORY,
    DDSCAPS_LOCALVIDMEM | DDSCAPS_VIDEOMEMORY
};

HRESULT KOffScreenSurface::CreateOffScreenSurface(IDirectDraw7 * pDD,
    int width, int height, int mem)
{
    m_ddsd.dwFlags      = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH;
    m_ddsd.ddsCaps.dwCaps = DDSCAPS_OFFSCREENPLAIN | DDSCAPS_3DDEVICE |
        MEMFLAGS[mem];
    m_ddsd.dwWidth      = width;
    m_ddsd.dwHeight      = height;

    return pDD->CreateSurface(&m_ddsd, &m_pSurface, NULL);
}

HRESULT KOffScreenSurface::CreateBitmapSurface(IDirectDraw7 * pDD,
    const BITMAPINFO * pDIB, int mem)
{
    if ( pDIB==NULL )
        return E_FAIL;
    HRESULT hr = CreateOffScreenSurface(pDD, pDIB->bmiHeader.biWidth,
        abs(pDIB->bmiHeader.biHeight), mem);

    if ( FAILED(hr) )
        return hr;

    return DrawBitmap(pDIB, 0, 0, m_ddsd.dwWidth, m_ddsd.dwHeight);
}
```

The `CreateOffScreenSurface` method creates an off-screen DirectDraw surface, which is a surface stored in memory but not shown on your display monitor. The memory used for an off-screen surface can be from system memory, nonlocal video memory, or local video memory, as determined by flags in the `ddsCaps.dwCaps` field. System memory is plentiful, as it is only limited by the size of system page file. Nonlocal video memory is the memory managed by the AGP (Advanced Graphics Port), which offers a faster way to send data to video memory. Nonlocal

video memory is scarcer than system memory, but faster to display. Local video memory is the most expensive and scarcest memory, but it is fastest to display. Note that if you are accessing surface pixels directly from an application, local video memory is the slowest because it's most distant from the CPU. The last parameter of CreateOffScreenSurface controls where surface memory is allocated.

Compared with creating a primary surface, creating an off-screen surface requires specifying the exact size of the surface. The CreateOffScreenSurfaceBpp method, whose implementation is not shown here, allows the creation of the off-screen surface with specific pixel format.

The first CreateBitmapSurface method accepts a packed DIB as input. It creates an off-screen surface based on the size of the bitmap, and then copies the bitmap to the surface using GDI. The second CreateBitmapSurface method, whose implementation is omitted here, loads a bitmap from an external file to create a surface. Both methods use DIB, which uses less system resource than the widely used DDB/DIB section in DirectX literature.

Once a bitmap is loaded to an off-screen surface, the IDirectDrawSurface7::Blt method can be used to copy the bitmap to another surface. The KDDSurface class provides two BitBlt methods, which are simple wrapper functions around the Blt method to make the call more like GDI calls. Here is one of them:

```
HRESULT KDDSurface::BitBlt(int x, int y, int w, int h,  
    IDirectDrawSurface7 * pSrc, DWORD flag)  
{  
    RECT rc = { x, y, x+w, y+h };  
  
    return m_pSurface->Blt(& rc, pSrc, NULL, flag, NULL);  
}
```

Transparency through Color Keying

The IDirectDrawSurface7::Blt method supports transparent bitmap drawing using color keying. Color key is an attribute of a DirectDraw surface, which can be specified on either the source surface or destination surface. The DDCOLORKEY structure is used to specify a color key, which can be either a single color or a color range.

Here is the SetSourceColorKey method in the KDDSurface class, which sets a source color key using a single physical color

```
HRESULT KDDSurface::SetSourceColorKey(DWORD color)  
{  
    DDCOLORKEY key;  
  
    key.dwColorSpaceLowValue = color;  
    key.dwColorSpaceHighValue = color;  
  
    return m_pSurface->SetColorKey(DDCKEY_SRCBLT, & key);  
}
```

To draw an off-screen bitmap surface with a source color key, the same Blt method can be called with a DDBLT_KEYSRC flag. Only pixels not matching the source code key will be drawn. DirectDraw also supports

destination color keying.

Font and Text

As a simple and low-level API designed for performance, DirectX does not have built-in font or text support. Even OpenGL implementation on Windows depends on window extensions to provide fonts.

When using GDI to provide fonts and to draw text to a DirectDraw surface, using a GDI device context is an option. But games and applications that want high performance can't afford to use the slow GDI path in time-critical drawing. Another option commonly used in games is pre-generating a bitmap with a set of glyphs used in the game. These bitmaps may be called *font bitmaps*. Fonts are converted to bitmaps, and text drawing is converted to bitmap copying. The problem with the second approach is inflexibility, as fonts are limited and text size is predetermined.

A better approach once again is combining the GDI approach and the font-bitmap approach. The idea is to dynamically generate font bitmaps according to the font typeface name and text point size, and then to use DirectDraw methods to display text from the font bitmaps. Font bitmaps are generated only when an application is first loaded, which brings flexibility in choosing typeface and text size, while having limited impact on performance. Font bitmaps can even be cached as bitmap files on the disk. They can be loaded into off-screen surfaces, perhaps in local video memory for maximum performance using the hardware-accelerated Blt method.

Listing 18-4 shows the KDDFont class, which supports dynamically generated font bitmaps in off-screen surface.

Listing 18-4 KDDFont Class for dynamic font bitmaps and text drawing

