Part of the five-volume
Microsoft® Win32® Developer's Reference Library

Microsoft®

The essential reference to Win32® technologies and APIs

David Iseminger
Series Editor
www.iseminger.com

Microsoft® Windows® GDI

BASED ON
msdn™ library

Microsoft®

The essential reference to Win32® technologies and APIs

David Iseminger
Series Editor

Microsoft® Windows® GDI

BASED ON
msdn™ library

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

Library of Congress Cataloging-in-Publication Data
Iseminger, David, 1969-
Microsoft Win32 Developer's Reference Library / David Iseminger.
p. cm.
ISBN 0-7356-0816-4
1. Microsoft Win32. 2. Operating systems (Computers) I. Title.
QA76.76.O63 I74 1999
005.26'8--dc21 99-045609
CIP

Printed and bound in the United States of America.

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

Distributed in Canada by Penguin Books Canada Limited.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at mspress.microsoft.com.

The example companies, organizations, products, people, and events depicted herein are fictitious. No association with any real company, organization, product, person, or event is intended or should be inferred.

Acquisitions Editor: Ben Ryan
Project Editor: Wendy Zucker

Part No. 097-0002308

Acknowledgements

Acknowledgements are often tricky things; generally, the day after books are printed you think of someone who absolutely should have been recognized, whom you now have rudely omitted. You'd think authors would keep an ongoing list. Oh well, here goes:

First, thanks to Ben Ryan at Microsoft Press for sharing my enthusiasm about the series idea, and for keeping up with the myriad of issues that cropped up, and for managing the business details associated with publishing this series. Thanks also to Steve Guty at Microsoft Press for seeing certain publishing issues through the wringer.

Wendy Zucker kept in step with the difficult and tight schedule at Microsoft Press, and orchestrated things in the way only project editors can endure. John Pierce was also instrumental in seeing the publishing process through completion; many thanks to both of them. The cool Win32 cover art was created by Greg Hickman—thanks for the excellent work; I'm a firm believer that artwork and packaging are integral to the success of a project. Marketing acknowledgements go out to Jocelyn Paul, for her coordination efforts with MSDN and her other unsung victories.

On the SDK side of things, thanks to Morgan Seeley for introducing me to the editor at Microsoft Press, and thereby routing this series to the right place. Throughout the process, Julie Solon provided lots of Win32 feedback and helped gather feedback from others, all of which was quite helpful in compiling the right collection of technologies…thanks to Julie for the help on that. Guy Smith pointed me to the information I needed for Volumes 4 and 5, and was always very responsive.

On the developer side of things, thanks go out to Lars Opstad and Paramesh Vaidyanathan for their help and openness, respectively, with letting me provide the common coding errors found in Chapter 5 of each of these volumes. Thanks on my behalf, and on behalf of anyone who finds that information useful (I'm sure that includes a bunch of people!).

Thanks are also in order for artist-guru David Deyo for transforming my functional "circled i" logo into a 3D piece of art, as well as for his work on the Iseminger.com site. You can see more of his artwork through links found at www.iseminger.com.

Last, but certainly not least, thanks to Margot Hutchison for doing all the things great agents do best.

Contents

CHAPTER 1

Introduction

Welcome to the *Microsoft Win32 Developer's Reference Library*, your comprehensive reference guide to the Win32 development environment. This library, and the entire Windows Programming Reference Series, is designed to deliver the most complete, authoritative, and accessible reference information available for Windows programming—without sacrificing focus. You'll notice that each book is dedicated to a logical group of technologies or development concerns; this approach has been taken specifically to enable you—the time-pressed and information-overloaded applications developer—to find the information you need quickly, efficiently, and intuitively.

In addition to its focus on Win32 reference material, the Win32 Library contains hard-won insider tips and tricks designed to make your programming life easier. For example, a thorough explanation and detailed tour of the new version of MSDN Online is included, as is a section that helps you get the most out of your MSDN Subscription. Don't have an MSDN subscription, or don't know why you should? I've included information about that too, including the differences among the three levels of MSDN subscriptions, what each level offers, and why you'd want a subscription when MSDN Online is available over the Internet.

Microsoft is fairly well known for its programming, so doesn't it make sense to share some of that knowledge? I thought it made sense, so that's why this—the Windows Programming Reference Series—is the source where you'll find such shared knowledge. Part 1 of each volume contains advice on how to avoid common programming problems. There is a reason for including so much reference, overview, shared-knowledge, and programming information about Win32 in a single publication: the Win32 Library is geared toward being your one-stop printed reference resource for the Win32 programming environment.

To ensure that you don't get lost in all the information provided in the Win32 Library, each volume's appendixes provide an all-encompassing programming directory to help you easily find the particular programming element you're looking for. This directory suite, which covers all the functions, structures, enumerations, and other programming elements found in Win32, gets you quickly to the volume and page you need, and also provides an overview of Microsoft technologies that would otherwise take you hours of time, reams of paper, and potfuls of coffee to compile yourself.

How the Win32 Library Is Structured

The Win32 Library consists of five volumes, each of which focuses on a particular area of the Win32 programming environment. The programming areas into which the five Win32 Library volumes have been divided and include the following:

Volume 1: Base Services
Volume 2: User Interface
Volume 3: GDI (Graphics Device Interface)
Volume 4: Common Controls
Volume 5: The Windows Shell

Dividing the Win32 Library—and therefore, dividing Win32—into these functional categories enables a software developer who's focusing on a particular programming area (such as the user interface) to maintain that focus under the confines of one volume. This approach enables you to keep one reference book open and handy, or tucked under your arm while researching that aspect of Windows programming on sandy beaches, without risking back problems (from toting around a 2,000-page Win32 tome), and without having to shuffle among multiple, less-focused books.

Within each Win32 Library volume there is also a deliberate structure. This per-volume structure has been created to further focus the reference material in a developer friendly manner and to enable developers to easily gather the information they need. To that end, each volume in the Win32 Library has the following parts:

Part 1: Introduction and Overview
Part 2: Reference
Part 3: Windows Programming Directory

Part 1 provides an introduction to the Win32 Library and to the Windows Programming Reference Series (what you're reading now), and a handful of chapters designed to help you get the most out of Win32, MSDN and MSDN Online, including a collection of insider tips and tricks. Just as each volume's Reference section (Part 2) contains different reference material, each volume's Part 1 contains different tips and tricks. To ensure that you don't miss out on some of them, make sure you take a look at Part 1 in each Win32 Library volume.

Part 2 contains the Win32 reference material particular to its volume, but it is *much* more than a simple collection of function and structure definitions. Because a comprehensive reference resource should include information about *how to use* a particular technology, as well as its definitions of programming elements, the information in Part 2 combines complete programming element definitions as well as instructional and explanatory material for each programming area.

Part 3 is the directory of Windows programming information. One of the biggest challenges of the IT professional is finding information in the sea of available resources, and Windows programming is no exception. In order to help you get a handle on Win32 programming references and Microsoft technologies in general, Part 3 puts all such information into an understandable, manageable directory that enables you to quickly find the information you need.

How the Win32 Library Is Designed

The Win32 Library, and all libraries in the Windows Programming Reference Series, is designed to deliver the most pertinent information in the most accessible way possible. The Win32 Library is also designed to integrate seamlessly with MSDN and MSDN Online by providing a look-and-feel that is consistent with their electronic counterparts. In other words, the way that a given function reference appears on the pages of this book has been designed specifically to emulate the way that MSDN and MSDN Online present their function reference pages.

The reason for maintaining such integration is simple: make it easy for you—the developer of Windows applications—to use the tools and get the ongoing information you need create quality programs. By providing a “common interface” among reference resources, your familiarity with the Win32 Library reference material can be immediately applied to MSDN or MSDN Online, and vice versa. In a word, it means *consistency*.

You’ll find this philosophy of consistency and simplicity applied throughout Windows Programming Reference Series publications. I’ve designed the series to go hand-in-hand with MSDN and MSDN Online resources. Such consistency lets you leverage your familiarity with electronic reference material, and apply that familiarity to let you get away from your computer if you’d like, take a book with you, and—in the absence of keyboards and e-mail and upright chairs—get your programming reading and research done. Of course, each of the Win32 Library books fits nicely right next to your mouse pad as well, even when opened to a particular reference page.

With any job, the simpler and more consistent your tools are, the more time you can spend doing work rather than figuring out how to use your tools. The structure and design of the Win32 Library provides you with a comprehensive, pre-sharpened toolset to build compelling Windows applications.

CHAPTER 2

What's in This Volume?

Similar to the first two volumes, this third volume of the Win32 Library—*Volume 3: GDI (Graphical Device Interface)*—focuses on one of the areas of Windows development that most applications programmers must work with throughout the process of creating their applications. Graphical Device Interface, commonly referred to as GDI, provides a comprehensive set of functions, structures, and other programmatic elements that developers can use in their applications to generate graphical output for displays, printers, and other devices or objects.

The things that applications can do with GDI programming elements includes drawing lines or shaped objects, specifying the colors or fills of such drawn objects, and applying the objects used to create them, such as brushes and pens. The categories of GDI elements in this volume of the Win32 Library include:

Bitmaps
Brushes
Clipping
Colors
Coordinate Spaces and Transformations
Device Contexts
Filled Shapes
Fonts and Text
Lines and Curves
Metafiles
Painting and Drawing
Paths
Pens
Printing and Print Spooler
Rectangles
Regions

Bitmaps enable application developers to manipulate graphical images that are stored on disk. Bitmaps are collections of structures that are stored on disk and that specify or contain information about the bitmap. Such information includes the header (which stores data about the bitmap, such as resolution and dimensions), a palette, and an array of bits that define the relationship between the pixels in the image.

A **Brush** is a tool used to paint the interior of shapes (such as squares or circles). Brushes can be used by all sorts of applications, such as drawing programs (filling shapes) and information managers (coloring a task box red for "overdue" indication).

Clipping is used to limit a given object's output to a specified region or path. For example, an application developer might use the clipping function to keep text from spilling over into areas or regions in which the text would clutter the graphical appearance, or would otherwise be inappropriate.

The reference section that covers **Colors** provides developers with the programmatic interfaces they need to enrich their applications with the various colors that Windows applications are capable of displaying.

You can use **Coordinate Spaces and Transformations** in a Windows application to do such things as rotate, skew, or to zoom in or out of a particular graphical area within a Windows application's graphical space.

By using a **Device Context**, Windows applications enable continued device independence. A device context, through the use of a pre-defined structure, defines a set of graphics objects and the attributes associated with them, as well as the graphics modes that affect their output.

Filled Shapes come in five forms—ellipse, chord, pie, polygon, and rectangle—and are outlined and filled by the current pen and brush. The filled shape reference provides functions that enable developers to use filled shapes in their applications.

Using **Fonts and Text** provides developers with the means to display text on output devices, as well as the capability to install, query, and select different fonts.

Lines and Curves are used by applications to draw graphical output onto raster devices. The lines and curves section provides reference for developers to...well...use lines and curves in their applications.

A **Metafile** stores pictures in a device-independent format. Metafiles guarantee device independence, whereas bitmaps do not. However, metafiles draw slowly, so keep that in mind when determining which format is most appropriate for your application.

The reference section titled **Painting and Drawing** provides an explanation of how Windows manages output to the display, and explains what applications must do to draw in a window.

A **Path** is one or more shapes that is outlined, filled, or both.

Pens are graphic tools that applications can use to draw lines and curves.

In order to print to any given printer device, applications use the functions, structures, messages, and escape functions in the **Printing and Print Spooler** reference chapter.

Windows applications specify rectangular areas and manipulate those areas through the functional reference found in the chapter titled **Rectangles**.

Regions are various-shaped areas that can be used for various programming reasons, such as filling or cursor-location testing.

Each of these GDI element categories is thoroughly explained, and its programmatic reference information detailed, in individual chapters in Part 2 of this volume. In general, each chapter begins with explanatory information on the given category, with the associated programming elements—functions, structures, enumerations, and other programming elements—detailed thereafter. For more information on any of these categories, check out the table of contents at the beginning of the book, and then jump to the appropriate chapter.

CHAPTER 3

Using Microsoft Reference Resources

These days it isn't the availability of information that's the problem, it's the availability of information. You read that right...but I'll clarify.

Not long ago, getting the information you needed was a challenge because there wasn't enough of it; to find the information you needed, you had to find out where such information might be located and then actually get access to that location, because it wasn't at your fingertips or on some globally available backbone, and such searching took time. In short, the availability of information was limited.

Today, information surrounds us and sometimes stifles us; we're overloaded with too much information, and if we don't take measures to filter out what we don't need to meet our goals, soon we become inundated and unable to discern what's "junk information" and what's information that we need to stay current, and therefore competitive. In short, the overload of available information makes it more difficult for us to find what we *really* need, and wading through the deluge slows us down.

This truism applies to Microsoft's own reference material as well; not because there is information that isn't needed, but rather because there is so much information that finding what *you* need can be as challenging as figuring out what to do with it once you have it. Developers need a way to cut through the information that isn't pertinent to them, and to get what they're looking for. One way to ensure you can get to the information you need is to know the tools you use; carpenters know how to use nail guns, and it makes them more efficient. Bankers know how to use ten-key machines, and it makes them more adept. If you're a developer of Windows applications, two tools you should know are MSDN and MSDN Online. The third tool for developers—reference books from the Windows Programming Reference Series—can help you get the most out of the first two.

Books in the Windows Programming Reference Series, such as those found in the *Microsoft Win32 Developer's Reference Library*, provide reference material that focuses on a given area of Windows programming. MSDN and MSDN Online, in comparison, contain all of the reference material that all Microsoft programming technologies has amassed over the past few years, and create one large repository of information. Regardless of how well such information is organized, there's a lot of it, and if you don't know your way around, finding what you need (even though it's in there, somewhere) can be frustrating and time-consuming and just an overall bad experience.

This chapter will give you the insight and tips you need to navigate MSDN and MSDN Online, and to enable you to use each of them to the fullest of their capabilities. Also, other Microsoft reference resources are investigated, and by the end of the chapter, you'll know where to go for the Microsoft reference information you need (and how to quickly and efficiently get there).

The Microsoft Developer Network (MSDN)

MSDN stands for Microsoft Developer Network, and its intent is to provide developers with a network of information to enable the development of Windows applications. Many people have either worked with MSDN or have heard of it, and quite a few have one of the three available subscription levels to MSDN, but there are many, many more who don't have subscriptions and could use some concise direction on what MSDN can do for a developer or development group. If you fall into any of these categories, this section is for you.

There is some clarification to be done with MSDN and its offerings; if you've heard of MSDN, or have had experience with MSDN Online, you may have asked yourself one of these questions during the process of getting up to speed with either resource:

- Why do I need a subscription to MSDN if resources such as MSDN Online are accessible for free over the Internet?
- What are the differences among the three levels of MSDN subscriptions?
- What happened to Site Builder Network...or, What is this Web Library?
- Is there a difference between MSDN and MSDN Online, other than the fact that one is on the Internet and the other is on a CD? Do their features overlap, separate, coincide, or what?

If you have asked these questions, then lurking somewhere in the back of your thoughts has probably been a sneaking suspicion that maybe you aren't getting as much out of MSDN as you could. Or, maybe you're wondering whether you're paying too much for too little, or not enough to get the resources you need. Regardless, you want to be in the know, not in the dark.

By the end of this chapter, you will know the answers to all these questions and more, along with some effective tips and hints on how to make the most effective use of MSDN and MSDN Online.

Comparing MSDN and MSDN Online

Part of the challenge of differentiating between MSDN and MSDN Online comes with determining which has the features you need. Confounding this differentiation is the fact that both have some content in common, yet each offers content unavailable with the other. But can their differences be boiled down? Yes, if broad strokes and some generalities are used:

- MSDN provides reference content *and* the latest Microsoft product software, all shipped to its subscribers on CD (or in some cases, on DVD).

- MSDN Online provides reference content *and* a development community forum, and is available only over the Internet.

Each delivery mechanism for the content that Microsoft is making available to Windows developers is appropriate for the medium, and each plays on the strength of the medium to provide its "customers" with the best presentation of material possible. These strengths and media considerations enable MSDN and MSDN Online to provide developers with different feature sets, each of which has its advantages.

MSDN is perhaps less "immediate" than MSDN Online because it gets to its subscribers in the form of CDs that come in the mail. However, MSDN can sit in your CD drive (or on your hard drive), and isn't subject to Internet speeds or failures. Also, MSDN has a software download feature that enables subscribers to automatically update their local MSDN content, over the Internet, as soon as it comes available without having to wait for the update CD to come in the mail. The interface with which MSDN displays its material—which looks a whole lot like a specialized browser window—is also linked to the Internet as a browser-like window. To further coordinate MSDN with the immediacy of the Internet, MSDN Online has a section of the site dedicated to MSDN subscribers that enables subscription material to be updated (on their local machines) as soon as it's available.

MSDN Online has lots of editorial and technical columns that are published directly to the site and are tailored (not surprisingly) to the issues and challenges faced by developers of Windows applications or Windows-based web sites. MSDN Online also has a customizable interface (much like MSN.com) that enables visitors to tailor the information that's presented upon visiting the site to the areas of Windows development in which they are most interested. However, MSDN Online, while full of up-to-date reference material and extensive online developer community content, doesn't come with Microsoft product software, and doesn't reside on your local machine.

Since it's easy to confuse the differences and similarities between MSDN and MSDN Online, it makes sense to figure out a way to quickly identify how and where they depart. Figure 3-1 puts the differences—and similarities—between MSDN and MSDN Online into a quickly identifiable format.

One feature that you will notice is shared between MSDN and MSDN Online is the interface—they are very similar. That's almost certainly a result of attempting to ensure that developers' user experience with MSDN is easily associated with the experience found on MSDN Online, and vice-versa.

Remember, too, that if you are an MSDN subscriber you can still use MSDN Online and its features. So it isn't an "either/or" question with regard to whether you need an MSDN subscription or whether you should use MSDN Online; if you have an MSDN subscription, you will probably continue to use MSDN Online along with the additional features provided with your MSDN subscription.