```

template <int MaxChar>
class KDDFont : public KOffScreenSurface
{
    int    m_offset [MaxChar]; // A width
    int    m_advance[MaxChar]; // A + B + C
    int    m_pos   [MaxChar]; // horizontal position
    int    m_width [MaxChar]; // - min(A, 0) + B - min(C,0)

    unsigned m_firstchar;
    int     m_nChar;

public:
    HRESULT CreateFont(IDirectDraw7 * pDD, const LOGFONT & lf,
                       unsigned firstchar, unsigned lastchar, COLORREF crColor);
    int     TextOut(IDirectDrawSurface7 * pSurface, int x, int y,
                   const TCHAR * mess, int nChar=0);
};

template <int MaxChar>
HRESULT KDDFont<MaxChar>::CreateFont(IDirectDraw7 * pDD,
                                       const LOGFONT & lf, unsigned firstchar,
                                       unsigned lastchar, COLORREF crColor)
{

```

```
m_firstchar = firstchar;
m_nChar    = lastchar - firstchar + 1;
if ( m_nChar > MaxChar )
    return E_INVALIDARG;

HFONT hFont = CreateFontIndirect(&lf);

if ( hFont==NULL )
    return E_INVALIDARG;

HRESULT hr;
ABC    abc[MaxChar];

int height;
{
    HDC hDC = ::GetDC(NULL);

    if ( hDC )
    {
        HGDIOBJ hOld = SelectObject(hDC, hFont);

        TEXTMETRIC tm;
        GetTextMetrics(hDC, & tm);
        height = tm.tmHeight;

        if ( GetCharABCWidths(hDC, firstchar, lastchar, abc) )
            hr = S_OK;
        else
            hr = E_INVALIDARG;

        SelectObject(hDC, hOld);
        ::ReleaseDC(NULL, hDC);
    }
}

if ( SUCCEEDED(hr) )
{
    int width = 0;

    for (int i=0; i<m_nChar; i++)
    {
        m_offset[i] = abc[i].abcA;
        m_width[i]  = - min(abc[i].abcA, 0) + abc[i].abcB -
                      min(abc[i].abcC, 0);
        m_advance[i] = abc[i].abcA + abc[i].abcB + abc[i].abcC;
        width += m_width[i];
    }

    hr = CreateOffScreenSurface(pDD, width, height);
```

```
if ( SUCCEEDED(hr) )
{
    GetDC();
    int x = 0;
    PatBlt(m_hDC, 0, 0, GetWidth(), GetHeight(), BLACKNESS);

    SetBkMode(m_hDC, TRANSPARENT);
    SetTextColor(m_hDC, crColor); // white foreground
    HGDIOBJ hOld = SelectObject(m_hDC, hFont);
    SetTextAlign(m_hDC, TA_TOP | TA_LEFT);

    for (int i=0; i<m_nChar; i++)
    {
        TCHAR ch = firstchar + i;
        m_pos[i] = x;
        ::TextOut(m_hDC, x-m_offset[i], 0, & ch, 1);
        x += m_width[i];
    }

    SelectObject(m_hDC, hOld);
    ReleaseDC();
    SetSourceColorKey(0); // black as source color key
}

}

DeleteObject(hFont);
return hr;
};

template<int MaxChar>
int KDDFont<MaxChar>::TextOut(IDirectDrawSurface7 * pDest, int x, int y,
    const TCHAR * mess, int nChar)
{
    if ( nChar<=0 )
        nChar = _tcslen(mess);

    for (int i=0; i<nChar; i++)
    {
        int ch = mess[i] - m_firstchar;
        if ( (ch<0) || (ch>m_nChar) )
            ch = 0;

        RECT dst = { x + m_offset[ch],           y,
                    x + m_offset[ch] + m_width[ch], y + GetHeight() };
        RECT src = { m_pos[ch], 0, m_pos[ch] + m_width[ch], GetHeight() };

        pDest->Blt(& dst, m_pSurface, & src, DDBLT_KEYSRC, NULL);
        x += m_advance[ch];
    }
}
```

```
}
```

```
    return x;  
}
```

The KDDFont class is designed to support generic fonts supported by GDI. To be more specific, it supports monospace fonts, variable-pitch fonts, and text metrics using ABC widths. Being a generic font class, the KDDFont adds extra fields to the KOff ScreenSurface class. The m_offset array keeps the A-width for each glyph, the m_advance array specifies the amount to move for the next character, the m_pos array stores the horizontal location of each glyph on the surface, and the m_width array keeps glyph widths. The class converts a range of characters in a font to a font bitmap, for which it uses two member variables to keep the first and last character for the character range, and a template parameter for a maximum number of characters.

The KDDFont::CreateFont method initializes a font bitmap, given an IDirectDraw7 object pointer, a GDI LOGFONT structure, character range, and text foreground color. It uses the LOGFONT structure to create a GDI logical font, and it queries for character ABC widths to calculate values for the four array members in the class. From these calculations, the height and width for the font bitmap are known; then a DirectDraw off-screen surface is created. GDI is used to clear the bitmap and draw each glyph into the bitmap. Note that you cannot convert the characters within the range to a string and draw them in a single call because the characters may overlap horizontally with each other. Instead, each character is drawn individually according to its calculated position in the bitmap. The characters are drawn using black as the background color, and black is also the source color key for the font surface.

The KDDFont::TextOut method uses a font bitmap surface to draw a character string to a DirectDraw surface. The four arrays kept in KDDFont are used to locate character glyphs and align them properly to form a line of text. The IDirectDrawSurface7::Blt method is used to draw each character transparently using the source color key on the font surface.

The TextOut method draws text transparently using the color specified when the font surface is created. You can also add methods to change the text color or to add special blending effects when displaying text. A font surface can also be saved to a bitmap, and thereafter GDI is not needed in text drawing. Another idea is adding extra space between glyphs to implement special effects.

Not Quite a Game

Bitmaps and transparent bitmaps are the main ingredients of a game program. Bitmaps form the background of a game scene. Transparent bitmaps, which are called sprites in games, form objects flying around in the scene. They bump into each other to generate actions. The keyboard, mouse, or other input devices are used to control your sprite to play a game.

Here is a DirectDraw pseudogame, which illustrates how to use bitmap, sprite, and text in DirectDraw surface.

```
class KSpriteDemo : public KMDIChild, public KDIRECTDRAW  
{  
    KOFFSCREENSURFACE m_background;  
    POINT      m_backpos;  
    KOFFSCREENSURFACE m_sprite;  
    POINT      m_spritepos;  
    KDDFONT<128>  m_font;
```

```
HINSTANCE      m_hInst;
HWND          m_hTop;

void OnDraw(void);
void OnCreate(void);

void MoveSprite(int dx, int dy)
{
    m_spritepos.x += dx * 5;
    m_spritepos.y += dy * 5;
    OnDraw();
}

LRESULT WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    switch( uMsg )
    {
        case WM_PAINT:
            OnDraw();
            ValidateRect(hWnd, NULL);
            return 0;

        case WM_KEYDOWN:
            switch ( wParam )
            {
                case VK_HOME : MoveSprite(-1, -1); break;
                case VK_UP   : MoveSprite( 0, -1); break;
                case VK_PRIOR: MoveSprite(+1, -1); break;

                case VK_LEFT : MoveSprite(-1,  0); break;
                case VK_RIGHT: MoveSprite( 1,  0); break;

                case VK_END  : MoveSprite(-1,  1); break;
                case VK_DOWN : MoveSprite( 0,  1); break;
                case VK_NEXT : MoveSprite( 1,  1); break;
            }
            return 0;

        case WM_CREATE:
            m_hWnd = hWnd;
            OnCreate();
            // fall through
        default:
            return KMDIChild::WndProc(hWnd, uMsg, wParam, lParam);
    }
}

public:
```

```
KSpriteDemo(HMODULE hModule, HWND hTop)
{
    m_spritepos.x = 0;
    m_spritepos.y = 0;
    m_backpos.x = 0;
    m_backpos.y = 0;

    m_hInst = hModule;
    m_hTop = hTop;
}
};
```

The KSpriteDemo class derives from the KMDIChild window for MDI child window support, and from the KDirectDraw class for DirectDraw support. The m_background and m_backpos member variables are for managing a background bitmap loaded as a DirectDraw off-screen surface. The background bitmap can be bigger than the window size to allow for a bigger scrollable scene. The m_sprite and m_spritepos member variables are for a flying plane on the background scene. The m_font member variable is an instance of the KDDFont class.

The OnCreate method handles background bitmap, sprite, and font initialization, the OnDraw method handles the drawing of the scene, and the MoveSprite method handles the keyboard control of the flying plane. Here is the implementation of the OnCreate method.

```
void KSpriteDemo::OnCreate(void)
{
    if ( SUCCEEDED(SetupDirectDraw(m_hTop, m_hWnd, false)) )
    {
        BITMAPINFO * pDIB = LoadBMP(m_hInst, MAKEINTRESOURCE(IDB_NIGHT));

        if ( pDIB )
            m_background.CreateBitmapSurface(m_pDD, pDIB);

        pDIB = LoadBMP(m_hInst, MAKEINTRESOURCE(IDB_PLANE));

        if ( pDIB )
        {
            m_sprite.CreateBitmapSurface(m_pDD, pDIB);
            m_sprite.SetSourceColorKey(0); // black
        }

        LOGFONT lf;
        memset(&lf, 0, sizeof(lf));
        lf.lfHeight = - 36;
        lf.lfWeight = FW_BOLD;
        lf.lfItalic = TRUE;
        lf.lfQuality = ANTIALIASED_QUALITY;
        _tcscpy(lf.lfFaceName, "Times New Roman");
    }
}
```

```
m_font.CreateFont(m_pDD, lf, ' ', 0x7F, RGB(0xFF, 0xFF, 0));
}

else
{
    MessageBox(NULL, _T("Unable to Initialize DirectDraw"),
    _T("KSpriteDemo"), MB_OK);
    CloseWindow(m_hWnd);
}
}
```

The OnCreate method calls the KDirectDraw::SetupDirectDraw method to initialize a DirectDraw environment in windowed mode. It then loads the background image from resource as a packed DIB and initializes the background bitmap surface. The sprite bitmap is also loaded from resource and set to use black as the source color key. The font bitmap surface is initialized from a 36-pixel-high, antialiased italic LOGFONT structure.

Here is the implementation of the OnDraw method.

```
void KSpriteDemo::OnDraw(void)
{
    SetClientRect(m_hWnd);

    int dy = (m_rcDest.bottom - m_rcDest.top - m_background.GetHeight()) / 2;

    // draw area not covered by the background image
    if (dy > 0)
    {
        DWORD color = m_primary.ColorMatch(0x80, 0x40, 0);

        m_primary.FillColor(m_rcDest.left, m_rcDest.top,
        m_rcDest.right, m_rcDest.top + dy, color); // dark warm color

        color = m_primary.ColorMatch(0, 0x40, 0x80);
        m_primary.FillColor(m_rcDest.left, m_rcDest.bottom - dy - 1,
        m_rcDest.right, m_rcDest.bottom, color); // dark cool color
    }
    else
        dy = 0;

    // draw background image

    // if right edge of sprite is offscreen, move background to the left
    while ( (m_spritepos.x + m_sprite.GetWidth() + m_backpos.x) >
            (m_rcDest.right - m_rcDest.left) )
        m_backpos.x -= 100;