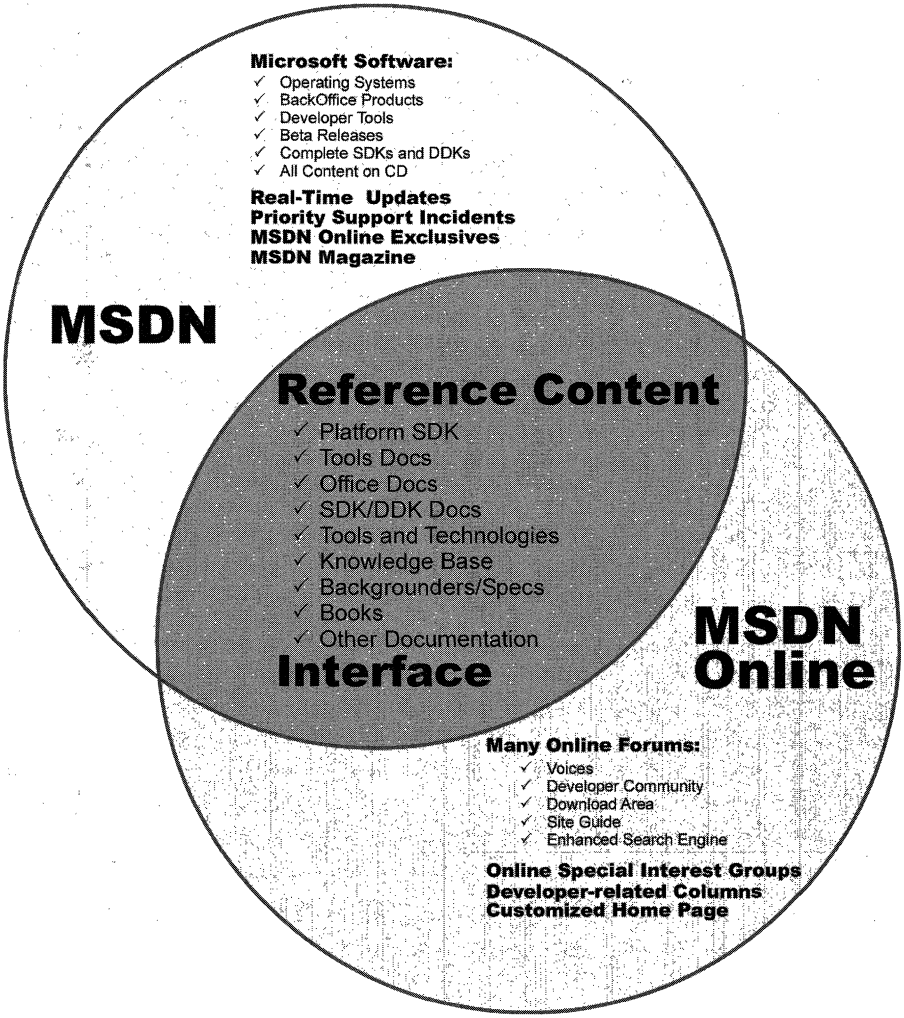


Figure 3-1: The similarities and differences in coverage between MSDN and MSDN Online.

MSDN Subscriptions

If you're wondering whether you might benefit from a subscription to MSDN, but you aren't quite sure what the differences between its subscription levels are, you aren't alone. This section aims to provide a quick guide to the differences in subscription levels, and an approximation what each subscription level costs.

There are three subscription levels for MSDN: Library, Professional, and Universal. Each has a different set of features. Each progressive level encompasses the lower level's features, and includes additional features. In other words, with the Professional subscription, you get everything provided in the Library subscription, plus additional features; with the Universal subscription, you get everything provided in the Professional subscription, plus even more features.

MSDN Library Subscription

The MSDN Library subscription is the basic MSDN subscription. While the Library subscription doesn't come with the Microsoft product software that the Professional and Universal subscriptions provide, it does come with other features that developers may find necessary in their development effort. With the Library subscription, you get the following:

- The Microsoft reference library, including SDK and DDK documentation, updated quarterly
- Lots of sample code, which you can cut and paste into your projects, royalty free
- The complete Microsoft Knowledge Base—*the* collection of bugs and workarounds
- Technology specifications for Microsoft technologies
- The complete set of product documentation, such as Visual Studio, Office, and others
- Complete (and in some cases, partial) electronic copies of selected books and magazines
- Conference and seminar papers—if you weren't there, you can use MSDN's notes

In addition to these items, you also get:

- Archives of MSDN Online columns
- Periodic e-mails from Microsoft chock full of development-related information
- A subscription to MSDN News, a bi-monthly newspaper from the MSDN folks
- Access to subscriber-exclusive areas and material on MSDN Online

MSDN Professional Subscription

The Professional subscription is a superset of the Library subscription. In addition to the features outlined in the previous section, MSDN Professional subscribers get the following:

- Complete set of Windows operating systems, including release versions of Windows 95, Windows 98, and Windows NT 4 Server and Workstation
- Windows SDKs and DDKs in their entirety
- International versions of Windows operating systems (as chosen)
- Priority technical support for two incidents in a development and test environment

MSDN Universal Subscription

The Universal subscription is the all-encompassing version of the MSDN subscription. In addition to everything provided in the Professional subscription, Universal subscribers get the following:

- The latest version of Visual Studio, Enterprise Edition
- The BackOffice test platform, which includes all sorts of Microsoft product software incorporated in the BackOffice family, each with special 10-connection license for use in the development of your software products
- Additional development tools, such as Office Developer, Front Page, and Project
- Priority technical support for two additional incidents in a development and test environment (for a total of four incidents)

Purchasing an MSDN Subscription

Of course, all of the features that you get with MSDN subscriptions aren't free. MSDN subscriptions are one-year subscriptions, which are current as of this writing. Just as each MSDN subscription escalates in functionality of incorporation of features, so does each escalate in price. Please note that prices are subject to change.

The MSDN Library Subscription has a retail price of $199, but if you're renewing an existing subscription you get a $100 rebate in the box. There are other perks for existing Microsoft customers, but those vary. Check out the Web site for more details.

The MSDN Professional Subscription is a bit more expensive than the Library, with a retail price of $699. If you're an existing customer renewing your subscription, you again get a break in the box, this time in the amount of a $200 rebate. You also get that break if you're an existing Library subscriber who's upgrading to a Professional subscription.

The MSDN Universal Subscription takes a big jump in price, at $2,499. If you're upgrading from the Professional subscription, the price drops to $1,999, and if you're upgrading from the Library subscription level there's an in-the-box rebate for $200.

As is often the case, there are academic and volume discounts available from various resellers, including Microsoft, so those who are in school or in the corporate environment can use their status (as learner or learned) to get a better deal—and in most cases, the deal is much better. Also, if your organization is using lots of Microsoft products, whether MSDN is a part of that group or not, whomever's in charge of purchasing should look into Microsoft Open License program; the Open License program gives purchasing breaks for customers that buy lots of products. Check out *www.microsoft.com/licensing* for more details. Who knows, if your organization qualifies you could end up getting an engraved pen from your purchasing department, or if you're really lucky maybe even a plaque of some sort, for saving your company thousands of dollars on Microsoft products.

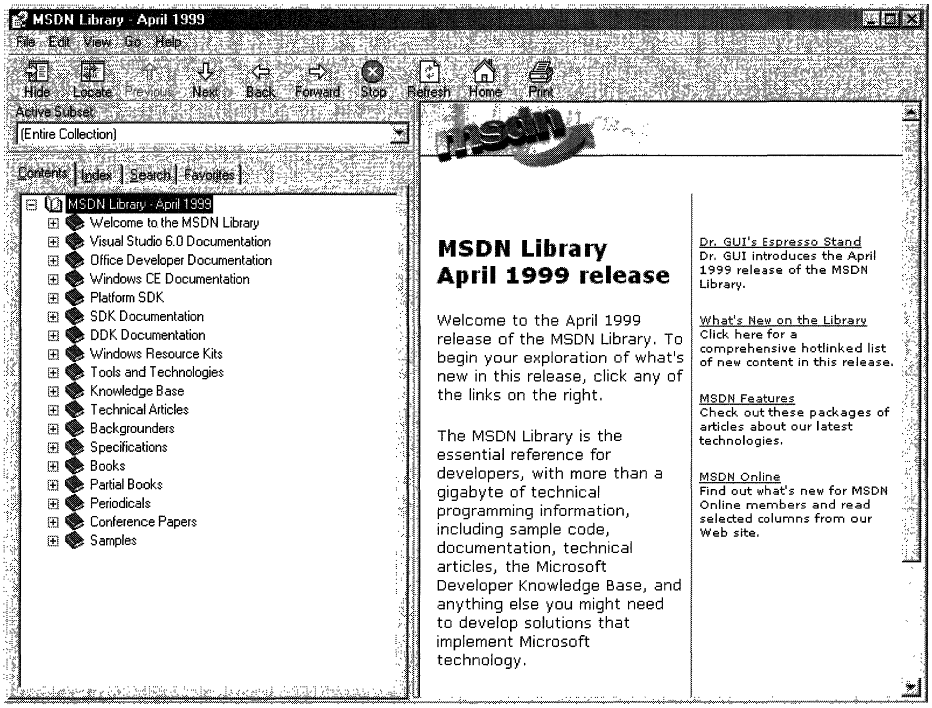
You can get MSDN subscriptions from a number of sources, including online sites specializing in computer-related information, such as *www.iseminger.com* (shameless self-promotion, I know), or from your favorite online software site. Note that not all software resellers carry MSDN subscriptions; you might have to hunt around to find one. Of course, if you have a local software reseller that you frequent, you can check out whether they carry MSDN subscriptions, too.

As an added bonus for owners of this Win32 Library, in the back of Volume 1: *Base Services*, you'll find a $200 rebate good toward an MSDN Universal subscription. For those of you doing the math, that means you actually *make* money when you purchase the Win32 Library and an MSDN Universal subscription. That means every developer in your organization can have the printed Win32 Library on their desk and the MSDN Universal subscription available on their desktop and still come out $50 ahead. That's the kind of math even accountants can like.

Using MSDN

MSDN subscriptions come with an installable interface, and the Professional and Universal subscriptions also come with a bunch of Microsoft product software such as Windows platform versions and BackOffice applications. There's no need to tell you how to use Microsoft product software, but there's a lot to be said for providing some quick but useful guidance on getting the most out of the interface to present and navigate through the seemingly endless supply of reference material provided with any MSDN subscription.

To those who have used MSDN, the interface shown in Figure 3-2 is likely familiar; it's the navigational front-end to MSDN reference material.



MSDN Library April 1999 release

Welcome to the April 1999 release of the MSDN Library. To begin your exploration of what's new in this release, click any of the links on the right.

The MSDN Library is the essential reference for developers, with more than a gigabyte of technical programming information, including sample code, documentation, technical articles, the Microsoft Developer Knowledge Base, and anything else you might need to develop solutions that implement Microsoft technology.

Dr. GUI's Espresso Stand
Dr. GUI introduces the April 1999 release of the MSDN Library.

What's New on the Library
Click here for a comprehensive hotlinked list of new content in this release.

MSDN Features
Check out these packages of articles about our latest technologies.

MSDN Online
Find out what's new for MSDN Online members and read selected columns from our Web site.

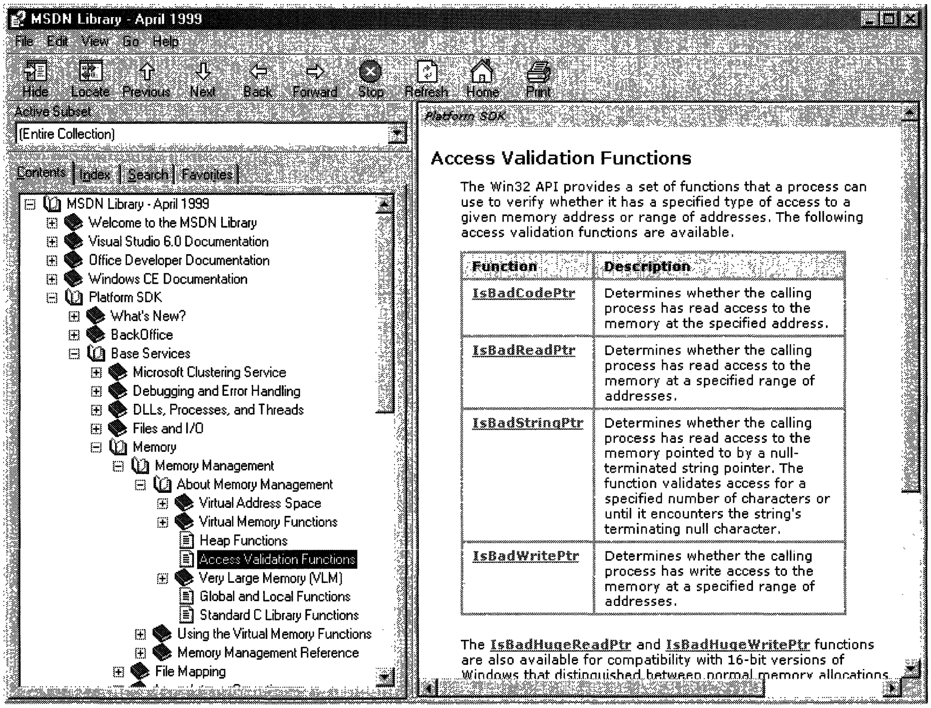
Figure 3-2: The MSDN interface.

The interface is familiar and straightforward enough, but if you don't have a grasp of its features and navigation tools, you can be left a little lost in its sea of information. With a few sentences of explanation and some tips for effective navigation, however, you can increase its effectiveness dramatically.

Navigating MSDN

One of the primary features of MSDN—and to many, its primary drawback—is the sheer volume of information it contains, over 1.1GB and growing. The creators of MSDN likely realized this, though, and have taken steps to assuage the problem. Most of those steps relate to enabling developers to selectively navigate through MSDN's content.

Basic navigation through MSDN is simple, and a lot like navigating through Windows Explorer and its folder structure. Instead of folders, MSDN has books into which it organizes its topics; expand a book by clicking the + box to its left, and its contents are displayed with its nested books or reference pages, as shown in Figure 3-3. If you don't see the left pane in your MSDN viewer, go to the **View** menu and select **Navigation Tabs** and they'll appear.


MSDN Library - April 1999
File Edit View Go Help
Hide Locate Previous Next Back Forward Stop Refresh Home Print
Active Subset
(Entire Collection)
Contents Index Search Favorites
MSDN Library - April 1999
Welcome to the MSDN Library
Visual Studio 6.0 Documentation
Office Developer Documentation
Windows CE Documentation
Platform SDK
What's New?
BackOffice
Base Services
Microsoft Clustering Service
Debugging and Error Handling
DLLs, Processes, and Threads
Files and I/O
Memory
Memory Management
About Memory Management
Virtual Address Space
Virtual Memory Functions
Heap Functions
Access Validation Functions
Very Large Memory (VLM)
Global and Local Functions
Standard C Library Functions
Using the Virtual Memory Functions
Memory Management Reference
File Mapping
Platform SDK

Access Validation Functions

The Win32 API provides a set of functions that a process can use to verify whether it has a specified type of access to a given memory address or range of addresses. The following access validation functions are available.

Function	Description
IsBadCodePtr	Determines whether the calling process has read access to the memory at the specified address.
IsBadReadPtr	Determines whether the calling process has read access to the memory at a specified range of addresses.
IsBadStringPtr	Determines whether the calling process has read access to the memory pointed to by a null-terminated string pointer. The function validates access for a specified number of characters or until it encounters the string's terminating null character.
IsBadWritePtr	Determines whether the calling process has write access to the memory at a specified range of addresses.

The IsBadHugeReadPtr and IsBadHugeWritePtr functions are also available for compatibility with 16-bit versions of Windows that distinguished between normal memory allocations

Figure 3-3: Basic navigation through MSDN.

The four tabs in the left pane of MSDN—increasingly referred to as property sheets these days—are the primary means of navigating through MSDN content. These four tabs, in coordination with the **Active Subset** drop-down box above the four tabs, are the tools you use to search through MSDN content. When used to their full extent, these coordinated navigation tools greatly improve your MSDN experience.

The **Active Subset** drop-down box is a filter mechanism; choose the subset of MSDN information you're interested in working with from the drop-down box, and the information in each of the four navigation tabs (including the **Contents** tab) limits the information it displays to the information contained in the selected subset. This means that any searches you do in the **Search** tab, and in the index presented in the **Index** tab, are filtered by their results and/or matches to the subset you define, greatly narrowing the number of potential results for a given inquiry, thereby enabling you to better find the information you're *really* looking for. In the **Index** tab, results that might match your inquiry but *aren't* in the subset you have chosen are grayed out (but still selectable). In the **Search** tab, they simply aren't displayed.

MSDN comes with the following pre-defined subsets:

Entire Collection

MSDN, Books and Periodicals

MSDN, Content on Disk 2 only

MSDN, Content on Disk 3 only

MSDN, Knowledge Base

MSDN, Office Development

MSDN, Technical Articles and Backgrounders

Platform SDK, BackOffice

Platform SDK, Base Services

Platform SDK, Component Services

Platform SDK, Data Access Services

Platform SDK, Graphics and Multimedia Services

Platform SDK, Management Services

Platform SDK, Messaging and Collaboration Services

Platform SDK, Networking Services

Platform SDK, Security

Platform SDK, Tools and Languages

Platform SDK, User Interface Services

Platform SDK, Web Services

Platform SDK, What's New?

Platform SDK, Win32 API

Repository 2.0 Documentation

Visual Basic Documentation

Visual C++ Documentation

Visual C++, Platform SDK and WinCE Docs

Visual C++, Platform SDK, and Enterprise Docs

Visual FoxPro Documentation

Visual InterDev Documentation

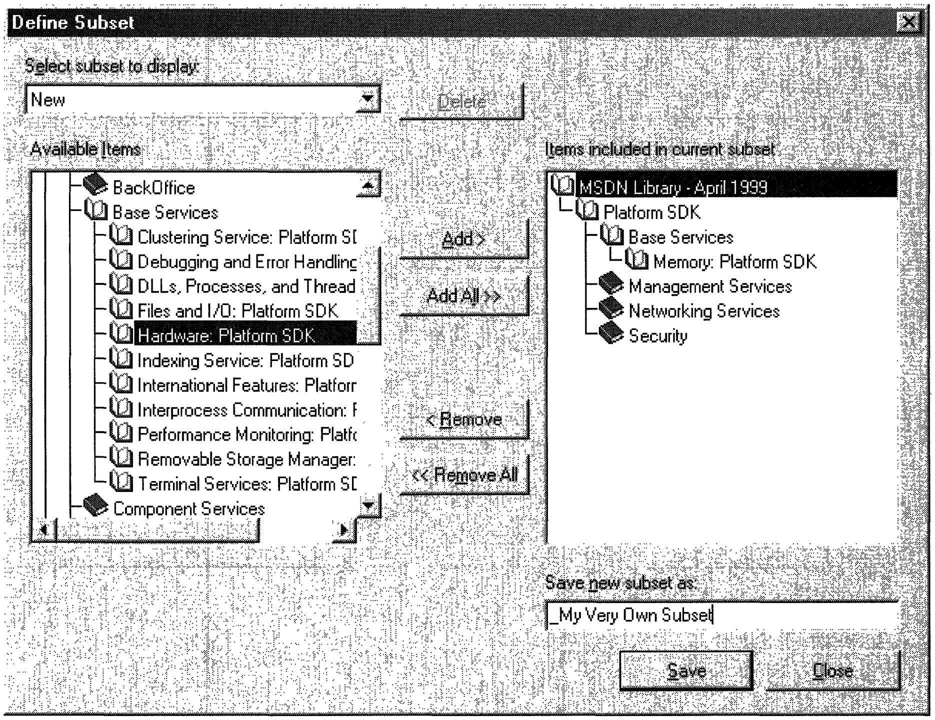
Visual J++ Documentation

Visual SourceSafe Documentation

Visual Studio Product Documentation

As you can see, these filtering options essentially mirror the structure of information delivery used by MSDN. But what if you are interested in viewing the information in a handful of these subsets? For example, what if you want to search on a certain keyword through the Platform SDK's Security, Networking Services, and Management Services subsets, as well as a little section that's nested way into the Base Services subset? Simple—you define your own subset.

You define subsets by choosing the **View** menu, and then selecting the **Define Subsets** menu item. You're presented with the window shown in Figure 3-4.



Select subset to display: New

Available Items

- BackOffice
- Base Services
  - Clustering Service: Platform SI
  - Debugging and Error Handling
  - DLLs, Processes, and Thread
  - Files and I/O: Platform SDK
  - Hardware: Platform SDK
  - Indexing Service: Platform SD
  - International Features: Platforr
  - Interprocess Communication: I
  - Performance Monitoring: Platfc
  - Removable Storage Manager:
  - Terminal Services: Platform SI
- Component Services

Items included in current subset

- MSDN Library - April 1999
  - Platform SDK
    - Base Services
      - Memory: Platform SDK
    - Management Services
    - Networking Services
    - Security

Save new subset as: _My Very Own Subset

Figure 3-4: The Define Subsets window.

Defining a subset is easy; just take the following steps:

1. Choose the information you want in the new subset; you can choose entire subsets or selected books/content within available subsets.
2. Add your selected information to the subset you're creating by clicking the **Add** button.
3. Name the newly created subset by typing in a name in the **Save New Subset** As box. Note that defined subsets (including any you create) are arranged in alphabetical order.

You can also delete entire subsets from the MSDN installation, if you so desire. Simply select the subset you want to delete from the **Select Subset To Display** drop-down box, and then click the nearby Remove button.

Once you have defined a subset, it becomes available in MSDN just like the pre-defined subsets and filters the information available in the four navigation tabs just like the pre-defined subsets do.

Quick Tips

Now that you know how to navigate MSDN, there are a handful of tips and tricks that you can use to make MSDN as effective as it can be.

Use the Locate button to get your bearings. Perhaps it's human nature to need to know where you are in the grand scheme of things, but regardless, it can be bothersome to have a reference page displayed in the right pane (perhaps jumped to from a search), without the **Contents** tab in the left pane being synchronized in terms of the reference page's location in the information tree. Even if you know the general technology in which your reference page resides, it's nice to find out where it is in the content structure. This is easy to fix: simply click the **Locate** button in the navigation toolbar, and all will be synchronized.

Use the Back button just like a browser. The **Back** button in the navigation toolbar functions just like a browser's Back button; if you need information on a reference page you viewed previously, you can use the **Back** button to get there, rather than going through the process of doing another search.

Define your own subsets, and use them. Like I said at the beginning of this chapter, the availability of information these days can sometimes make it difficult to get our work done. By defining subsets of MSDN that are tailored to the work you do, you can become more efficient.

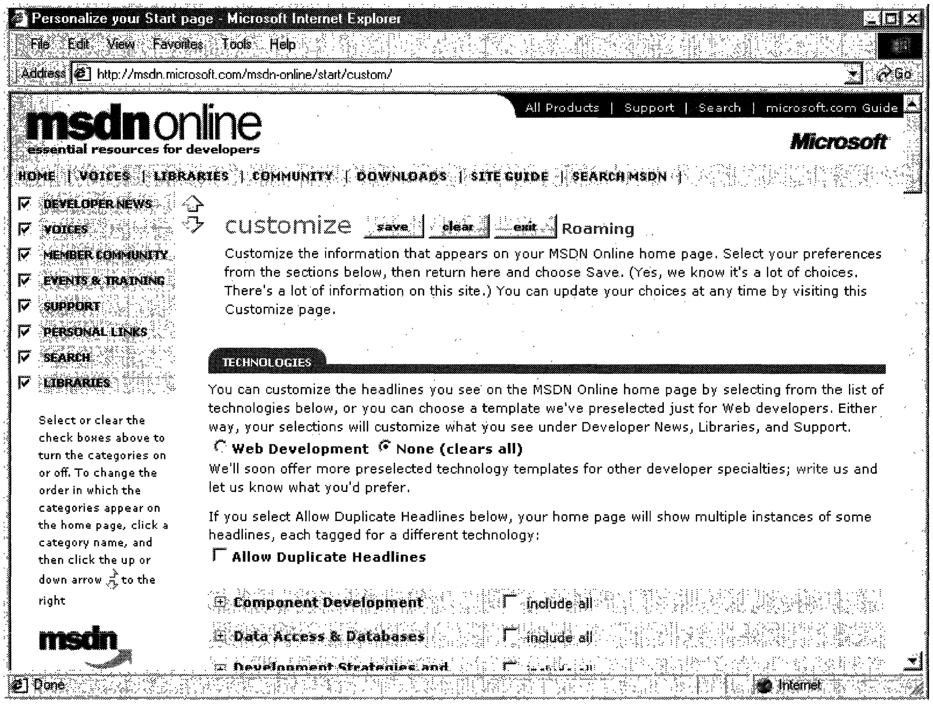
Use an underscore at the beginning of your named subsets. Subsets in the **Active Subset** drop-down box are arranged in alphabetical order, and the drop-down box shows only a few subsets at a time (making it difficult to get a grip on available subsets, I think). Underscores come before letters in alphabetical order, so if you use an underscore on all of your defined subsets, you get them placed at the front of the **Active Subset** listing of available subsets. Also, by using an underscore, you can immediately see which subsets you've defined, and which ones come with MSDN—it saves a few seconds at most, but those seconds can add up.

Using MSDN Online

MSDN Online shares a lot of similarities with MSDN, and that probably isn't by accident; when you can go from one developer resource to another and immediately be able to work with its content, your job is made easier. However, MSDN Online is different enough that it merits explaining in its own right...and it should be; it's a different delivery medium, and can take advantage of the Internet in ways that MSDN simply cannot.

If you've used Microsoft's home page before (*www.msn.com* or *home.microsoft.com*), you're familiar with the fact that you can customize the page to your liking; choose from an assortment of available national news, computer news, local news and weather, stock quotes, and other collections of information or news that suit your tastes or interests. You can even insert a few Web links and have them readily accessible when you visit the site. The MSDN Online home page can be customized in a similar way, but its collection of headlines, information, and news sources are all about development. The information you choose determines what information you see when you go to the MSDN Online home page, just like the Microsoft home page.

There are a couple of ways to get to the customization page; you can go to the MSDN Online home page (*msdn.microsoft.com*) and click the **Customize** button at the top of the page, or you can go there directly by pointing your browser to *msdn.microsoft.com/msdn-online/start/custom*. However you get there, the page you'll see is shown in Figure 3-5.


Personalize your Start page - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
All Products | Support | Search | microsoft.com Guide
msdn online
essential resources for developers
Microsoft
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS | SITE GUIDE | SEARCH MSDN |
customize
save
clear
exit
Roaming
TECHNOLOGIES
msdn
Done
Internet

Address: http://msdn.microsoft.com/msdn-online/start/custom/

☑ DEVELOPER NEWS
☑ VOICES
☑ MEMBER COMMUNITY
☑ EVENTS & TRAINING
☑ SUPPORT
☑ PERSONAL LINKS
☑ SEARCH
☑ LIBRARIES

Select or clear the check boxes above to turn the categories on or off. To change the order in which the categories appear on the home page, click a category name, and then click the up or down arrow to the right

Customize the information that appears on your MSDN Online home page. Select your preferences from the sections below, then return here and choose Save. (Yes, we know it's a lot of choices. There's a lot of information on this site.) You can update your choices at any time by visiting this Customize page.

You can customize the headlines you see on the MSDN Online home page by selecting from the list of technologies below, or you can choose a template we've preselected just for Web developers. Either way, your selections will customize what you see under Developer News, Libraries, and Support.

○ **Web Development** ● **None (clears all)**

We'll soon offer more preselected technology templates for other developer specialties; write us and let us know what you'd prefer.

If you select Allow Duplicate Headlines below, your home page will show multiple instances of some headlines, each tagged for a different technology:

□ **Allow Duplicate Headlines**

⊞ **Component Development** □ Include all
⊞ **Data Access & Databases** □ Include all
⊞ **Development Strategies and** □ Include all

Figure 3-5: The MSDN Online configuration page.

As you can see from Figure 3-5, there are lots of technologies to choose from. If you're interested in Web development, you can choose the option button near the top of the Technologies section for Web Development, and a pre-defined subset of Web-centric technologies is selected. For more Win32-centric technologies, you can choose the appropriate technologies. If you want to choose all the technologies in a given technology group, check the **Include All** box in the technology's shaded title area.

You can also choose which categories are included in the information MSDN Online presents to you, as well as their arranged order. The available categories include:

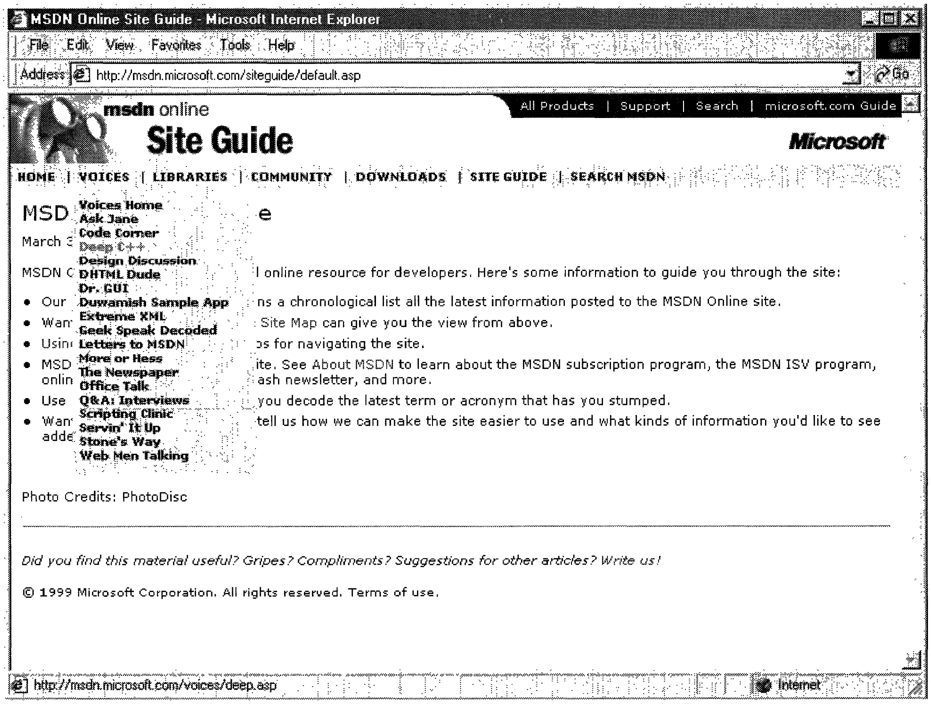
Developer News
Voices
Member Community
Events & Training
Support
Personal Links
Search
Libraries

Once you've defined your profile—that is, customized the MSDN Online content you want to see—MSDN Online shows you the most recent information pertinent to your profile when you go to MSDN Online's home page, with the categories you've chosen included in the order you specify. Note that clearing a given category—such as Libraries—clears that category from the body of your MSDN Online home page (and excludes headlines for that category), but does not remove that category from the MSDN Online site navigation bar. In other words, if you clear the category it won't be part of your customized MSDN Online page's headlines, but it'll still be available as a site feature.

Finally, if you want your profile to be available to you regardless of which computer you're using, you can direct MSDN Online to create a *roaming profile*. Creating a roaming profile for MSDN Online results in your profile being stored on MSDN Online's server, much like roaming profiles in Windows 2000, and thereby makes your profile available to you regardless of the computer you're using. The option of creating a roaming profile is available when you customize your MSDN Online home page (and can be done any time thereafter). The creation of a roaming profile, however, requires that you become a registered member of MSDN Online. More information about becoming a registered MSDN Online user is provided in the section titled *MSDN Online Registered Users*.

Navigating MSDN Online

Once you're done customizing the MSDN Online home page to get the headlines you're most interested in seeing, moving through MSDN Online is really easy. A banner that sits just below the MSDN Online logo functions as a navigation bar, with drop-down menus that can take you to the available areas on MSDN Online, as Figure 3-6 illustrates.



MSDN Online Site Guide - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/siteguide/default.asp
Go
msdn online
Site Guide
All Products | Support | Search | microsoft.com Guide
Microsoft
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS | SITE GUIDE | SEARCH MSDN
Voices Home
Ask Jane
Code Corner
Deep C++
Design Discussion
DHTML Dude
Dr. GUI
Duwamish Sample App
Extreme XML
Geek Speak Decoded
Letters to MSDN
More or Hess
The Newspaper
Office Talk
Q&A: Interviews
Scripting Clinic
Servin' It Up
Stone's Way
Web Men Talking
http://msdn.microsoft.com/voices/deep.asp
Internet

MSD e

March 3

MSDN C l online resource for developers. Here's some information to guide you through the site:

- Our ns a chronological list all the latest information posted to the MSDN Online site.
- Wan Site Map can give you the view from above.
- Usin ɔs for navigating the site.
- MSD ite. See About MSDN to learn about the MSDN subscription program, the MSDN ISV program, onlin ash newsletter, and more.
- Use you decode the latest term or acronym that has you stumped.
- Wan tell us how we can make the site easier to use and what kinds of information you'd like to see adde

Photo Credits: PhotoDisc

Did you find this material useful? Gripes? Compliments? Suggestions for other articles? Write us!

© 1999 Microsoft Corporation. All rights reserved. Terms of use.

Figure 3-6: The MSDN Online navigation bar with its drop-down menus.

The list of available menu categories—which group the available sites and features within MSDN Online—includes:

Home

Voices

Voices

Libraries

Community

Downloads

Site Guide

Search MSDN

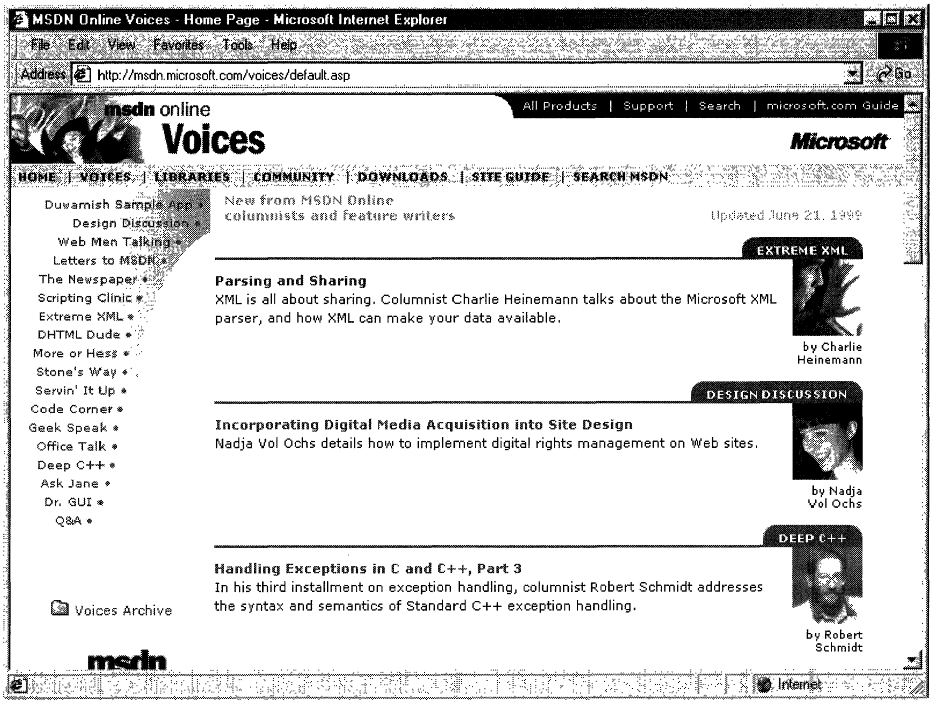
The navigation bar is available regardless of where you are in MSDN Online, so the capability to explore the site from this familiar menu is always available, leaving you a click away from any area on MSDN Online. These menu categories create a functional and logical grouping of MSDN Online's feature offerings.

MSDN Online Features

Each of MSDN Online's seven feature categories contains various sites that comprise the features available to developers visiting MSDN Online.

Home is already familiar; clicking on **Home** in the navigation bar takes you to the MSDN Online home page that you've (perhaps) customized, showing you all the latest headlines for technologies that you've indicated you're interested in reading about.