    // if left edge of sprite is offscreen, move background to the right
    while ( (m_spritepos.x + m_backpos.x) < 0 )
        m_backpos.x += 100;
```

```
// make sure background position in within the right range  
m_backpos.x = max(m_backpos.x, m_rcDest.right - m_background.GetWidth()  
    - m_rcDest.left);  
m_backpos.x = min(m_backpos.x, 0);  
  
m_primary.BitBlt(m_rcDest.left + m_backpos.x,  
    m_rcDest.top + m_backpos.y + dy, m_background);  
  
// draw sprite  
m_primary.BitBlt(m_rcDest.left + m_spritepos.x + m_backpos.x,  
    m_rcDest.top + m_spritepos.y + m_backpos.y, m_sprite,  
    DDBLT_KEYSRC);  
  
m_font.TextOut(m_primary, m_rcDest.left+5, m_rcDest.top+1,  
    "Hello, DirectDraw!");  
}
```

The drawing of the “game” scene is divided into four parts. The first part covers the client area not covered by the background image. A long and narrow back ground image is used in the program. The code tries to put the background image in the center of the window vertically and fills the top and bottom margins with solid color fills. The second part draws the background image, but most of the code deals with calculating how the background image should be scrolled to make the plane visible. The third part draws the transparent plane sprite using source color keying. The last part displays a simple fixed string on the top left corner of the window.

When the program is running, you can use keyboard arrow keys to fly the plane on the scrollable background scene. [Figure 18-4](#) shows a screen shot from the DEMODD program, which uses the KSpriteDemo class.

Figure 18-4. Drawing text, bitmap, and sprite in DirectDraw.



If you are really into game programming, add more enemy sprite objects, implement collision detection, and add

weapon firing to make a real game.

[< BACK](#) [NEXT >](#)

18.4 DIRECT3D IMMEDIATE MODE

Although DirectDraw provides hardware acceleration, it provides only limited drawing functions. When using DirectDraw, you constantly have the feeling that you're writing a device driver, rather than an application program, because you have take into account lots of details.

The Direct3D Immediate Mode, on the contrary, is a much more powerful graphics programming API. Direct3D supports logical color, depth buffer, clipping, alpha blending, texture, viewport, world transformation, view matrix, projection, lighting, drawing points, lines and triangles, texture, midmap, fogging, environment mapping, stencil buffering, etc.

Although Direct3D Immediate Mode is designed to be a three-dimensional graphics API, you can certainly use it to draw two-dimensional graphics, which just lie on a single plane of the 3D space. In this section, we will briefly discuss how to write a simple Direct3D Immediate Mode program.

Creating a Direct3D Immediate Mode Environment

To use Direct3D IM, you need more than a DirectDraw object and a DirectDraw surface as encapsulated by the KDirectDraw class. Normally, you need a Direct3D object, a Direct3DDevice object, a background drawing surface, and a depth buffer. The Direct3D object controls access to Direct3D support. The background drawing buffer is where drawing actually happens, with the help of a depth buffer to implement hidden-surface removal. And finally, the Direct3DDevice object acts like a drawing device.

[Listing 18-5](#) shows the KDirect3D class that encapsulates a Direct3D environment.

Listing 18-5 KDirect3D Class for Direct3D IM Environment

```
class KDirect3D : public KDirectDraw
{
protected:
    IDirect3D7      * m_pD3D;
    IDirect3DDevice7 * m_pD3DDevice;

    KOffScreenSurface  m_backsurface;
    KOffScreenSurface  m_zbuffer;
    bool              m_bReady;

    virtual HRESULT Discharge(void);

    virtual HRESULT OnRender(void)
    {
        return S_OK;
    }
}
```

```
virtual HRESULT OnInit(HINSTANCE hInst)
{
    m_bReady = true;

    return S_OK;
}

virtual HRESULT OnDischarge(void)
{
    m_bReady = false;
    return S_OK;
}

public:

KDIRECT3D(void);
~KDIRECT3D(void)
{
    Discharge();
}

virtual HRESULT SetupDirectDraw(HWND hWnd, HWND hTop,
                               int nBufferCount=0, bool bFullScreen=false,
                               int width=0, int height=0, int bpp=0);

virtual HRESULT ShowFrame(HWND hWnd);
virtual HRESULT RestoreSurfaces(void);

virtual HRESULT Render(HWND hWnd);
virtual HRESULT ReCreate(HINSTANCE hInst, HWND hTop, HWND hWnd);
virtual HRESULT OnResize(HINSTANCE hInst, int width, int height,
                       HWND hTop, HWND hWnd);
};

HRESULT KDIRECT3D::SetupDirectDraw(HWND hTop, HWND hWnd, int nBufferCount,
                                   bool bFullScreen, int width, int height, int bpp)
{
    HRESULT hr = KDIRECTDRAW::SetupDirectDraw(hTop, hWnd, nBufferCount,
                                              bFullScreen, width, height, bpp);
    if ( FAILED( hr ) )
        return hr;

    // reject 8-bpp device
    if ( GetDisplayBpp(m_pDD)<=8 )
        return DDERR_INVALIDMODE;

    // create back surface
    hr = m_backsurface.CreateOffScreenSurface(m_pDD, width, height);
    if ( FAILED(hr) )
```

```
return hr;

// Query DirectDraw for access to Direct3D
m_pDD->QueryInterface( IID_IDirect3D7, (void **) & m_pD3D );
if ( FAILED(hr) )
    return hr;

CLSID iidDevice = IID_IDirect3DHALDevice;
// create Z-buffer
hr = m_zbuffer.CreateZBuffer(m_pD3D, m_pDD, iidDevice, width, height);

if ( FAILED(hr) )
{
    iidDevice = IID_IDirect3DRGBDevice;
    hr = m_zbuffer.CreateZBuffer(m_pD3D, m_pDD, iidDevice,
        width, height);
}

if ( FAILED(hr) )
    return hr;

// attach Z-buffer to back surface
hr = m_backsurface.Attach(m_zbuffer);
if ( FAILED(hr) )
    return hr;

hr = m_pD3D->CreateDevice( iidDevice, m_backsurface, & m_pD3DDevice );

{
    D3DVIEWPORT7 vp = { 0, 0, width, height, (float)0.0, (float)1.0 };

    return m_pD3DDevice->SetViewport( &vp );
}
}
```

The KDirect3D class adds five member variables to the KDirectDraw class: a pointer to a Direct3D7 object, a pointer to a Direct3DDevice7 object, a background buffer, a depth buffer, and a Boolean value.

The main initialization routine is the same as SetupDirectDraw, which first calls the KDirectDraw::SetupDirectDraw implementation to set up DirectDraw support. After DirectDraw support is properly initialized, the routine checks if the system is running in a palette-based mode, and returns an error value if this is the case. A Direct 3D program works best in a high color or true color display mode; a palette-based mode is supported, but it is too restrictive.

Direct3D drawing is very complicated, so it's preferable to keep the drawing away from the user's eyes by drawing to a background surface. For full-screen mode, double buffering or triple buffering can be used. For windowed mode, a separate background drawing surface can be used. The routine creates an off-screen surface that is the same size as the window's client area. To create the depth buffer for the 3D drawing surface, we first need to get a pointer to the IDirect3D7 interface, which is supported by the DirectDraw object created using DirectDrawCreateEx. The depth buffer itself is also an off-screen surface, except we have to use the IDirect3D7::EnumZBufferFormats method to

query for the depth buffer formats supported by the current device. The routine tries to create the Z-buffer using a hardware-accelerated device first but switches to a software emulation device if the attempt fails. Once created, Z-buffer needs to be attached to the background surface.

The last part of the KDirect3D::SetupDirectDraw method creates a Direct3D Device7 object based on the background surface and sets up a viewport on the device. The Direct3DDevice7 object provides the interface to 3D rendering features implemented on a 3D-enabled surface. The first four fields in the viewport define a rectangle area on the surface where rendering happens; the last two fields define the range of values in the Z-buffer.

Handling Window Resizing

The background surface and Z-buffer are both created using the client window's size when running in windowed mode. However, when the user resizes the window, these surfaces need to be recreated with the new window size. An easy implementation is to destroy all DirectDraw/Direct3D objects and recreate everything from scratch.

Here are related methods that handle Direct3D environment destruction and recreation.

```
HRESULT KDirect3D::Discharge(void)
{
    SAFE_RELEASE(m_pD3DDevice);
    m_backsurface.Discharge();
    m_zbuffer.Discharge();

    SAFE_RELEASE(m_pD3D);

    return KDirectDraw::Discharge();
}

HRESULT KDirect3D::ReCreate(HINSTANCE hInst, HWND hTop, HWND hWnd)
{
    if ( FAILED(OnDischarge()) )
        return E_FAIL;
    if ( FAILED( Discharge() ) ) // free all resources
        return E_FAIL;
    SetClientRect(hWnd);
    HRESULT hr = SetupDirectDraw(hTop, hWnd, 0, false,
        m_rcDest.right - m_rcDest.left,
        m_rcDest.bottom - m_rcDest.top);

    if ( SUCCEEDED(hr) )
        return OnInit(hInst);
    else
        return hr;
}

HRESULT KDirect3D::OnResize(HINSTANCE hInst, int width, int height,
    HWND hTop, HWND hWnd)
{
```

```
if ( ! m_bReady )
    return S_OK;

if ( width == (m_rcDest.right - m_rcDest.left) )
if ( height == (m_rcDest.bottom - m_rcDest.top) )
    return S_OK;

return ReCreate(hInst, hTop, hWnd);
}
```

The Discharge method frees all resources allocated with a KDirect3D object. The ReCreate method calls Discharge to free resources, and then SetupDirectDraw to create a new Direct3D environment. The OnSize method calls ReCreate when a window's size is actually changed.

Two-Step Rendering

With a background drawing surface, rendering is done in two steps, first drawing on the background surface, and then copying from the background surface to primary surface. The same technique can be applied to a DirectDraw object to implement flicker less drawing.

Here are the two methods for supporting two-step rendering in the KDirect3D class.

```
HRESULT KDirect3D::Render(HWND hWnd)
```

```
{
    if ( ! m_bReady )
        return S_OK;
    HRESULT hr = OnRender();
    if ( FAILED(hr) )
        return hr;

    hr = ShowFrame(hWnd);
```

```
    if ( hr = DDERR_SURFACELOST )
        return RestoreSurfaces();
```

```
    else
        return hr;
```

```
}
```

```
HRESULT KDirect3D::ShowFrame(HWND hWnd)
```

```
{
    if ( m_bReady )
    {
        SetClientRect(hWnd);
        return m_primary.Blt(&m_rcDest, m_backsurface, NULL, DDBLT_WAIT);
    }
    else
        return S_OK;
```

}

The KDIRECT3D::Render method calls the virtual method OnRender to do actual drawing, and then ShowFrame to copy from the background surface to the primary surface. When multiple DirectX applications are running, it's possible that video memory resources allocated for the surface have been reclaimed by other applications. The code checks for the so-called surface-lost condition and calls IDIRECTDRAWSURFACE7::Restore method on each surface to restore it.

Putting Direct3D into a Window

The KDIRECT3D class is designed to be a generic class that can be used anywhere. As such, window message handling should be implemented in a separate class. Here is a simple window that supports Direct3D Immediate Mode.