Voices is a collection of columns and articles that comprise MSDN Online's magazine section, and can be linked to directly at *msdn.microsoft.com/voices*. The Voices home page is shown in Figure 3-7.



http://msdn.microsoft.com/voices/default.asp

New from MSDN Online
columnists and feature writers

Updated June 21, 1999

Parsing and Sharing
XML is all about sharing. Columnist Charlie Heinemann talks about the Microsoft XML parser, and how XML can make your data available.

by Charlie Heinemann

Incorporating Digital Media Acquisition into Site Design
Nadja Vol Ochs details how to implement digital rights management on Web sites.

by Nadja Vol Ochs

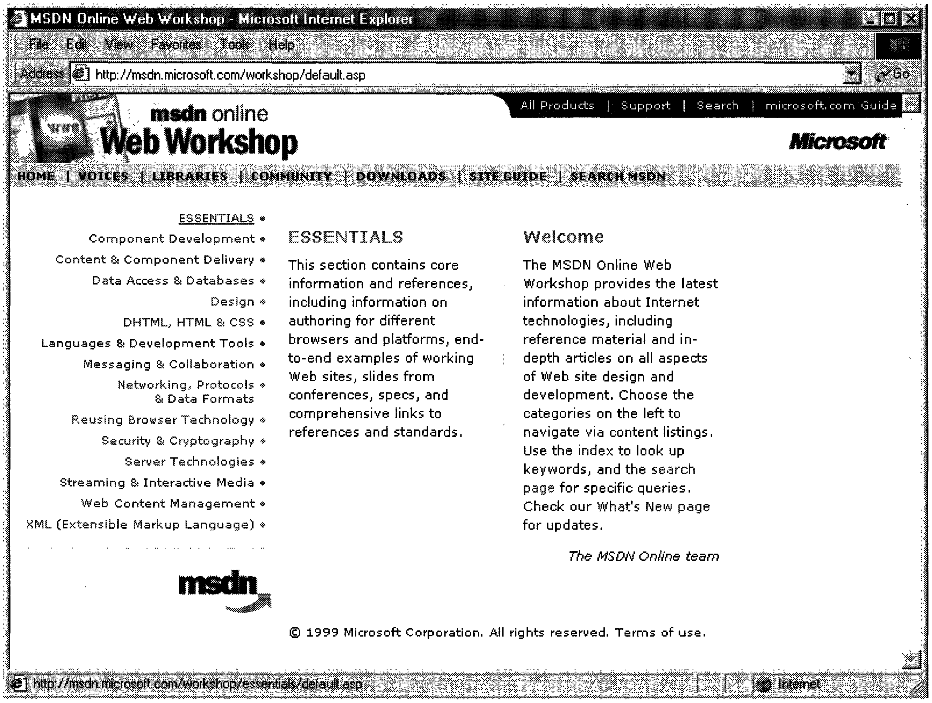
Handling Exceptions in C and C++, Part 3
In his third installment on exception handling, columnist Robert Schmidt addresses the syntax and semantics of Standard C++ exception handling.

by Robert Schmidt

Figure 3-7: The Voices home page.

Each of thet "voices" in the Voices site, adds its own particular twist on the issues that face developers. Both application and Web developers can get their fill of magazine-like articles from the sizable list of different articles available (and frequently refreshed) in the Voices site.

Libraries is where the reference material available on MSDN Online lives. The Libraries site is divided into two sections: **Library** and **Web Workshop**. This distinction divides the reference material between what used to be MSDN and Site Builder Network; that is, Windows application development and Web development. Choosing **Library** from the **Libraries** menu takes you to a page you can explore in traditional MSDN fashion and gain access to traditional MSDN reference material; the Library home page can be linked to directly at *msdn.microsoft.com/library*. Choosing **Web Workshop** takes you to a site that enables you to explore the Web Workshop in a slightly different way, starting with a bulleted list of start points, as shown in Figure 3-8. The Web Workshop home page can be linked to directly at *msdn.microsoft.com/workshop*.


MSDN Online Web Workshop - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/workshop/default.asp
Go
All Products | Support | Search | microsoft.com Guide
msdn online
Web Workshop
Microsoft
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS | SITE GUIDE | SEARCH MSDN
ESSENTIALS •
Component Development •
Content & Component Delivery •
Data Access & Databases •
Design •
DHTML, HTML & CSS •
Languages & Development Tools •
Messaging & Collaboration •
Networking, Protocols & Data Formats •
Reusing Browser Technology •
Security & Cryptography •
Server Technologies •
Streaming & Interactive Media •
Web Content Management •
XML (Extensible Markup Language) •
msdn
http://msdn.microsoft.com/workshop/essentials/default.asp
Internet

ESSENTIALS

This section contains core information and references, including information on authoring for different browsers and platforms, end-to-end examples of working Web sites, slides from conferences, specs, and comprehensive links to references and standards.

Welcome

The MSDN Online Web Workshop provides the latest information about Internet technologies, including reference material and in-depth articles on all aspects of Web site design and development. Choose the categories on the left to navigate via content listings. Use the index to look up keywords, and the search page for specific queries. Check our What's New page for updates.

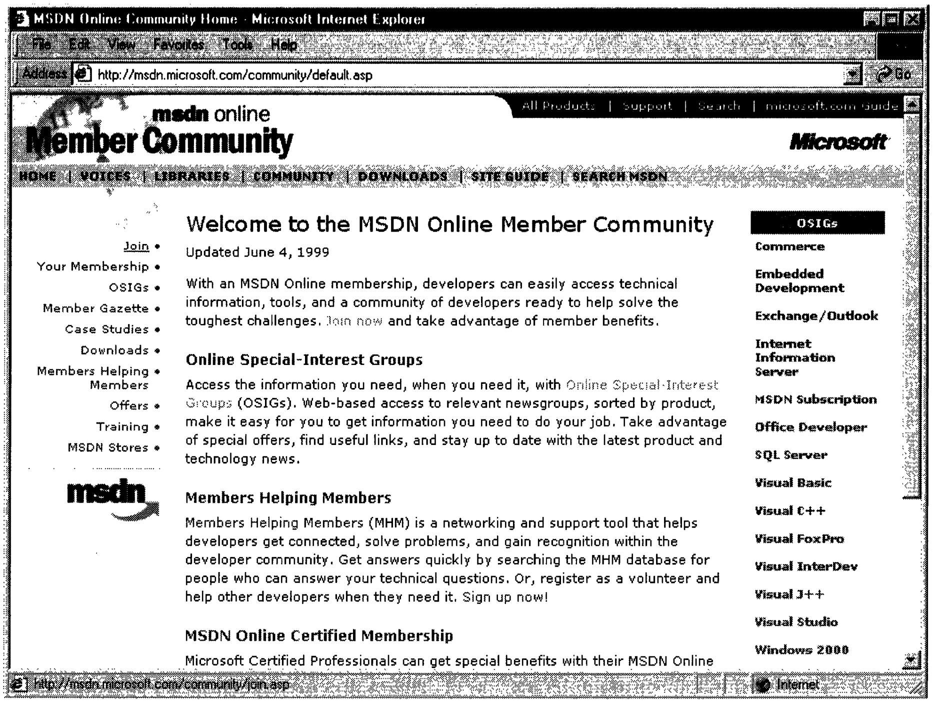
The MSDN Online team

© 1999 Microsoft Corporation. All rights reserved. Terms of use.

Figure 3-8: The Web Workshop home page, with its bulleted list of navigation start points.

Community is a place where developers can go to take advantage of the online forum of Windows and Web developers, in which ideas or techniques can be shared, advice can be found or given (through MHM, or Members Helping Members), and Online Special Interest Groups (OSIGs) can find a forum to voice their opinions or chat with other developers. The Community site is full of all sorts of useful stuff, including featured books, promotions and downloads, case studies, and more. The Community home page can be linked to directly at *msdn.microsoft.com/community*. Figure 3-9 provides a look at the Community home page.

The **Downloads** site is where developers can find all sorts of useful items fit to be downloaded, such as tools, samples, images, and sounds. The Downloads site is also where MSDN subscribers go to get their subscription content updated over the Internet to the latest and greatest releases, as described previously in this chapter in the *Using MSDN* section. The Downloads home page can be linked to directly at *msdn.microsoft.com/downloads*. The Downloads home page is shown in Figure 3-10.


MSDN Online Community Home - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/community/default.asp
Go
All Products | Support | Search | microsoft.com Guide
msdn online
Member Community
Microsoft
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS | SITE GUIDE | SEARCH MSDN
Join •
Your Membership •
OSIGs •
Member Gazette •
Case Studies •
Downloads •
Members Helping Members •
Offers •
Training •
MSDN Stores •
msdn
OSIGs
Commerce
Embedded Development
Exchange/Outlook
Internet Information Server
MSDN Subscription
Office Developer
SQL Server
Visual Basic
Visual C++
Visual FoxPro
Visual InterDev
Visual J++
Visual Studio
Windows 2000
http://msdn.microsoft.com/community/join.asp
Internet

Welcome to the MSDN Online Member Community

Updated June 4, 1999

With an MSDN Online membership, developers can easily access technical information, tools, and a community of developers ready to help solve the toughest challenges. Join now and take advantage of member benefits.

Online Special-Interest Groups

Access the information you need, when you need it, with Online Special-Interest Groups (OSIGs). Web-based access to relevant newsgroups, sorted by product, make it easy for you to get information you need to do your job. Take advantage of special offers, find useful links, and stay up to date with the latest product and technology news.

Members Helping Members

Members Helping Members (MHM) is a networking and support tool that helps developers get connected, solve problems, and gain recognition within the developer community. Get answers quickly by searching the MHM database for people who can answer your technical questions. Or, register as a volunteer and help other developers when they need it. Sign up now!

MSDN Online Certified Membership

Microsoft Certified Professionals can get special benefits with their MSDN Online

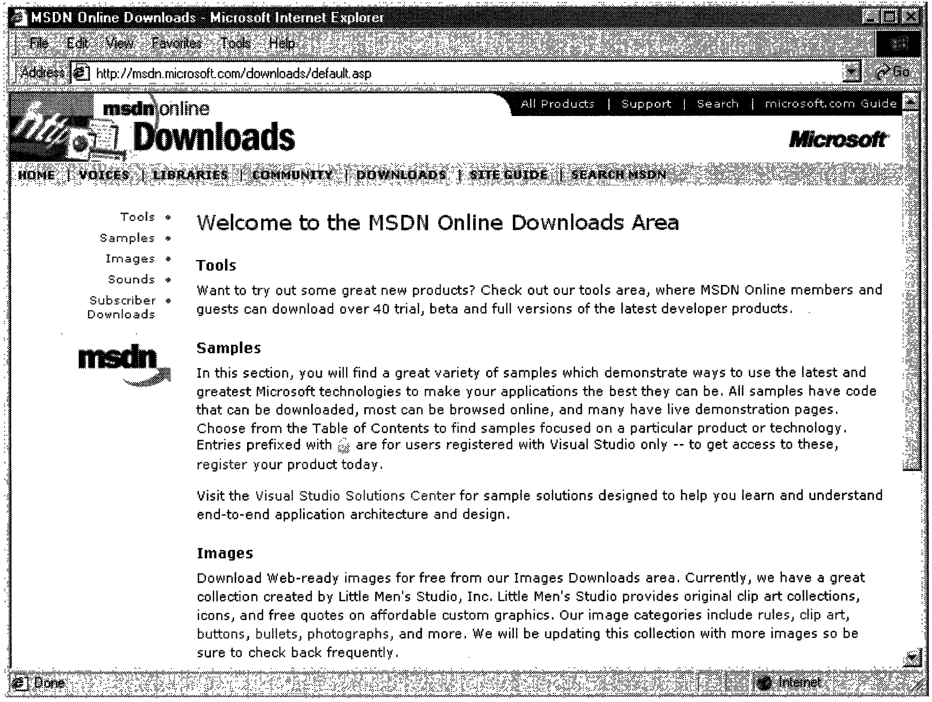
Figure 3-9: The Community home page.

The **Site Guide** is just what its name suggests; a guide to the MSDN Online site that aims at helping developers find items of interest, and includes links to other pages on MSDN Online such as a recently posted files listing, site maps, glossaries, and other useful links. The Site Guide home page can be linked to directly at *msdn.microsoft.com/siteguide*.

The **Search MSDN** site on MSDN Online has been improved over previous versions, and includes the capability to restrict searches to either library (**Library** or **Web Workshop**), as well as other finely-tuned search capabilities. The Search MSDN home page can be linked to directly at *msdn.microsoft.com/search*. The Search MSDN home page is shown in Figure 3-11.

MSDN Online Registered Users

You may have noticed that some features of MSDN Online—such as the capability to create a roaming profile of the entry ticket to some community features—require you to become a registered user. Unlike MSDN subscriptions, becoming a registered user of MSDN Online won't cost you anything more than a few minutes of registration time.


MSDN Online Downloads - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://msdn.microsoft.com/downloads/default.asp
Go
msdn online
Downloads
All Products | Support | Search | microsoft.com Guide
Microsoft
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS | SITE GUIDE | SEARCH MSDN
Tools
Samples
Images
Sounds
Subscriber Downloads
msdn
Done
Internet

Welcome to the MSDN Online Downloads Area

Tools

Want to try out some great new products? Check out our tools area, where MSDN Online members and guests can download over 40 trial, beta and full versions of the latest developer products.

Samples

In this section, you will find a great variety of samples which demonstrate ways to use the latest and greatest Microsoft technologies to make your applications the best they can be. All samples have code that can be downloaded, most can be browsed online, and many have live demonstration pages. Choose from the Table of Contents to find samples focused on a particular product or technology. Entries prefixed with are for users registered with Visual Studio only -- to get access to these, register your product today.

Visit the Visual Studio Solutions Center for sample solutions designed to help you learn and understand end-to-end application architecture and design.

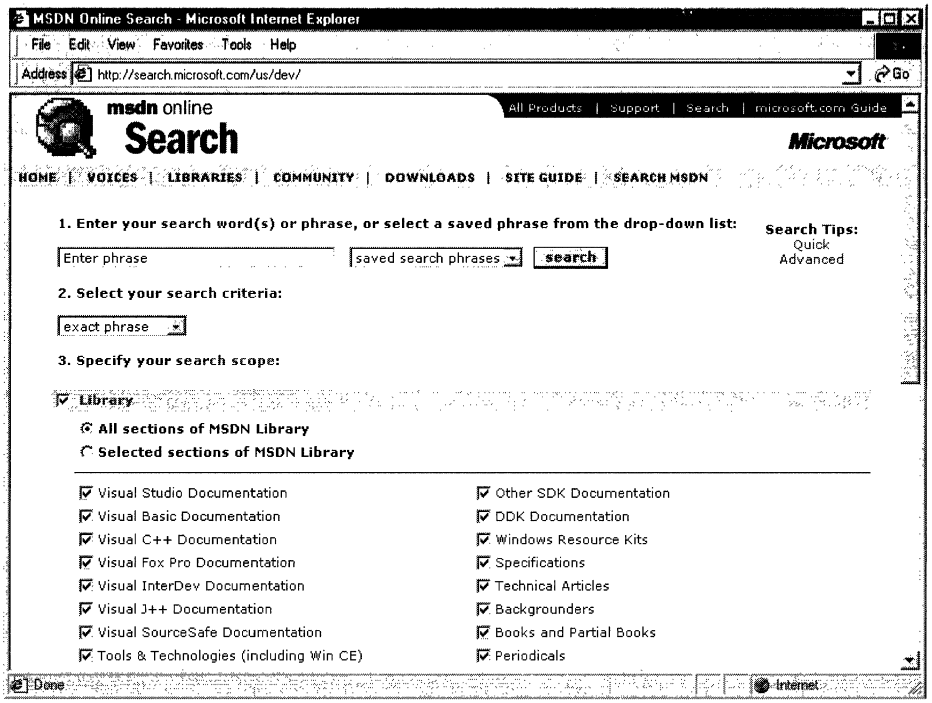
Images

Download Web-ready images for free from our Images Downloads area. Currently, we have a great collection created by Little Men's Studio, Inc. Little Men's Studio provides original clip art collections, icons, and free quotes on affordable custom graphics. Our image categories include rules, clip art, buttons, bullets, photographs, and more. We will be updating this collection with more images so be sure to check back frequently.

Figure 3-10: The Downloads home page.

Some features of MSDN Online require registration before you can take advantage of their offerings. For example, becoming a member of an Online Special Interest Group (OSIG) requires registration. That feature alone is reason enough to register. Rather than attempting to call your developer buddy for an answer to a question (only to find out that she's on vacation for two days, and your deadline is in a few hours), you can go to MSDN Online's Community site and ferret through your OSIG to find the answer in a handful of clicks. Who knows; maybe your developer buddy will begin calling you with questions—you don't have to tell her where you're getting all your answers.

There are actually a number of advantages to being a registered user, such as the choice to receive newsletters right in your inbox—if you want to. You can also get all sorts of other timely information, such as chat reminders that let you know when experts on a given subject will be chatting in the MSDN Online Community site. You can also sign up to get newsletters based on your membership in various OSIGs—again, only if you want to. It's easy for me to suggest that you become a registered user for MSDN Online—I'm a registered user, and it's a great resource.


MSDN Online Search - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address
Go
msdn online
Search
All Products | Support | Search | microsoft.com Guide
Microsoft
HOME | VOICES | LIBRARIES | COMMUNITY | DOWNLOADS | SITE GUIDE | SEARCH MSDN
Search Tips:
Quick
Advanced
search
Done
Internet

http://search.microsoft.com/us/dev/

1. Enter your search word(s) or phrase, or select a saved phrase from the drop-down list:

Enter phrase | saved search phrases

2. Select your search criteria:

exact phrase

3. Specify your search scope:

☑ Library

● All sections of MSDN Library

○ Selected sections of MSDN Library

☑ Visual Studio Documentation
☑ Visual Basic Documentation
☑ Visual C++ Documentation
☑ Visual Fox Pro Documentation
☑ Visual InterDev Documentation
☑ Visual J++ Documentation
☑ Visual SourceSafe Documentation
☑ Tools & Technologies (including Win CE)
☑ Other SDK Documentation
☑ DDK Documentation
☑ Windows Resource Kits
☑ Specifications
☑ Technical Articles
☑ Backgrounders
☑ Books and Partial Books
☑ Periodicals

Figure 3-11: The Search MSDN home page.