```
class KD3DWin : public KWindow, public KDIRECT3D
{
    bool    m_bActive;
    HINSTANCE m_hInst;
    LRESULT WndProc(HWND hWnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
    {
        switch( uMsg )
        {
            case WM_CREATE:
                m_hWnd = hWnd;
                m_bActive = false;

                if ( FAILED( ReCreate(m_hInst, hWnd, hWnd) ) )
                    CloseWindow(hWnd);

                SetTimer(hWnd, 101, 1, NULL);
                return 0;

            case WM_PAINT:
                ShowFrame(hWnd);
                break;

            case WM_SIZE:
                m_bActive = (SIZE_MAXHIDE!=wParam) &&
                            (SIZE_MINIMIZED!=wParam);

                if ( m_bActive && FAILED(OnResize(m_hInst, LOWORD(lParam),
                                                HIWORD(lParam), hWnd, hWnd)) )
                    CloseWindow(hWnd);
                break;

            case WM_TIMER:
                if ( m_bActive )
```

```
    Render(hWnd);
    return 0;

    case WM_DESTROY:
        KillTimer(hWnd, 101);
        Discharge();
        PostQuitMessage(0);
        return 0L;
    }

    return DefWindowProc( hWnd, uMsg, wParam, lParam );
}

void GetWndClassEx(WNDCLASSEX & wc)
{
    KWindow::GetWndClassEx(wc);
    wc.style |= (CS_HREDRAW | CS_VREDRAW);
}

public:

KD3DWin(HINSTANCE hInst)
{
    m_hInst = hInst;
}
};
```

The KD3DWin class is derived from the KWindow class for basic window support, and from the KDirect3D class for Direct3D support. The Direct3D environment is created during WM_CREATE message handling, resized when WM_SIZE message is received, and finally destroyed in handling WM_DESTROY message. The WM_PAINT message handling simply calls KDirect3D::ShowFrame to display from the background surface.

The KD3DWin class creates a timer to drive animation in the window. WM_TIMER message handling calls the KDirect3D::Render method to render and display a new scene. The frequency for timer messages is limited by the operating-system design. On Windows 95/98, you can get a maximum of only 18–19 messages per second; on Windows NT/2000, you can get up to 100 messages per second. Normally DirectX programs use idle cycles in the message loop to drive animation for a higher frame rate. But changing the main thread's message loop for a single window may not be a good idea. An alternative is to use a separate thread for rendering.

Texture Surface

The basic shapes you can draw with a Direct3D device are points, lines, and triangles, which covers the most basic geometric shapes in one-dimensional, two-dimensional, and three-dimensional worlds. To make geometric shapes look like real-life objects, the simplest thing Direct3D allows is applying texture to triangles.

A texture bitmap first needs to be loaded into a texture surface to be used by Direct3D. A texture surface is an off-screen surface with a bitmap loaded, except that the Direct3D device supports several types of texture bitmap formats for performance and features. So the application has to choose the right texture format to use by selecting among the available texture formats.

The KOffScreenSurface::CreateTextSurface method is the basic method to create a texture surface. Creating a texture surface from a bitmap takes just a few more steps.

```
HRESULT CALLBACK TextureCallback(DDPIXELFORMAT* pddpf, void * param)
{
    // find a simple >=16-bpp texture format
    if ( (pddpf->dwFlags & (DDPF_LUMINANCE | DDPF_BUMPLUMINANCE |
        DDPF_BUMPUDUV | DDPF_ALPHAPIXELS))==0 )
        if ( (pddpf->dwFourCC == 0) && (pddpf->dwRGBBitCount>=16) )
    {
        memcpy(param, pddpf, sizeof(DDPIXELFORMAT));
        return DDENUMRET_CANCEL; // stop search
    }

    return DDENUMRET_OK; // continue
}

HRESULT KOffScreenSurface::CreateTextureSurface(IDirect3DDevice7 *
    pD3DDevice, IDirectDraw7 * pDD, unsigned width, unsigned height)
{
    // query device caps
    D3DDEVICEDESC7 ddDesc;

    HRESULT hr = pD3DDevice->GetCaps(&ddDesc);
    if ( FAILED(hr) )
        return hr;

    m_ddsd.dwFlags      = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH |
        DDSD_PIXELFORMAT | DDSD_TEXTURESTAGE;
    m_ddsd.ddsCaps.dwCaps = DDSCAPS_TEXTURE;
    m_ddsd.dwWidth      = width;
    m_ddsd.dwHeight     = height;

    // Turn on texture management for hardware devices
    if ( (ddDesc.deviceGUID == IID_IDirect3DHALDevice) ||
        (ddDesc.deviceGUID == IID_IDirect3DTnLHalDevice) )
        m_ddsd.ddsCaps.dwCaps2 = DDSCAPS2_TEXTUREMANAGE;
    else
        m_ddsd.ddsCaps.dwCaps |= DDSCAPS_SYSTEMMEMORY;
    // Adjust width and height, if the driver requires it
    if ( ddDesc.dpcTriCaps.dwTextureCaps & D3DPTEXTURECAPS_POW2 )
    {
        for (m_ddsd.dwWidth=1; width>m_ddsd.dwWidth; m_ddsd.dwWidth<<=1);
        for (m_ddsd.dwHeight=1; height>m_ddsd.dwHeight;
            m_ddsd.dwHeight<<=1);
    }
}
```

```
if ( ddDesc.dpcTriCaps.dwTextureCaps & D3DPTEXTURECAPS_SQUAREONLY )
{
    if ( m_ddsd.dwWidth > m_ddsd.dwHeight )
        m_ddsd.dwHeight = m_ddsd.dwWidth;
    else
        m_ddsd.dwWidth = m_ddsd.dwHeight;
}

memset(& m_ddsd.ddpfPixelFormat, 0, sizeof(m_ddsd.ddpfPixelFormat));
pD3DDevice->EnumTextureFormats(TextureCallback,
    & m_ddsd.ddpfPixelFormat);

if ( m_ddsd.ddpfPixelFormat.dwRGBBitCount )
    return pDD->CreateSurface( & m_ddsd, & m_pSurface, NULL );
else
    return E_FAIL;
}
```

A Direct3D Immediate Mode Sample

With the environment set up by the KDirect3D class and KD3DWin class, and texture bitmap support, to implement a Direct3D window you just derive from the K Direct 3D class and override a few methods. [Listing 18-6](#) shows a simple Direct3D Immediate Mode example that draws a pyramid shape with textures, Z-buffer, and animation.

Listing 18-6 Direct3D Immediate Mode Sample Program

```
class KDirect3DDemo : public KDirect3D
{
    KOffScreenSurface m_texture[4];

public:
    HRESULT OnRender(void);
    HRESULT OnInit(HINSTANCE hInst);
    HRESULT OnDischarge(void);
};

HRESULT KDirect3DDemo::OnInit(HINSTANCE hInst)
{
    D3DMATERIAL7 mtrl;
    memset(&mtrl, 0, sizeof(mtrl));
    mtrl.ambient.r = 1.0f;
    mtrl.ambient.g = 1.0f;
    mtrl.ambient.b = 1.0f;
    m_pD3DDevice->SetMaterial( &mtrl );

    m_pD3DDevice->SetRenderState( D3DRENDERSTATE_AMBIENT,
```

```
    RGBA_MAKE(255, 255, 255, 0) );\n\nD3DMATRIX mat;\nmemset(& mat, 0, sizeof(mat));\nmat._11 = mat._22 = mat._33 = mat._44 = 1.0f;\n\n// view matrix, on z-axis by 10 units.\nD3DMATRIX matView = mat;\nmatView._43 = 10.0f;\nm_pD3DDevice->SetTransform( D3DTRANSFORMSTATE_VIEW, &matView );\n\nmat._11 = 2.0f;\nmat._22 = 2.0f;\nmat._34 = 1.0f;\nmat._43 = -0.1f;\nmat._44 = 0.0f;\nm_pD3DDevice->SetTransform( D3DTRANSFORMSTATE_PROJECTION, &mat);\n\n// enable Z-buffer\nm_pD3DDevice->SetRenderState( D3DRENDERSTATE_ZENABLE, TRUE);\n\nfor (int i=0; i<4; i++)\n{\n    const int nResID[] = { IDB_TIGER, IDB_PANDA,\n                           IDB_WHALE, IDB_ELEPHANT };\n\n    BITMAPINFO * pDIB = LoadBMP(hInst, MAKEINTRESOURCE(nResID[i]));\n\n    if ( pDIB )\n        m_texture[i].CreateTextureSurface(m_pD3DDevice, m_pDD, pDIB);\n    else\n        return E_FAIL;\n}\n\nm_bReady = true;\nreturn S_OK;\n}\n\nHRESULT KDIRECT3DDEMO::ONDISCHARGE(void)\n{\n    m_bReady = false;\n\n    for (int i=0; i<4; i++)\n        m_texture[i].Discharge();\n\n    return S_OK;\n}
```

The KDIRECT3DDEMO class derives from the KDIRECT3D class. It has an array of KOFFSCREENSURFACE objects to hold

four textures. It overrides three methods to handle initialization, destruction, and rendering.

The OnInit method sets up simple white material, ambient white lighting, fixed view matrix, fixed projection matrix, and enables Z-buffering. The last part of the OnInit method initializes four texture surfaces using bitmap resources. The OnDischarge method releases resources allocated by the OnInit method.

Here is the implementation of the OnRender method.

```
HRESULT DrawTriangle(IDirect3DDevice7 * pDevice,
    int x0, int y0, int z0,
    int x1, int y1, int z1,
    int x2, int y2, int z2)
{
    D3DVERTEX vertices[3];

    D3DVECTOR p1( (float)x0, (float)y0, (float)z0 );
    D3DVECTOR p2( (float)x1, (float)y1, (float)z1 );
    D3DVECTOR p3( (float)x2, (float)y2, (float)z2 );

    D3DVECTOR vNormal = Normalize(CrossProduct(p1-p2, p2-p3));

    // Initialize the 3 vertices for the front of the triangle
    vertices[0] = D3DVERTEX( p1, vNormal, 0.5f, 0.0f );
    vertices[1] = D3DVERTEX( p2, vNormal, 1.0f, 1.0f );
    vertices[2] = D3DVERTEX( p3, vNormal, 0.0f, 1.0f );

    return pDevice->DrawPrimitive(D3DPT_TRIANGLELIST, D3DFVF_VERTEX,
        vertices, 3, NULL);
}