The Windows Programming Reference Series

The Windows Programming Reference Series provides developers with timely, concise, and focused material on a given topic, enabling developers to get their work done as efficiently as possible. In addition to providing reference material for Microsoft technologies, each Library in the Windows Programming Reference Series also includes material that helps developers get the most out of its technologies, and provides insights that might otherwise be difficult to find.

The Windows Programming Reference Series is currently planned to include the following libraries:

Win32 Library
Active Directory Library
Networking Services Library

CHAPTER 4

Finding the Developer Resources You Need

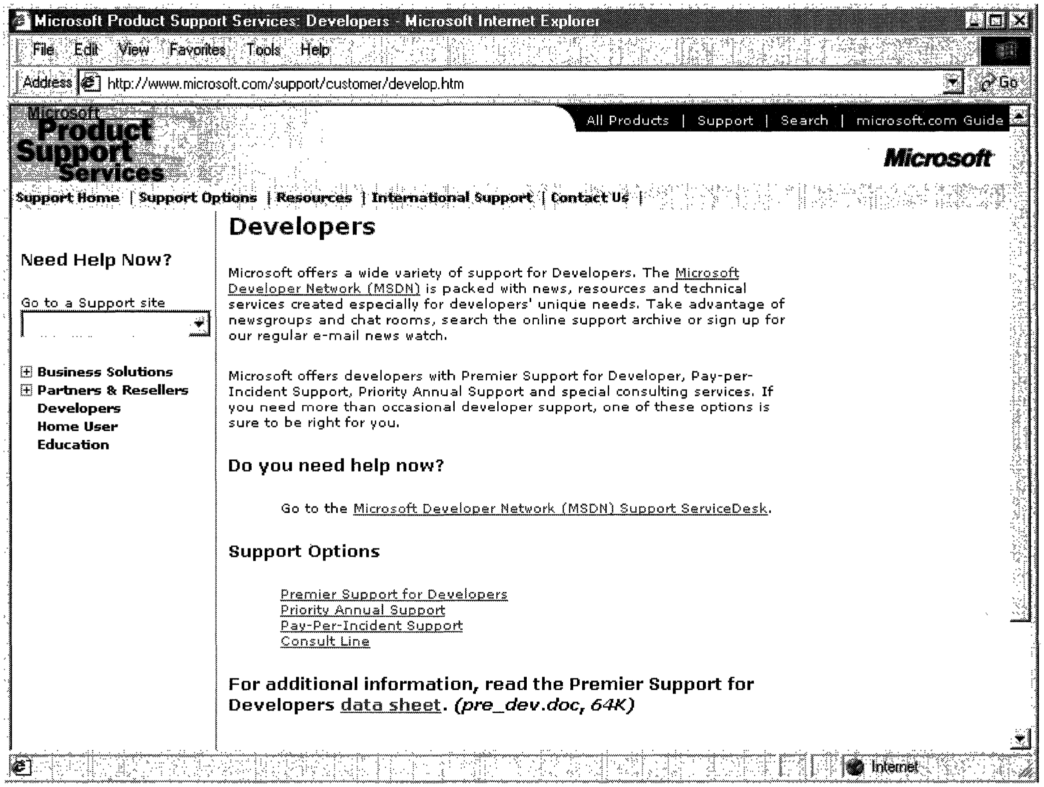
There are all sorts of resources out there for developers of Windows applications, and they can provide answers to a multitude of questions or problems that developers face every day, but finding those resources is sometimes harder than the original problem. This chapter aims to provide you with a one-stop resource to find as many developer resources as are available, again making your job of actually developing the application just a little easier.

While Microsoft provides lots of resource material through MSDN and MSDN Online, and although the Windows Programming Resource Series provides lots of focused reference material and development tips and tricks, there is a *lot* more information to be had. Some of it is from Microsoft, some from the general development community, and some from companies that specialize in such development services. Regardless of which resource you choose, in this chapter you can find out what your development resource options are and, therefore, be more informed about the resources that are available to you.

Microsoft provides developer resources through a number of different media, channels, and approaches. The extensiveness of Microsoft's resource offerings mirrors the fact that many are appropriate under various circumstances. For example, you wouldn't go to a conference to find the answer to a specific development problem in your programming project; instead, you might use one of the other Microsoft resources.

Developer Support

Microsoft's support sites cover a wide variety of support issues and approaches, including all of Microsoft's products, but most of those sites are not pertinent to developers. Some sites, however, *are* designed for developer support; the Product Services Support page for developers is a good central place to find the support information you need. Figure 4-1 shows the Product Services Support page for developers, which can be found at *www.microsoft.com/support/customer/develop.htm*.


Microsoft Product Support Services: Developers - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.microsoft.com/support/customer/develop.htm
Go
Microsoft Product Support Services
All Products | Support | Search | microsoft.com Guide
Microsoft
Support Home | Support Options | Resources | International Support | Contact Us |
Need Help Now?
Go to a Support site
Business Solutions
Partners & Resellers
Developers
Home User
Education
Internet

Developers

Microsoft offers a wide variety of support for Developers. The Microsoft Developer Network (MSDN) is packed with news, resources and technical services created especially for developers' unique needs. Take advantage of newsgroups and chat rooms, search the online support archive or sign up for our regular e-mail news watch.

Microsoft offers developers with Premier Support for Developer, Pay-per-Incident Support, Priority Annual Support and special consulting services. If you need more than occasional developer support, one of these options is sure to be right for you.

Do you need help now?

Go to the Microsoft Developer Network (MSDN) Support ServiceDesk.

Support Options

Premier Support for Developers
Priority Annual Support
Pay-Per-Incident Support
Consult Line

For additional information, read the Premier Support for Developers data sheet. ***(pre_dev.doc, 64K)***

Figure 4-1: The Product Services Support page for developers.

Note that there are a number of options for support from Microsoft, including everything from simple online searches of known bugs in the Knowledge Base to hands-on consulting support from Microsoft Consulting Services, and everything in between. The Web page displayed in Figure 4-1 is a good starting point from which you can find out more information about Microsoft's support services.

Premier Support from Microsoft provides extensive support for developers, and there are different packages geared toward different Microsoft customers. The packages of Premier Support that Microsoft provides are:

- Premier Support for Enterprises
- Premier Support for Developers
- Premier Support for Microsoft Certified Solution Providers
- Premier Support for OEMs

If you're a developer, you might fall into any of these categories. To find out more information about Microsoft's Premier Support, get in contact with them at 1-800-936-2000.

Priority Annual Support from Microsoft is geared toward developers or organizations that have more than an occasional need to call Microsoft with support questions, and need priority handling of their support questions or issues. There are three packages of Priority Annual Support offered by Microsoft:

- Priority Comprehensive Support
- Priority Developer Support
- Priority Desktop Support

As a developer, the best support option for you is the Priority Developer Support. To get more information about Priority Developer Support, you can reach Microsoft at 1-800-936-3500.

Microsoft also offers a **Pay-Per-Incident** support option, so you can get help if there's just one question for which you must have an answer. With Pay-Per-Incident support, you call a toll-free number and provide your Visa, MasterCard, or American Express card number, after which you receive support for your incident. In loose terms, an incident is some problem or issue that can't be broken down into sub-issues or sub-problems (that is, it can't be broken down into smaller pieces). The number to call for Pay-Per-Incident support is 1-800-936-5800.

Note that Microsoft provides two priority technical support incidents as part of the MSDN Professional Subscription, and provides four priority technical support incidents as part of the MSDN Universal Subscription.

You can also **submit questions** to Microsoft engineers through Microsoft's support Web site, but if you're on a deadline you might want to rethink this approach, or consider going to MSDN Online and looking into the Community site there for help with your development question. To submit a question to Microsoft engineers online, go to *support.microsoft.com/support/webresponse.asp*.

Online Resources

Microsoft also provides extensive developer support through its community of developers found on MSDN Online. At MSDN Online's Community site, you will find OSIGs that cover all sorts of issues in an online, ongoing fashion. To get to MSDN Online's Community site, go to *msdn.microsoft.com/community*.

Microsoft's MSDN Online also provides its **Knowledge Base** online, which is part of the Personal Support Center on Microsoft's corporate site. You can search the Knowledge Base online at *support.microsoft.com/support/search*.

Microsoft provides a number of **newsgroups** that developers can use to view information on newsgroup-specific topics, providing yet another developer resource for finding information about creating Windows applications. To find out which newsgroups are available, and how to get to them, go to *support.microsoft.com/support/news*.

There is a handful of newsgroups that will probably be of particular interest to readers of the *Microsoft Win32 Developer's Reference Library*, and they are the following:

*microsoft.public.win32.programmer.**
*microsoft.public.vc.**
*microsoft.public.vb.**
*microsoft.public.platformsdk.**
*microsoft.public.cert.**
*microsoft.public.certification.**

Of course, Microsoft isn't the only newsgroup provider on which newsgroups pertaining to Windows development are hosted. Usenet has all sorts of newsgroups—too many to list—that host ongoing discussions pertaining to developing applications on the Windows platform. You can access newsgroups on Windows development just as you access any other newsgroup; generally, you'll need to contact your ISP to find out the name of the mail server, and then use a newsreader application to visit, read, or post to the Usenet groups.

Learning Products

Microsoft provides a number of products that help enable developers to learn the particular tasks or tools that they need to achieve their goals (or to finish their tasks). One product line that is geared toward developers is called the **Mastering Series**, and its products provide comprehensive, well-structured, interactive teaching tools for a wide variety of development topics.

The Mastering Series from Microsoft consists of interactive tools that group books and CDs together so that you can master the topic in question. To get more information about the Mastering Series of products, or to find out what kind of offerings the Mastering Series has, check out *msdn.microsoft.com/mastering*.

Other learning products are available from other vendors, too, such as other publishers, other applications providers that create tutorial-type content and applications, and companies that issue videos (both taped and broadcast over the Internet) on specific technologies. For one example of a company that issues technology-based instructional or overview videos, take a look at *www.compchannel.com*.

Another way of learning about development in a particular language (such as Visual C++, Visual FoxPro, or Visual Basic), for a particular operating system, or for a particular product (such as SQL Server or Commerce Server) is to go through and read the preparation materials available to get certified as a Microsoft Certified Solution Developer (MCSD). Before you get too defensive about not having enough time to get certified, or in having no interest in getting your certification (maybe you do—there *are* benefits, you know), let me state that the point of the journey is not necessarily to arrive. In other words, you don't have to get your certification for the preparation materials to be useful; in fact, they might teach you things that you thought you knew well, but actually didn't know as well as you thought you did. The fact of the matter is that the coursework and the requirements to get through the certification process are rigorous, difficult, and quite detail-oriented. If you have what it takes to get your certification, you have an extremely strong grasp on the fundamentals (and then some) of application programming and the developer-oriented information about Windows platforms.

You are required to take a set of core exams to get an MCSD certification, and then you must choose one topic from many available elective exams to complete your certification requirements. Core exams are chosen from among a group of available exams; you must pass a total of three exams to complete the core requirements. There are "tracks" that candidates generally choose and that point their certification in a given direction, such as Visual C++ development or Visual Basic development. The core exams and their exam numbers are as follows.

Desktop Applications Development (one required):

- Designing and Implementing Desktop Applications with Microsoft Visual C++ 6.0 (70-016)
- Designing and Implementing Desktop Applications with Microsoft Visual FoxPro 6.0 (70-155)
- Designing and Implementing Desktop Applications with Microsoft Visual Basic 6.0 (70-176)

Distributed Applications Development (one required):

- Designing and Implementing Distributed Applications with Microsoft Visual C++ 6.0 (70-015)
- Designing and Implementing Distributed Applications with Microsoft Visual FoxPro 6.0 (70-156)
- Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0 (70-175)

Solutions Architecture:

- Analyzing Requirements and Defining Solution Architectures (70-100)

Elective exams enable candidates to choose from a number of additional exams to complete their MCSD exam requirements. The following lists the available MCSD elective exams.

Available elective exams:

- Any Desktop or Distributed exam not used as a core requirement
- Designing and Implementing Data Warehouses with Microsoft SQL Server 7.0 and Microsoft Decision Support Services 1.0
- Developing Applications with C++ Using the Microsoft Foundation Class Library 4.0 Library
- Implementing OLE in Microsoft Foundation Class Library 4.0 Applications

- Implementing a Database Design on Microsoft SQL Server 6.5
- Designing and Implementing Databases with Microsoft SQL Server 7.0
- Designing and Implementing Web Sites with Microsoft FrontPage 98
- Designing and Implementing Commerce Solutions with Microsoft Site Server 3.0, Commerce Edition
- Microsoft Access for Windows 95 and the Microsoft Access Developer's Toolkit
- Designing and Implementing Solutions with Microsoft Office 2000 and Microsoft Visual Basic for Applications
- Designing and Implementing Database Applications with Microsoft Access 2000
- Designing and Implementing Collaborative Solutions with Microsoft Outlook 2000 and Microsoft Exchange Server 5.5
- Designing and Implementing Web Solutions with Microsoft Visual InterDev 6.0
- Designing and Implementing Distributed Applications with Microsoft Visual FoxPro 6.0
- Designing and Implementing Desktop Applications with Microsoft Visual FoxPro 6.0
- Developing Applications with Microsoft Visual Basic 5.0
- Designing and Implementing Distributed Applications with Microsoft Visual Basic 6.0
- Designing and Implementing Desktop Applications with Microsoft Visual Basic 6.0

The best news about these exams isn't that there are lots from which to choose. The best news is that, because there are exams that must be passed to become certified, there are books and other materials out there to *teach you* how to meet the knowledge level necessary to pass the exams, and that means those resources are available to you—regardless of whether you care one whit about becoming an MCSD or not.

The way to leverage this information is to get study materials for one or more of these exams—and don't be fooled by believing that if the book is bigger it must be better, because that certainly isn't always the case—and go through the exam preparation material. Such exam preparation material is available from all sorts of publishers, including Microsoft Press, IDG, Sybex, and others. Most exam preparation texts also have practice exams that let you self-assess your grasp of the material. You might be surprised by how much you learn, even though you might have been in the field working on complex projects for some time.

Of course, these exam requirements, and the exams themselves, can change over time; more electives become available, exams based on revised versions of software are retired, and so on. For more information about the certification process, or for more information about the exams, check out *www.microsoft.com/train_cert/dev*.

Conferences

As in any industry, Microsoft and the development community as a whole sponsor conferences throughout the year—occurring throughout the country and around the world—on various topics. There are probably more conferences available than any brhuman being could possibly attend and still be sane, but often a given conference is geared toward a particular topic, so choosing to focus on a given development topic enables developers to select the number of conferences that apply to their efforts and interests.

MSDN itself hosts or sponsors almost a hundred conferences a year (some of them are regional and duplicated in different locations, so these could be considered one conference that happens multiple times). Other conferences are held in one central location, such as the big one—the Professional Developers Conference (PDC). Regardless of which conference you're looking for, Microsoft has provided a central site for providing event information, and enables users (such as yourself) to search the site for conferences, based on many different criteria. To find out what conferences or other events are going on in your area of interest of development focus, go to *events.microsoft.com.*

Other Resources

There are other resources available for developers of Windows applications, some of which might be mainstays for one developer and unheard of for another. The listing of developer resources in this chapter has been geared toward getting you more than started with finding the developer resources you need: it's geared toward getting you 100 percent of the way, but there are always exceptions.

Perhaps you're just getting started, and you want to get more hands-on instruction than MSDN Online or MCSD preparation materials provide. Where can you go? One option is to check out your local college for instructor-led courses. Most community colleges offer night classes, in case you have that pesky day job with which to contend and, increasingly, community colleges are outfitted with rather nice computer labs that enable you to get hands-on development instruction and experience, without having to work on a 386/20.

There are undoubtedly other resources that some people know about that have been useful, or maybe invaluable. If you have a resource that should be shared with others, let me know about it by sending me e-mail at the following address, and—who knows?—maybe someone else will benefit from your knowledge:

wprs@microsoft.com

If you're sending e-mail about a particularly useful resource, type "Resources" in the subject line. There aren't any guarantees that you'll get a reply, but I'll read all of the e-mail and do what I can to ensure your resource idea gets considered.

CHAPTER 5

Getting the Most Out of Win32 Technologies: Part 3

This chapter is the third of the five-part collection of common programming errors included in the Win32 Library to help you avoid these simple programming pitfalls. This collection of common programming errors is distributed in each Win32 Library volume's Chapter 5 in the following fashion:

Volume 1: Overview and Solution Summary

Volume 2: Avoiding Invalid Validations

Volume 3: RPC Errors and Kernel-Mode Specifiers

Volume 4: Buffer Overflows and Miscellaneous Errors

Volume 5: Memory Abuse and Miscalculations

As you'll notice, not all of these pitfalls are necessarily confined to Win32 programming (some are networking services based, for example). However, since these common coding errors must be avoided in any Windows application, they're provided here in their entirety to round out the benefits of owning the Win32 Library.

This, of course, is Volume 3, and the errors and examples found in this chapter provide insights that can help you avoid problems with RPC errors and kernel-mode specifiers in your programming projects. So, without further ado, here they are!

RPC Errors

The use of RPC requires that programmers be aware of a number of issues that can cause errors or expose their applications to various attacks:

- Check **unique** pointers for **NULL** before dereferencing.
- When using a switch_is construct that has a default clause:
  - verify that the value switch is within expected range.
  - verify that pointers within the switched object are not null before dereferencing them.
- Don't use **NULL** DACLs; they don't protect anything.
- Impersonate before acting on behalf of the caller, and check the result.
- Stop impersonating when finished acting on behalf of the caller, and check

the result.

- Don't expect strings to be zero terminated unless **string** is specified in the .idl file.
- Don't copy arbitrary length data into independently-sized buffers.
- Check length of **size_is** specified data before dereferencing corresponding pointers.
- Be aware that calculations in midl definitions using **size_is** and **length_is** can overflow, and that it may be impossible for the server to detect this.
- Use strict context handles.

Using pointer_default(unique) and embedded pointers

When an RPC structure contains pointers, its pointers default to the default pointer type (typically set by **pointer_default(unique)**). Under such circumstances, **unique** pointers can be **NULL** and must be verified to be non-**NULL** before being dereferenced.