HRESULT KDIRECT3DDemo::OnRender(void)
{
    m_pD3DDevice->Clear(0, NULL, D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
        RGBA_MAKE(0, 0, 0xff, 0), 1.0f, 0);

    if ( FAILED( m_pD3DDevice->BeginScene() ) )
        return E_FAIL;

    double time = GetTickCount() / 2000.0;
    D3DMATRIX matLocal;
    memset(&matLocal, 0, sizeof(matLocal));
    matLocal._11 = matLocal._33 = (FLOAT) cos( time );
    matLocal._13 = matLocal._31 = (FLOAT) sin( time );
    matLocal._22 = matLocal._44 = 1.0f;
    m_pD3DDevice->SetTransform( D3DTRANSFORMSTATE_WORLD, &matLocal );

    m_pD3DDevice->SetTexture( 0, m_texture[0] );
    DrawTriangle(m_pD3DDevice, 0, 3, 0, 3, -3, 0, -3, 3);
```

```
m_pD3DDevice->SetTexture( 0, m_texture[1] );
DrawTriangle(m_pD3DDevice, 0, 3, 0, 0, -3, -3, 3, -3, 0);

m_pD3DDevice->SetTexture( 0, m_texture[2] );
DrawTriangle(m_pD3DDevice, 0, 3, 0, -3, -3, 0, 0, -3, -3);

m_pD3DDevice->SetTexture( 0, m_texture[3] );
DrawTriangle(m_pD3DDevice, 0, 3, 0, 0, -3, 3, -3, -3, 0);

m_pD3DDevice->EndScene();

return S_OK;
}
```

The OnRender method clears the 3D device with a solid blue color and resets the Z-buffer, using the IDirect3DDevice7::Clear method. The rendering starts with BeginScene and ends with EndScene. It then queries for system time and uses it to set up a rotating world-transformation matrix around the y-axis. The cycle for the rotation is around 12 seconds ($2 * \pi * 2$). Following that are calls to draw four faces of a pyramid, each using a different texture.

Each face of the pyramid is drawn as a triangle in 3D space. The helper routine DrawTriangle accepts three 3D points using integer coordinates. Direct3D uses the D3D VERTEX structure to specify a vertex for point, line, and triangle drawing. A simple vertex contains a 3D point, a normal vector, and a texture coordinate. The 3D point determines the location, the normal vector specifies the direction of the surface the vertex is on, and the texture coordinate specifies the corresponding pixel location on the texture bitmap. When a texture is applied, Direct3D automatically interpolates the texture bitmap over every pixel in the whole triangle. The DrawTriangle routine converts integer coordinates to floating-point coordinates, calculates the normal vector for the surface, and sets a fixed texture coordinate for each vertex. The data are stored in a D3D VERTEX array and drawn using a single call to the IDirect3DDevice::DrawPrimitive method. The DrawPrimitive method is the primary drawing call on a Direct3D device. It draws point list, line list, line strip, triangle list, triangle strip, and triangle fan.

[Figure 18-5](#) shows one scene during the rotation.

Figure 18-5. Direct3D immediate mode sample.



With a rich feature set, Direct3D Immediate Mode programming is much more complicated than what can be shown here. Refer to Microsoft DirectX documentation, which includes a quite good tutorial and sample programs. There are also quite a few good DirectX books coming.

[< BACK](#) [NEXT >](#)

[< BACK](#)

18.5 SUMMARY

The focus of this chapter has been an introduction to DirectDraw and Direct3D Immediate Mode. We have shown how to build C++ classes for generic DirectDraw, Direct3D support that are separate from window handling and so can be integrated into any window.

For DirectDraw, this chapter has gone into quite some depth to show how to use the hardware-accelerated DirectDraw Blt method, how to access a locked surface directly, and how to use GDI for help. Classes and methods have been provided to handle off-screen bitmap surface, texture surface, Z-buffer surface, font surface, and text drawing.

We hope this chapter has demonstrated that DirectDraw/Direct3D programming is not hard, especially with nicely designed C++ classes. DirectDraw/Direct3D should not be restricted to game or educational programs; normal window applications can take advantage of hardware DirectDraw/Direct3D support for better data visualization and more friendly user interfaces.

Further Reading

Microsoft provides quite good documentation, tutorials, and sample programs for DirectDraw, Direct3D Immediate Mode, and other parts of the whole DirectX. Refer to MSDN for documentation and tutorials, and check Platform SDK or DirectX SDK for sample programs.

With Direct3D Immediate Mode, Microsoft provides a framework for building Direct3D applications, whose main class is CD3DApplication. The related source code can be found under the SDK Samples\MultiMedia\DIM directory, under the Include and Src\DIM subdirectory. We choose not to use Microsoft's Direct3D framework because it tightly links application, window, and DirectDraw/Direct3D support together. Using the framework directly, you can only create an application with a single window that supports Direct3D. Even the window message-dispatching routine uses a global variable to pass window messages to a virtual message handler defined in the CD3DApplication class. There is a very nice texture container class, but there is no class for the generic DirectDraw surface handling. There is also a very useful class for loading DirectX .X files, which is Microsoft's data format for 3D models. Over all, the D3D framework has lots of useful code, but you have to adapt it to add Direct3D support smoothly to your existing C++ window programs.

DirectX7 is relatively new. As such, good books are still on the horizon at the time of writing. With Microsoft's documentation, tutorial, and sample programs, what you really need may be a good OpenGL book, because Direct3D Immediate Mode and OpenGL have lots of things in common.

Sample Programs

[Chapter 18](#) comes with three sample programs and several wrapper classes for DirectDraw and Direct3D IM (see [Table 18-1](#)).

Table 18-1. Sample Programs for Chapter 18

Directory	Description
Samples\Chapt_18\ddbasic	Basic DirectDraw demo program.
Samples\Chapt_18\DemoDD	DirectDraw in MDI child window, pixel drawing, line drawing, area fill, bitmap, sprite, text drawing.
Samples\Chapt_18\DemoD3D	Direct3D Immediate Mode in SDI window, background buffer rendering, Z-buffer, texture surfaces.

[< BACK](#)