Example

```
[
    pointer_default(unique)
]

typedef struct _RPC_STRUCTURE {
    INSTANCE_DATA *Instance;
} RPC_STRUCTURE;

NTSTATUS
RpcInterface(
    [in] RPC_STRUCTURE *s
)
{
    INSTANCE_DATA *i;

    i = s -> Instance;

    if (*i) {  // error:  i may be NULL.
        [...]
```

A valid switch_is value in an RPC-capable structure doesn't ensure a non-NULL pointer

A valid value for the **switch** field does not change the default of embedded pointers from **unique**. Thus, even when it's valid, the pointer must still be verified to be non-**NULL** before being dereferenced.

Example

```
typedef struct _rpc_structure {
    ULONG type;
    [switch_is(type)] union {
        [case(1)]  INSTANCE_DATA *Instance;
    } field;
} RPC_STRUCTURE;

NTSTATUS
RpcInterface(
    [in] RPC_STRUCTURE *s
)
{
    INSTANCE_DATA *i;

    switch (s -> type) {
        case 1:
            i = s->Field.Instance;

            if (*i) {  // error:  i may be NULL.
                […]
```

A NULL DACL affords no protection

A NULL DACL grants access to everyone and protects nothing; it doesn't even protect an object from having its DACL changed to deny access to everyone. In general, an untrusted user should not be granted access to change a security-descriptor's Owner or DACL fields (unless they own the object, in which case no one else should be granted such access).

Example

```
InitializeSecurityDescriptor(pSecurityDescriptor,
                            SECURITY_DESCRIPTOR_REVISION);

fBool = SetSecurityDescriptorDacl(pSecurityDescriptor,
                                    TRUE,   // Dacl present
                                    NULL,   // NULL Dacl
                                    FALSE); // Not defaulted
if (fBool) {
    RpcStatus = RpcServerUseProtseqEp(TEXT("ncacn_np"),
                                    10, // max concurrent calls
                                    "PipeName",
                                    pSecurityDescriptor);
```

Remarks

This example exposes this error for RPC, but the error's scope goes beyond RPC. If you create a publicly accessible securable object and don't secure it against unauthorized users changing the DACL, *anyone* can lock the object such that *no one* can access it.

Allowing "all" access—for example, applying a DACL granting **EVENT_ALL_ACCESS** to everyone who accesses an event object—is an equally bad idea, because "all" access typically grants **WRITE_DAC** and **WRITE_OWNER** permissions. Granting either of these permissions explicitly also enables objects to be locked up. Use (**GENERIC_READ | GENERIC_WRITE | GENERIC_EXECUTE**) when it's necessary to grant broad access to an object to any non-administrative-level user.

Call RpcImpersonateClient() before any security-relevant operation

The purpose of many RPC servers is to act on behalf of a client, but they must protect system integrity while doing so. Many RPC servers run in the system context; impersonating the caller enables the server to use the user's credentials to access some objects, while otherwise being a part of the secure side of the system.

Example

```
BOOLEAN
RpcInterface(
    [in] ULONG Pid
)
{
    HANDLE Process;

    Process = OpenProcess(PROCESS_VM_WRITE,
                          FALSE,
                          Pid);
```

Remarks

Opening a process by pid *without* first impersonating can provide a caller with access to the process that it normally wouldn't have. The server now has a handle to a process—LSASS for example—allowing it to scribble in the address of a process the user would not have been allowed on its own.

Starting and stopping impersonation

There are a handful of issues that programmers should be on the lookout for when starting and/or stopping impersonation.

Always check the result of RpcImpersonateClient() before a security-relevant operation

The **RpcImpersonateClient()** function returns an indication of success or failure; skip the check and you may as well have skipped the call (which, as we saw previously in this chapter, can be dangerous).

Call RpcRevertToSelf() after security-relevant operations

Once a server has acted on behalf of the user by impersonating, it should revert to its own security context by calling **RpcRevertToSelf()**. Although the consequences of failing to undo impersonation are typically not as drastic as failing to impersonate, it can result in failure to function correctly, and cause spurious behavior such as extra audits.

Example (a *correct* example for once)

```
BOOLEAN
RpcFunction(
    [in] ULONG pid
)
{
    HANDLE Process;
    RPC_STATUS Status;

    Status = RpcImpersonateClient(NULL);

    if (Status != RPC_S_OK) {
        return FALSE;
    }

    Process = OpenProcess(PROCESS_VM_WRITE,
                          FALSE,
                              Pid);

    […]

    if (Process) {
        CloseHandle(Process);
    }

    if (RpcRevertToSelf() != RPC_S_OK) {
        // Something bad happened, might need
        // to do more than just clean up this call.

        return FALSE;
    }
    return TRUE;
}
```

Remarks

This example shows how to avoid this programming error in RPC, the scope of this error extends beyond RPC. Impersonation is possible over LPC, Named Pipes, and when using Tokens. In all cases, a decision must be made as to whose context (typically System versus untrusted user) should be used for various operations, and impersonation used where appropriate.

Strings are zero-terminated only when declared with strings in the .idl

Variably sized RPC buffers can be tricky to deal with. For the most part, variably sized RPC buffers consist of either character strings (which should contain **NULL** termination defining the size), or amorphous buffers for which there is a corresponding size value passed to the function. The examples that follow document some of the common errors involved in dealing with such buffers.

A buffer that hasn't been explicitly declared as a **string** type cannot be assumed to contain a **NULL** terminator, and thus must not be passed to C runtime string functions prior to verification of zero termination. This cannot be done by touching a byte outside the valid length of your buffer.

Example

```
BOOLEAN
RpcFunction(
    [in] DWORD NameSize,
    [in, size_is(NameSize)] PWCHAR Name
)
{
    if (Name && wcslen(Name) == 0) {
        return STATUS_CTX_WINSTATION_NAME_INVALID;
    }

    […]
```

Remarks

The *NameSize* parameter should be checked and used to bound any operations, either by explicitly attaching a **NULL**-terminator (on the server side), or by using bounded string operations with the size of the buffer specified.

Don't copy arbitrary length data into independently-sized buffers

Data buffers should not be assumed to be bound by an arbitrary size limit. An explicit check of the size of the indicated data must be made prior to copying to local fixed-size buffers.

Example

```
HRESULT
RpcSetInfo(
    [in, string] LPCWSTR pwszName
);
{
    WCHAR wszPath[MAX_PATH + 1];

    wcscpy(wszJobPath, "\\");
    wcscat(wszJobPath, pwszName);
```

Remarks

string guarantees that the *pwszName* parameter is zero terminated, not that its length is less than **MAX_PATH**.

Using size_is may result in a zero-length structure; it is not safe to dereference this without first checking its length

A **size_is** specifier can result in a zero-length buffer but a non-**NULL** buffer pointer (as reference pointers, such as passed parameters, cannot be **NULL**). A **unique** pointer can always be **NULL**. The best practice is to verify both the pointer as non-**NULL** and the buffer size as non-zero to avoid problems.

Example 1

```
ULONG
RpcServerSideRoutine(
    [in] ULONG StructureSize,
    [in, size_is(StructureSize)] PSTRUCTURE Structure
)
{
    ULONG NameLength = 0;

    if (Structure) {
        NameLength = Structure->NameLength;
[…]
```

Remarks

There is no guarantee in this example that the *StructureSize* parameter is sufficient to cover the *NameLength* member, and in fact, the Structure pointer may be non-**NULL**, while *StructureSize*, and thus the allocated buffer, indicate a zero length.

Example 2

```
ULONG
RpcServerSideRoutine2(
    [in] ULONG StructureSize,
    [in, unique, size_is(StructureSize)] PSTRUCTURE Structure
)
{
    ULONG NameLength = 0;
    if (StructureSize) {
        NameLength = Structure->NameLength;
[…]
```

Remarks

This example presents a similar problem. In this case, the *StructureSize* parameter could be non-zero, but Structure—being defined as **unique**—could contain a **NULL**.)

Calculations in a size_is or length_is specification are susceptible to overflow

Calculations in the midl definition for a **size_is** or **length_is** specification are subject to overflow problems. If you perform a calculation in a **size_is** or **length_is** specification, consider what difficulties overflow (or rounding) might cause.

Strict context handles

Context handles enable RPC servers to associate information with calls. RPC looks up context handles in a linked list associated with each binding handle. If you have more than one interface accessible from a single binding handle, then the code must be prepared to reject invalid handles or use strict context handles. Interfaces end up being accessible from a single binding handle if they share things like the same named pipe. Using the **[strict_context_handle]** on the interface definition in the .acf file causes RPC to only allow context handles to be used against interfaces that created them.

Kernel-Mode Specifiers

The most common programming errors associated with working in kernel mode are associated with improperly validating user-provided structures. The practice of improperly validating user-provided structures can cause problems either by the increased kernel-mode privilege, or by accessing memory that could cause a system crash. The following is a list of rules that should be observed in kernel mode:

- Probe any user-provided pointers within a try-except before reading or writing.
- Read user-mode memory only once; capture it for subsequent uses.
- Don't trust any user mode contents. Never trust the Thread Environment Block (TEB).
- Other threads may change kernel objects' states. Use locks.
- Never call kernel routines without access-checking objects passed to them.
- Validate buffer sizes for buffered I/O.
- Validate parameters on **METHOD_NEITHER**.

Don't access user-provided memory without probing

All memory accesses using pointers provided by user mode must be validated with a probe to stop user mode reading or writing of data for which the caller has no access. Some memory addresses have side effects, such as a bugcheck, or hardware effects as in the case of memory-mapped device registers. It's not enough to simply use **try-except** clauses. The obvious way to avoid these problems is to always probe user-provided addresses.

Example

```
NTSTATUS
NtBadFunction(
    PVOID Param,
    ULONG Size
)
{
    ROUND_COGG RoundCogg;
    try {
        // Probe will be skipped if Size == 0.
        ProbeForRead(Param,
                     Size,
                     1);
        // The next statement may cause an exception
    // when the above probe is skipped.  If an
    // address such as 0xFFFFFC00 is passed,
    // memory-management code will call KeBugCheck
```

```
        // directly, without returning control to
    // the exception handler.

        if (((PCOGG)Param)->Type == Round) {
            […]
        }
    } except (EXCEPTION_EXECUTE_HANDLER) {
        return GetExceptionCode();
    }
}
```

Don't do multiple user-mode reads without captures

Despite the probe and capture rules (that is, read once, and if you have to read twice, capture first and then read again), many programmers commit the common error of making kernel-mode reads of user-mode memory multiple times without a capture. This isn't the best approach. Along those lines, user-mode memory shouldn't be used for temporary storage of a kernel-mode algorithm; the data might have changed or become invalid during the interim. Data read from user-mode memory should be read only once. Data once written to user-mode memory shouldn't be reread without revalidation. To avoid this type of problem, probe once and (if necessary) capture for multiple reads.

Example

```
NTSTATUS
KernelRoutineCalledByUserMode(PUNICODE_STRING UnicodeString)

{
    UNICODE_STRING CapturedString;
    WCHAR          Buffer[…];
    […]
    try {
        ProbeForRead(UnicodeString,
                     sizeof(UNICODE_STRING),
                     sizeof(UCHAR));
        ProbeForRead(UnicodeString->Buffer,
                     UnicodeString->Length,
                     sizeof(UCHAR));
        CapturedString = *UnicodeString;
        RtlCopyMemory(Buffer,
                     CapturedString.Buffer,
                     CapturedString.Length);
[…]
```

Remarks

There is a tricky problem in this code example: the values for the length and buffer of the string have actually been read twice. The first time they were used to probe the buffer for read, and the second they were captured into the **CapturedString UNICODE_STRING** structure. The values might actually have been changed in the meantime, invalidating the probe and potentially causing mischief.

Don’t trust the TEB

Accessing the current Thread Environment Block (TEB) from kernel mode is just as dangerous as accessing any other user-mode memory. Although this is generally a system construct, it could still be modified from user mode. In general, validate any user-mode input into kernel mode, even if it’s an implicit “system” structure.

Avoid race conditions when modifying kernel data on user request

Often kernel-mode routines manipulate kernel objects and move them from one state to another. A kernel routine usually validates that the object is in the correct state before advancing to the next step; such checks must be done under locks if user mode can request the same transition from two threads at once. If it’s possible for the service to be reentered, avoid this potential problem by always using locks to validate that an object is in the correct state *before* advancing it to the next step. Possible reentry could include malicious attacks, incorrect calls, and so forth, and is not limited to the path taken when the function is used correctly.

Example

```
NTSTATUS
KernelFreeObject(HANDLE Object)
{
    if (!IsValidHandle(Object)) {
        return STATUS_INVALID_HANDLE;
    }
    return CallObjectFreeRoutineByHandle(Object);
}
```

Remarks

Two threads running nearly simultaneously in this routine may both get returns from *IsValidHandle*, implying that the handle is valid. Both threads would then call the free routine, probably causing something nasty to happen.

Dealing with common interfaces for user mode and kernel mode

Many kernel-mode interfaces have the same interface to manipulate objects from user mode. The object is often used without access checking, although it should not be accessible to user mode even for a short time. To avoid this problem, mark objects with the correct access mode.

Validating buffered I/O in device drivers

Device drivers using buffered I/O paths must validate input and output buffer sizes before writing or reading data. Validate that input buffers are large enough to contain request packets, and that output buffers are large enough to contain results.

Example

```
if (InputBufferLength < sizeof (INPUT_STRUCTURE) ||
    OutputBufferLength == 0) {
    return STATUS_BUFFER_TOO_SMALL;
}
[…Do some work here and get Results buffer…]
RtlCopyMemory(OutputBuffer,
            Results,
            ResultsLength);
[…]
```

Remarks

The lack of a size check on **OutputBuffer** could cause an access violation if **ResultsLength > OutputBufferLength**.

METHOD_NEITHER requires full probe and capture

Buffers sent to IOCTLs of type **METHOD_NEITHER** are simply pointers supplied by the user; they are neither probed nor captured before being passed to the intended driver. One way to avoid problems is to properly probe and capture data passed using **METHOD_NEITHER** IOCTLs. When creating new IOCTLs, a better solution is to use **METHOD_BUFFERED** if the data does not require a pointer to be completely expressed.

Example

```
case IOCTL_USING_METHOD_NEITHER:
    RtlCopyMemory(CapturedBuffer,
        IrpSp->Parameters.DeviceIoControl.Type3InputBuffer,
        IrpSp->Parameters.DeviceIoControl.InputBufferLength);
```

Remarks

The reference to *Type3InputBuffer* on a **METHOD_NEITHER** IOCTL dereferences a buffer pointer directly passed by the caller, and not a pointer buffered by the I/O subsystem. This situation can cause a bugcheck or direct access to kernel-mode memory by a user-mode process.

Solution Summary

It's nice to have a concise version of the solutions to these common programming problems, so this section summarizes how to avoid the issues discussed in this chapter.

RPC Errors

1. Using **pointer_default(unique)** and embedded pointers: Check unique pointers for **NULL** before dereferencing.
2. A valid **switch_is** value in an RPC-capable structure doesn't ensure a non-**NULL** pointer: When using a **switch_is** construct that has a default clause:
    - Verify that the value switching on is within expected range.
    - Verify that pointers within the switched object are not **NULL** before dereferencing them.
3. A **NULL** DACL affords no protection: Don't use **NULL** DACLs, they don't protect anything.
4. Call **RpcImpersonateClient()** before any security-relevant operation: Impersonate before acting on behalf of the caller, and check the result.
5. Starting and stopping impersonation: Stop impersonating when finished acting on behalf of the caller, and check the result.
6. Strings are only zero-terminated when declared with **string** in the .idl. Don't expect strings to be zero-terminated unless **string** is specified in the *.idl file.
7. Don't copy arbitrary length data into independently-sized buffers: This one's self-answering!
8. Using **size_is** may result in a zero-length structure; it's not safe to dereference this without first checking its length. Check length of **size_is** specified data before dereferencing corresponding pointers.
9. Calculations in a **size_is** or **length_is** specification are susceptible to overflow. Be aware that calculations in midl definitions using **size_is** and **length_is** can overflow, and that it might be impossible for the server to detect this.
10. Strict context handles: Use strict context handles.

Kernel-Mode Specifiers

1. Don't access user-provided memory without probing. Probe any user-provided pointers within a try-except before reading or writing.
2. Don't do multiple user-mode reads without captures. Read user-mode memory only once; capture it for subsequent uses.
3. Never trust the TEB. Don't trust any user mode contents.
4. Avoid race conditions when modifying kernel data on user request. Use locks to protect objects that can be changed by multiple threads.
5. Dealing with common interfaces for user mode and kernel mode. Never call kernel routines without access checking objects passed to them.
6. Validate buffered I/O in device drivers. Validate buffer sizes for buffered I/O.
7. **METHOD_NEITHER** requires full probe and capture. Validate parameters on **METHOD_NEITHER**.

CHAPTER 6

Bitmaps

A *bitmap* is a graphical object used to create, manipulate (scale, scroll, rotate, and paint), and store images as files on a disk. This overview describes the bitmap classes and bitmap operations.

About Bitmaps

A bitmap is one of the GDI objects that can be selected into a *device context* (DC). Device contexts are structures that define a set of graphic objects and their associated attributes, and graphic modes that affect output. The table below describes the GDI objects that can be selected into a device context:

Graphic object	**Use**
Bitmaps	Creates, manipulates (scale, scroll, rotate, and paint), and stores images as files on a disk.
Brushes	Paints the interior of polygons, ellipses, and paths.
Fonts	Draws text on video displays and other output devices.
Logical Palette	A color palette created by an application and associated with a given device context.
Paths	One or more figures (or shapes) that are filled and/or outlined.
Pens	A graphics tool that a Win32-based application uses to draw lines and curves.
Regions	A rectangle, polygon, or ellipse (or a combination of two or more of these shapes) that can be filled, painted, inverted, framed, and used to perform hit testing (testing for the cursor location).

From a developer's perspective, a bitmap consists of a collection of structures that specify or contain the following elements:

- A header that describes the resolution of the device on which the rectangle of pixels was created, the dimensions of the rectangle, the size of the array of bits, and so on.
- A logical palette.
- An array of bits that defines the relationship between pixels in the bitmapped image and entries in the logical palette.

A bitmap size is related to the type of image it contains. Bitmap images can be either monochrome or color. In an image, each pixel corresponds to one or more bits in a bitmap. Monochrome images have a ratio of 1 bit per pixel (bpp). Color imaging is more complex. The number of colors that can be displayed by a bitmap is equal to two raised to the number of bits per pixel. Thus, a 256-color bitmap requires 8 bpp (2^8 = 256).

Control Panel applications are examples of applications that use bitmaps. When you select a wallpaper for your desktop, you actually select a bitmap, which the system uses to paint the desktop background. The system creates the selected wallpaper pattern by repeatedly drawing a 32-by-32 pixel pattern on the desktop.

Figure 6-1 presents the developer's perspective of the bitmap found in the file Redbrick.bmp. It shows a palette array, a 32-by-32 pixel rectangle, and the index array that maps colors from the palette to pixels in the rectangle.

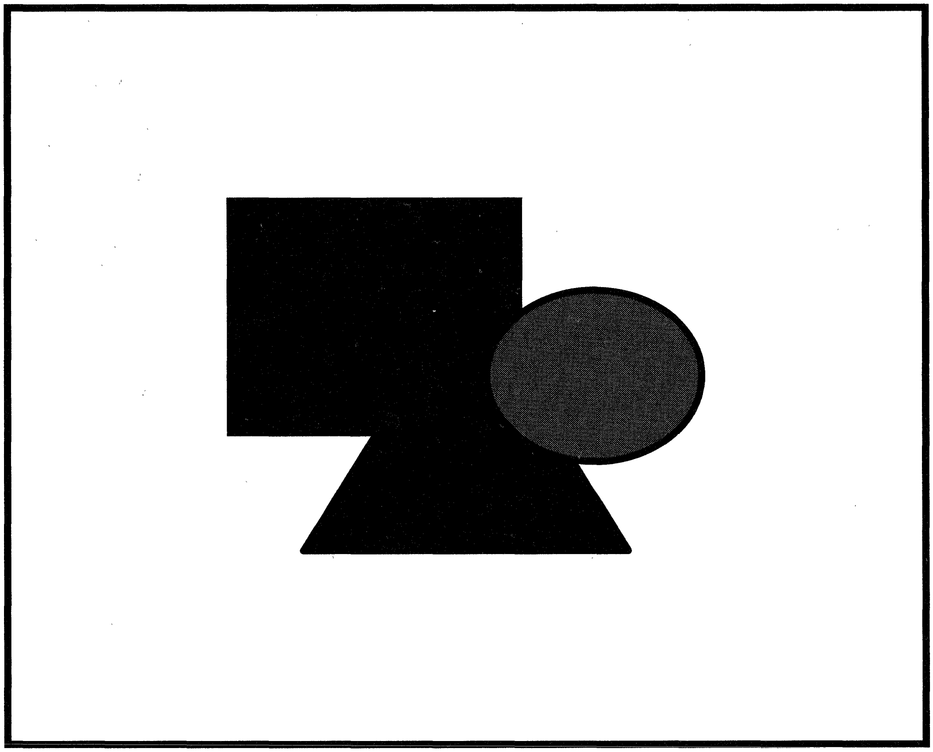


Figure 6-1: Developer's perspective of the Redbrick bitmap.

In the preceding example, the rectangle of pixels was created on a video graphics adaptor (VGA) display device using a palette of 16 colors. A 16-color palette requires 4-bit indexes; therefore, the array that maps palette colors to pixel colors is composed of 4-bit indexes, too. (For more information about logical color-palettes, see *Colors.*)

Note In the above bitmap, the system maps indexes to pixels, beginning with the bottom scan line of the rectangular region and ending with the top scan line. A *scan line* is a single row of adjacent *pixels* on a video display. For example, the first row of the array (row 0) corresponds to the bottom row of pixels, scan line 31. This is because the above bitmap is a bottom-up device-independent bitmap (DIB), a common type of bitmap. In top-down DIBs and in device-dependent bitmaps (DDBs), the system maps indexes to pixels beginning with the top scan line.

Bitmap Classifications

There are two classes of bitmaps:

- Device-independent bitmaps (DIBs). The DIB file format was designed to ensure that bitmapped graphics created using one application can be loaded and displayed in another application, retaining the same appearance as the original.
- Device-dependent bitmaps (DDBs) were the only bitmaps available in early versions of 16-bit Microsoft Windows (prior to version 3.0). However, as display technology improved and the variety of available display devices increased, certain inherent problems surfaced which could only be solved using DIBs. For example, there was no method of storing (or retrieving) the resolution of the display type on which a bitmap was created, so a drawing application could not determine quickly whether a bitmap was suitable for the type of video display device on which the application was running.

Device-Independent Bitmaps

Bitmaps that contain a color table are device-independent. A *color table* describes how pixel values correspond to RGB color values. *RGB* is a model for describing colors that are produced by emitting light. A DIB contains the following color and dimension information:

- The color format of the device on which the rectangular image was created.
- The resolution of the device on which the rectangular image was created.
- The palette for the device on which the image was created.
- An array of bits that maps red, green, blue (RGB) triplets to pixels in the rectangular image.
- A data-compression identifier that indicates the data compression scheme (if any) used to reduce the size of the array of bits.

The color and dimension information is stored in a **BITMAPINFO** structure.

The **BITMAPINFO** structure consists of a bitmap information header structure (see *Bitmap Header Types*) followed by two or more **RGBQUAD** structures. The bitmap information header structure specifies the dimensions of the pixel rectangle, describes the device's color technology, and identifies the compression schemes used to reduce the bitmap's size. The **RGBQUAD** structures identify the colors that appear in the pixel rectangle.

There are two varieties of DIBs:

- A bottom-up DIB, in which the origin lies at the lower-left corner.
- A top-down DIB, in which the origin lies at the upper-left corner.

If the height of a DIB, as indicated by the **Height** member of the bitmap information header structure, is a positive value, it is a bottom-up DIB; if the height is a negative value, it is a top-down DIB. Top-down DIBs cannot be compressed.

The color format is specified in terms of a count of color planes and color bits. The count of color planes is always 1; the count of color bits is 1 for monochrome bitmaps, 4 for VGA bitmaps, and 8, 16, 24, or 32 for bitmaps on other color devices. An application retrieves the number of color bits that a particular display (or printer) uses by calling the **GetDeviceCaps** function, specifying BITSPIXEL as the second argument.

The resolution of a display device is specified in pixels per meter. An application can retrieve the horizontal resolution for a video display, or printer, by following this three-step process:

1. Call the **GetDeviceCaps** function, specifying HORZRES as the second argument.
2. Call **GetDeviceCaps** a second time, specifying HORZSIZE as the second argument.
3. Divide the first return value by the second return value.

The application can retrieve the vertical resolution by using the same three-step process with different parameters: VERTRES in place of HORZRES, and VERTSIZE in place of HORZSIZE.

The palette is represented by an array of **RGBQUAD** structures that specify the red, green, and blue intensity components for each color in a display device's color palette. Each color index in the palette array maps to a specific pixel in the rectangular region associated with the bitmap. The size of this array, in bits, is equivalent to the width of the rectangle, in pixels, multiplied by the height of the rectangle, in pixels, multiplied by the count of color bits for the device. An application can retrieve the size of the device's palette by calling the **GetDeviceCaps** function, specifying the **NUMCOLORS** constant as the second argument.

The Microsoft Win32 API supports the compression of the palette array for 8-bpp and 4-bpp bottom-up DIBs. These arrays can be compressed by using the run-length encoding (RLE) scheme. The RLE scheme uses 2-byte values, the first byte specifying the number of consecutive pixels that use a color index and the second byte specifying the index. For more information about bitmap compression, see the description of the **BITMAPINFOHEADER**, **BITMAPCOREHEADER**, **BITMAPFILEHEADER**, **BITMAPV4HEADER**, and **BITMAPV5HEADER** structures.

An application can create a DIB from a DDB by initializing the required structures and calling the **GetDIBits** function. To determine whether a device supports this function, call the **GetDeviceCaps** function, specifying RC_DI_BITMAP as the RASTERCAPS flag.

An application that needs to copy a bitmap can use **TransparentBlt** to copy all pixels in a source bitmap to a destination bitmap, except for those pixels that match the transparent color.

An application can use a DIB to set pixels on the display device by calling the **SetDIBitsToDevice** or the **StretchDIBits** function. To determine whether a device supports the **SetDIBitsToDevice** function, call the **GetDeviceCaps** function, specifying RC_DIBTODEV as the RASTERCAPS flag. Specify RC_STRETCHDIB as the RASTERCAPS flag to determine if the device supports **StretchDIBits**.

An application that needs to display a pre-existing DIB can use the **SetDIBitsToDevice** function. For example, a spreadsheet application can open existing charts and display them in a window by using the **SetDIBitsToDevice** function. To repeatedly redraw a bitmap in a window, however, the application should use the **BitBlt** function. For example, a multimedia application that combines animated graphics with sound would benefit from calling the **BitBlt** function, because it executes faster than **SetDIBitsToDevice**.

Device-Dependent Bitmaps

Note Device-dependent bitmaps are supported only for compatibility with applications written for early versions of 16-bit Windows (prior to 3.0). If you are writing a Win32-based application, or porting a 16-bit Windows-based application to the Win32 API, you should use DIBs.

DDBs are described by using a single structure, the **BITMAP** structure. The members of this structure specify the width and height of a rectangular region, in pixels; the width of the array that maps entries from the device palette to pixels; and the device's color format, in terms of color planes and bits per pixel. An application can retrieve the color format of a device by calling the **GetDeviceCaps** function and specifying the appropriate constants.

There are two types of DDBs: discardable and nondiscardable. A discardable DDB is a bitmap that the system discards if the bitmap is not selected into a DC, and if system memory is low. The **CreateDiscardableBitmap** function creates discardable bitmaps. The **CreateBitmap**, **CreateCompatibleBitmap**, and **CreateBitmapIndirect** functions create nondiscardable bitmaps.

An application can create a DDB from a DIB by initializing the required structures and calling the **CreateDIBitmap** function. Specifying CBM_INIT in the call to **CreateDIBitmap** is equivalent to calling the **CreateCompatibleBitmap** function to create a DDB in the format of the device, and then calling the **SetDIBits** function to translate the DIB bits to the DDB. To determine whether a device supports the **SetDIBits** function, call the **GetDeviceCaps** function, specifying RC_DI_BITMAP as the RASTERCAPS flag.

Bitmap Header Types

The bitmap has four basic header types:

- **BITMAPCOREHEADER**
- **BITMAPINFOHEADER**
- **BITMAPV4HEADER**
- **BITMAPV5HEADER**

The four types of bitmap headers are differentiated by the **Size** member, which is the first **DWORD** in each of the structures.

The **BITMAPV5HEADER** structure is an extended **BITMAPV4HEADER** structure, which is an extended **BITMAPINFOHEADER** structure. However, the **BITMAPINFOHEADER** and **BITMAPCOREHEADER** have only the **Size** member in common with other bitmap header structures.

The **BITMAPCOREHEADER** and **BITMAPV4HEADER** formats have been superseded by **BITMAPINFOHEADER** and **BITMAPV5HEADER** formats, respectively. The **BITMAPCOREHEADER** and **BITMAPV4HEADER** formats are presented for completeness and backward compatibility.

The **BITMAPFILEHEADER** structure contains information about the type, size, and layout of a file that contains a DIB. A **BITMAPINFO** or **BITMAPCOREINFO** structure immediately follows the **BITMAPFILEHEADER** structure in the DIB file.

There are two formats for reading and storing bitmap data in a file, the *file format* and the *Win32 API format.* The file format and the format used by Win32 API are similar, but not identical. Figure 6-2 shows the two types of formats. All segments are used for the file format, while the Win32 API format excludes **BITMAPFILEHEADER**.

A color table describes how pixel values correspond to RGB color values. RGB is a model for describing colors that are produced by emitting light.

Profile data refers to either the profile file name (linked profile) or the actual profile bits (embedded profile). The file format places the profile data at the end of the file. The Win32 API format usually places the profile data just after the color table (if present). However, if the function receives a packed DIB, the profile data comes after the bitmap bits, like in the file format.

Profile data will exist only for **BITMAPV5HEADER** structures where **bV5CSType** is PROFILE_LINKED or PROFILE_EMBEDDED. For Win32 functions that receive packed DIBs, the profile data comes after the bitmap data.

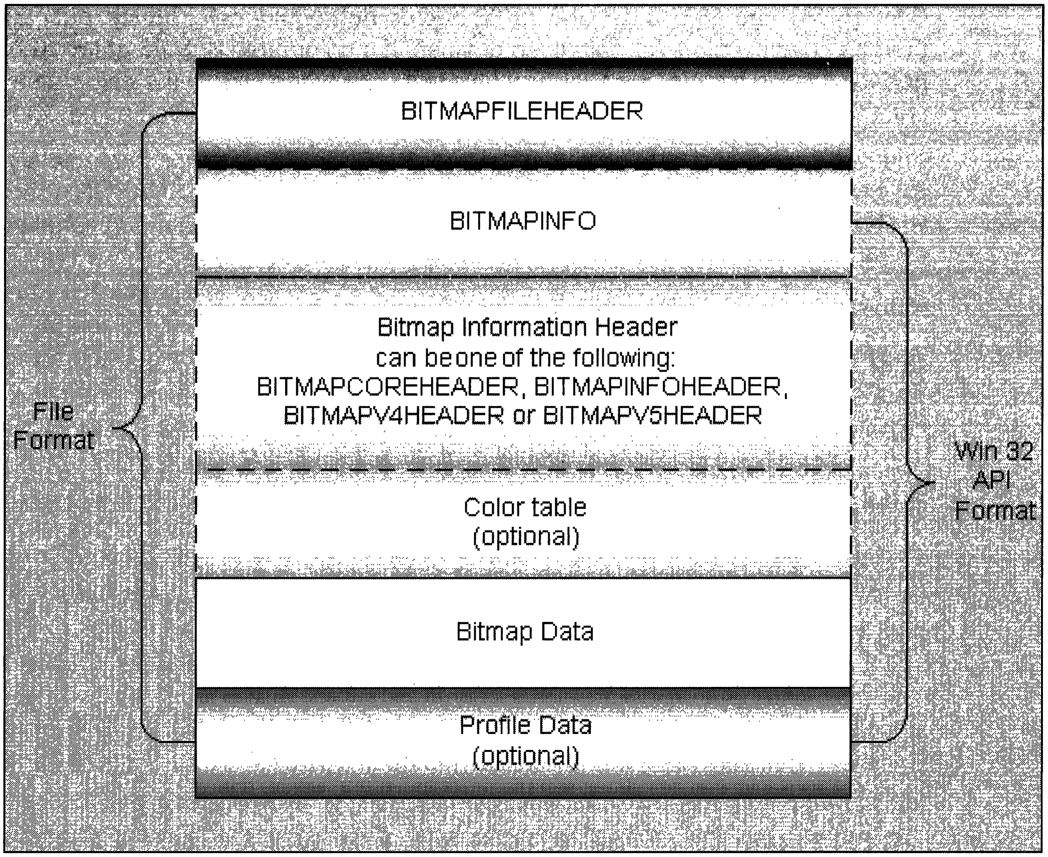

BITMAPFILEHEADER
BITMAPINFO
Bitmap Information Header
can be one of the following:
BITMAPCOREHEADER, BITMAPINFOHEADER,
BITMAPV4HEADER or BITMAPV5HEADER
Color table
(optional)
Bitmap Data
Profile Data
(optional)
File
Format
Win 32
API
Format

Figure 6-2: An example of the file format and the Win32 API format.

A *palettized device* is any device that uses palettes to assign colors. The classic example of a palettized device is a display running in 8-bit color depth (that is, 256 colors). The display in this mode uses a small color table to assign colors to a bitmap. The colors in a bitmap are assigned to the closest color in the palette that the device is using. The palettized device does not create an optimal palette for displaying the bitmap; it uses whatever is in the current palette. Applications are responsible for creating a palette and selecting it into the system. In general, 16-bpp, 24-bpp, and 32-bpp bitmaps do not contain color tables (a.k.a. optimal palettes for the bitmap); the application is responsible for generating an optimal palette in this case. However, 16-bpp, 24-bpp, and 32-bpp bitmaps can contain such optimal color tables for displaying on palettized devices; in this case, the application just needs to create a palette based on the color table present in the bitmap file.

Bitmaps that are of 1, 4, or 8 bpp must have a color table with a maximum size based on the bpp. The maximum size for 1-bpp, 4-bpp, and 8-bpp bitmaps is 2 to the power of the bpp. Thus, a 1-bpp bitmap has a maximum of two colors, the 4-bpp bitmap has a maximum of 16 colors, and the 8-bpp bitmap has a maximum of 256 colors.

Bitmaps that are 16 bpp, 24 bpp, or 32 bpp do not require color tables, but can have them to specify colors for palettized devices. If a color table is present for 16-bpp, 24-bpp, or 32-bpp bitmap, the **ClrUsed** field will specify the size of the color table, and the color table must have that many colors in it. **ClrUsed** of zero indicates no color table.

The red, green, and blue bit field masks for BI_BITFIELD bitmaps immediately follow the **BITMAPINFOHEADER**, **BITMAPV4HEADER**, and **BITMAPV5HEADER** structures. The **BITMAPV4HEADER** and **BITMAPV5HEADER** structures contain additional members for red, green, and blue masks, as follows:

Member	Meaning
RedMask	Color mask that specifies the red component of each pixel, valid only if the **Compression** member is set to BI_BITFIELDS.
GreenMask	Color mask that specifies the green component of each pixel, valid only if the **Compression** member is set to BI_BITFIELDS.
BlueMask	Color mask that specifies the blue component of each pixel, valid only if the **Compression** member is set to BI_BITFIELDS.

When the **biCompression** member of **BITMAPINFOHEADER** is set to BI_BITFIELDS and the function receives an argument of type **LPBITMAPINFO**, the color masks will immediately follow the header. The color table, if present, will follow the color masks. **BITMAPCOREHEADER** bitmaps do not support color masks.

By default, bitmap data is bottom-up in its format. Bottom-up means that the first scan line in the bitmap data is the last scan line to be displayed. For example, the 0th pixel of the 0th scan line of the bitmap data of a 10-pixel-by-10-pixel bitmap will be the 0th pixel of the ninth scan line of the displayed or printed image. Run-length encoded (RLE) format bitmaps and **BITMAPCOREHEADER** bitmaps can not be top-down bitmaps. The scan lines are **DWORD**-aligned, except for RLE-compressed bitmaps. They must be padded for scan-line widths, in bytes, that are not evenly divisible by four, except for RLE compressed bitmaps. For example, a 10-pixel-by-10-pixel, 24-bpp bitmap will have two padding bytes at the end of each scan line.

JPEG and PNG Extensions for Specific Bitmap Functions and Structures

Starting with the Microsoft Windows 98 and Windows 2000 operating systems, the **StretchDIBits** and **SetDIBitsToDevice** functions have been extended to allow JPEG and PNG images to be passed as the source image to printer devices. This extension is not intended as a means to supply general JPEG and PNG decompression to applications, but, instead, to allow applications to send JPEG-compressed and PNG-compressed images directly to printers that have hardware support for JPEG and PNG images, respectively.

The **BITMAPINFOHEADER**, **BITMAPV4HEADER**, and **BITMAPV5HEADER** structures are extended to allow specification of *biCompression* values indicating that the bitmap data is a JPEG or PNG image. These compression values are only valid for **SetDIBitsToDevice** and **StretchDIBits** when the *hdc* parameter specifies a printer device. To support metafile spooling of the printer, the application should not rely on the return value to determine whether the device supports the JPEG or PNG file. The application must issue QUERYESCSUPPORT with the corresponding escape before calling **SetDIBitsToDevice** and **StretchDIBits**. If the validation escape fails, then the application must fall back on its own JPEG or PNG support to decompress the image into a bitmap.

Bitmaps, Device Contexts, and Drawing Surfaces

A *device context* (DC) is a data structure defining the graphics objects, their associated attributes, and the graphics modes affecting output on a device. To create a DC, call the **CreateDC** function; to retrieve a DC, call the **GetDC** function.

Before returning a handle that identifies that DC, the system selects a drawing surface into the DC. If the application called the **CreateDC** function to create a device context for a VGA display, the dimensions of this drawing surface are 640 pixels by 480 pixels. If the application called the **GetDC** function, the dimensions reflect the size of the client area.

Before an application can begin drawing, it must select a bitmap with the appropriate width and height into the DC by calling the **SelectObject** function. When an application passes the handle to the DC to one of the graphics device interface (GDI) drawing functions, the requested output appears on the drawing surface selected into the DC.

For more information, see *Memory Device Contexts*.

Bitmap Creation

The Win32 API provides a number of functions to create bitmaps. To create a bitmap, use the **CreateBitmap**, **CreateBitmapIndirect**, or **CreateCompatibleBitmap** function, **CreateDIBitmap**, and **CreateDiscardableBitmap**.

These functions all you to specify the width and height, in pixels, of the bitmap. The **CreateBitmap** and **CreateBitmapIndirect** function also allow you to specify the number of color planes and the number of bits required to identify the color. On the other hand, the **CreateCompatibleBitmap** and **CreateDiscardableBitmap** functions use a specified device context to obtain the number of color planes and the number of bits required to identify the color.

The **CreateDIBitmap** function creates a device-independent bitmap. It contains a color table that describes how pixel values correspond to RGB color values. For more information, see *Device-Independent Bitmaps*.

After the bitmap has been created, you cannot change its size, number of color planes, or number of bits required to identify the color.

When you no longer need a bitmap, call the **DeleteObject** function to delete it.

Bitmap Rotation

The Win32 API provides a function to copy a bitmap into a parallelogram; this function, **PlgBlt**, performs a bit-block transfer from a rectangle in a source device context into a parallelogram in a destination device context. In order to rotate the bitmap, an application must provide the coordinates, in world units, to be used for the corners of the parallelogram. (For more information about rotation and world units, see *Coordinate Spaces and Transformations.*)

Bitmap Scaling

The Win32 API also provides a function to scale a bitmap; this function, **StretchBlt**, performs a bit-block transfer from a rectangle in a source device context into a rectangle in a destination device context. However, unlike the **BitBlt** function, which duplicates the source rectangle dimensions in the destination rectangle, **StretchBlt** allows an application to specify the dimensions of both the source and destination rectangles. When the destination bitmap is smaller than the source bitmap, the system combines rows or columns of color data (or both) in the bitmap before rendering the corresponding image on the display device. The system combines the color data according to the specified stretch mode, which the application defines by calling the **SetStretchBltMode** function. When the destination bitmap is larger than the source bitmap, the system scales or magnifies each pixel in the resultant image accordingly.

Bitmaps as Brushes

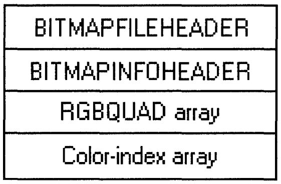
The Win32 API provides a number of functions that use the brush currently selected into a device context to perform bitmap operations. For example, the **PatBlt** function replicates the brush in a rectangular region within a window, and the **FloodFill** function replicates the brush inside an area in a window bounded by the specified color (unlike **PatBlt**, **FloodFill** does fill nonrectangular shapes).

The **FloodFill** function replicates the brush within a region bounded by a specified color. However, unlike the **PatBlt** function, **FloodFill** does not combine the color data for the brush with the color data for the pixels on the display; it sets the color of all pixels within the enclosed region on the display to the color of the brush that is currently selected into the device context.

Bitmap Storage

Bitmaps should be saved in a file that uses the established bitmap file format, and assigned a name with the three-character .bmp extension. The established bitmap file format consists of a **BITMAPFILEHEADER** structure, followed by either a **BITMAPINFOHEADER**, **BITMAPV4HEADER**, or **BITMAPV5HEADER** structure. An array of **RGBQUAD** structures (also called a color table) follows the bitmap information header structure. The color table is followed by a second array of indexes into the color table (the actual bitmap data).

The bitmap file format is shown here:



Windows 95 and Windows NT 4.0: Replace the **BITMAPINFOHEADER** structure with the **BITMAPV4HEADER** structure.

Windows 98 and Windows 2000: Replace the **BITMAPINFOHEADER** structure with the **BITMAPV5HEADER** structure.

The members of the **BITMAPFILEHEADER** structure identify the file; specify the size of the file, in bytes; and specify the offset, from the first byte in the header to the first byte of bitmap data. The members of the **BITMAPINFOHEADER**, **BITMAPV4HEADER**, or **BITMAPV5HEADER** structure specify the width and height of the bitmap, in pixels; the color format (count of color planes and color bits-per-pixel) of the display device on which the bitmap was created; whether the bitmap data was compressed before storage, and the type of compression used; the number of bytes of bitmap data; the resolution of the display device on which the bitmap was created;

and the number of colors represented in the data. The **RGBQUAD** structures specify the RGB intensity values for each of the colors in the device's palette. The color-index array maps indexes values from the **RGBQUAD** array to pixels in a rectangular region on the display.

The following hexadecimal output shows the contents of the file Redbrick.bmp:

```
0000    42 4d 76 02 00 00 00 00  00 00 76 00 00 00 28 00
0010    00 00 20 00 00 00 20 00  00 00 01 00 04 00 00 00
0020    00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
0030    00 00 00 00 00 00 00 00  00 00 00 00 80 00 00 80
0040    00 00 00 80 80 00 80 00  00 00 80 00 80 00 80 80
0050    00 00 80 80 80 00 c0 c0  c0 00 00 00 ff 00 00 ff
0060    00 00 00 ff ff 00 ff 00  00 00 ff 00 ff 00 ff ff
0070    00 00 ff ff ff 00 00 00  00 00 00 00 00 00 00 00
```

(continued)

(continued)

```
0080    00 00 00 00 00 00 00 00  00 00 00 00 00 00 09 00
0090    00 00 00 00 00 00 11 11  01 19 11 01 10 10 09 09
00a0    01 09 11 11 01 90 11 01  19 09 09 91 11 10 09 11
00b0    09 11 19 10 90 11 19 01  19 19 10 10 11 10 09 01
00c0    91 10 91 09 10 10 90 99  11 11 11 11 19 00 09 01
00d0    91 01 01 19 00 99 11 10  11 91 99 11 09 90 09 91
00e0    01 11 11 11 91 10 09 19  01 00 11 90 91 10 09 01
00f0    11 99 10 01 11 11 91 11  11 19 10 11 99 10 09 10
0100    01 11 11 11 19 10 11 09  09 10 19 10 10 10 09 01
0110    11 19 00 01 10 19 10 11  11 01 99 01 11 90 09 19
0120    11 91 11 91 01 11 19 10  99 00 01 19 09 10 09 19
0130    10 91 11 01 11 11 91 01  91 19 11 00 99 90 09 01
0140    01 99 19 01 91 10 19 91  91 09 11 99 11 10 09 91
0150    11 10 11 91 99 10 90 11  01 11 11 19 11 90 09 11
0160    00 19 10 11 01 11 99 99  99 99 99 99 99 99 09 99
0170    99 99 99 99 99 99 00 00  00 00 00 00 00 00 00 00
0180    00 00 00 00 00 00 90 00  00 00 00 00 00 00 00 00
0190    00 00 00 00 00 00 99 11  11 11 19 10 19 19 11 09
01a0    10 90 91 90 91 00 91 19  19 09 01 10 09 01 11 11
01b0    91 11 11 11 10 00 91 11  01 19 10 11 10 01 01 11
01c0    90 11 11 11 91 00 99 09  19 10 11 90 09 90 91 01
01d0    19 09 91 11 01 00 90 10  19 11 00 11 11 00 10 11
01e0    01 10 11 19 11 00 90 19  10 91 01 90 19 99 00 11
01f0    91 01 11 01 91 00 99 09  09 01 10 11 91 01 10 91
0200    99 11 10 90 91 00 91 11  00 10 11 01 10 19 19 09
0210    10 00 99 01 01 00 91 01  19 91 19 91 11 09 10 11
0220    00 91 00 10 90 00 99 01  11 10 09 10 10 19 09 01
0230    91 90 11 09 11 00 90 99  11 11 11 90 19 01 19 01
0240    91 01 01 19 09 00 91 10  11 91 99 09 09 90 11 91
0250    01 19 11 11 91 00 91 19  01 00 11 00 91 10 11 01
0260    11 11 10 01 11 00 99 99  99 99 99 99 99 99 99 99
0270    99 99 99 99 99 90
```

The following table shows the data bytes associated with the structures in a bitmap file:

Structure	Corresponding bytes
BITMAPFILEHEADER	0x00 – 0x0D
BITMAPINFOHEADER	0x0E – 0x31
RGBQUAD array	0x32 – 0x75
Color-index array	0x76 – 0x275

Bitmap Compression

The Win32 API supports formats for compressing bitmaps that define their colors with 8 bpp or 4 bpp. Compression reduces the disk and memory storage required for the bitmap.

Compression forms part of the following member names in the bitmap information header structures for different platforms. In the discussion that follows, compression is used to mean all of these variants:

Operating system	Compression
Windows NT 3.51 and earlier	**biCompression**
Windows NT 4.0 and Windows 95	**bV4Compression**
Windows 2000 and Windows 98	**bV5Compression**

When the **Compression** member of the bitmap information header structure is BI_RLE8, a run-length encoding (RLE) format is used to compress an 8-bit bitmap. This format can be compressed in encoded or absolute mode. Both modes can occur anywhere in the same bitmap:

- **Encoded mode** consists of two bytes: the first byte specifies the number of consecutive pixels to be drawn using the color index contained in the second byte. In addition, the first byte of the pair can be set to zero to indicate an escape character that denotes the end of a line, the end of a bitmap, or a delta, depending on the value of the second byte. The interpretation of the escape depends on the value of the second byte of the pair, which can be one of the following values:

Value	Meaning
0	End of line.
1	End of bitmap.
2	Delta. The 2 bytes following the escape contain unsigned values indicating the horizontal and vertical offsets of the next pixel from the current position.

- In **absolute mode**, the first byte is zero and the second byte is a value in the range 03H through FFH. The second byte represents the number of bytes that follow, each of which contains the color index of a single pixel. When the second byte is two or less, the escape has the same meaning as encoded mode. In absolute mode, each run must be aligned on a word boundary.

The following example shows the hexadecimal values of an 8-bit compressed bitmap:

```
03 04 05 06 00 03 45 56 67 00 02 78 00 02 05 01
02 78 00 00 09 1E 00 01
```

The bitmap expands as follows (two-digit values represent a color index for a single pixel):

```
04 04 04
06 06 06 06 06
45 56 67
78 78
move current position 5 right and 1 down
78 78
end of line
1E 1E 1E 1E 1E 1E 1E 1E 1E
end of RLE bitmap
```

When the **Compression** member is BI_RLE4, the bitmap is compressed by using a run-length encoding format for a 4-bit bitmap, which also uses encoded and absolute modes:

- In encoded mode, the first byte of the pair contains the number of pixels to be drawn using the color indexes in the second byte. The second byte contains two color indexes, one in its high-order 4 bits and one in its low-order 4 bits. The first of the pixels is drawn using the color specified by the high-order 4 bits, the second is drawn using the color in the low-order 4 bits, the third is drawn using the color in the high-order 4 bits, and so on, until all the pixels specified by the first byte have been drawn.
- In absolute mode, the first byte is zero. The second byte contains the number of color indexes that follow. Subsequent bytes contain color indexes in their high-order and low-order 4 bits, one color index for each pixel. In absolute mode, each run must be aligned on a word boundary. The end-of-line, end-of-bitmap, and delta escapes described for BI_RLE8 also apply to BI_RLE4 compression.

The following example shows the hexadecimal values of a 4-bit compressed bitmap:

```
03 04 05 06 00 06 45 56 67 00 04 78 00 02 05 01
04 78 00 00 09 1E 00 01
```

The bitmap expands as follows (single-digit values represent a color index for a single pixel):

```
0 4 0
0 6 0 6 0
4 5 5 6 6 7
7 8 7 8
move current position 5 right and 1 down
7 8 7 8
end of line
1 E 1 E 1 E 1 E 1
end of RLE bitmap
```

Alpha Blending

Alpha blending is used to display an alpha bitmap, which is a bitmap that has transparent or semitransparent pixels. In addition to a red, green, and blue color channel, each pixel in an alpha bitmap has a transparency component known as its *alpha channel*. The alpha channel typically contains as many bits as a color channel. For example, an 8-bit alpha channel can represent 256 levels of transparency, from 0 (the entire bitmap is transparent) to 255 (the entire bitmap is opaque).

Alpha blending mechanisms are invoked by calling **AlphaBlend**, which references the **BLENDFUNCTION** structure.

Alpha Values per Pixel

Alpha values per pixel are only supported for 32-bpp BI_RGB. This formula is defined as:

```
typedef struct {
  BYTE   Blue;
  BYTE   Green;
  BYTE    Red;
  BYTE    Alpha;
};
```

This is represented in memory, as shown in the following table:

31:24	23:16	15:08	07:00
Alpha	Red	Green	Blue

Global Alpha Blending Settings

Bitmaps can also be displayed with a transparency factor applied to the entire bitmap. Any bitmap format can be displayed with a global constant alpha value by setting **SourceConstantAlpha** in the **BLENDFUNCTION** structure. The global constant alpha value has 256 levels of transparency, from 0 (entire bitmap is completely transparent) to 255 (entire bitmap is completely opaque). The global constant alpha value is combined with the per-pixel alpha value.

Smooth Shading

Smooth shading is a method of shading a region with a color gradient. Including color information, along with the bounds of drawing primitive, specifies the color gradient. GDI linearly interpolates the color of the inside of the primitive passed on the color endpoints. Color and vertex information is included with position information in the **TRIVERTEX** structure.

Use the **GradientFill** function to fill a triangle or rectangle structure. To fill a triangle with smooth shading, call **GradientFill** with the three triangle endpoints. To fill a rectangle with smooth shading, call **GradientFill** with the upper-left and lower-right rectangle coordinates. **GradientFill** references the **TRIVERTEX**, **GRADIENT_RECT**, and **GRADIENT_TRIANGLE** structures.

For an example, see *Drawing a Shaded Triangle*.

ICM-Enabled Bitmap Functions

Windows 98 and Windows 2000 have been designed to work with Microsoft Image Color Management (ICM). ICM technology ensures that a color image, graphic object, or text object is rendered as closely as possible to its original intent on any device, despite differences in imaging technologies and color capabilities between devices. Whether you are scanning an image or other graphic on a color scanner, downloading it over the Internet, viewing or editing it onscreen, or printing it on paper, film, or other media, ICM 2.0 helps you keep colors consistent and accurate. For more information on ICM, see *About Image Color Management Version 2.0*.

There are various functions in the GDI that use or operate on color data. The following bitmap functions are enabled for use with ICM:

- **BitBlt**
- **CreateDIBitmap**
- **CreateDIBSection**
- **MaskBlt**
- **SetDIBColorTable**
- **SetDIBits**
- **SetDIBitsToDevice**
- **StretchBlt**
- **StretchDIBits**

Bitmap Reference

Bitmap Functions

AlphaBlend

The **AlphaBlend** function displays bitmaps that have transparent or semitransparent pixels.

```
BOOL AlphaBlend(
  HDC hdcDest,                 // handle to destination DC
  int nXOriginDest,            // x-coord of upper-left corner
  int nYOriginDest,            // y-coord of upper-left corner
  int nWidthDest,              // destination width
  int nHeightDest,             // destination height
  HDC hdcSrc,                  // handle to source DC
  int nXOriginSrc,             // x-coord of upper-left corner
  int nYOriginSrc,             // y-coord of upper-left corner
  int nWidthSrc,               // source width
  int nHeightSrc,              // source height
  BLENDFUNCTION blendFunction  // alpha-blending function
);
```

Parameter

hdcDest

[in] Handle to the destination device context.

nXOriginDest

[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nYOriginDest

[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nWidthDest

[in] Specifies the width, in logical units, of the destination rectangle.

nHeightDest

[in] Specifies the height, in logical units, of the destination rectangle.

hdcSrc

[in] Handle to the source device context.

nXOriginSrc

[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the source rectangle.

nYOriginSrc

[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the source rectangle.

nWidthSrc

[in] Specifies the width, in logical units, of the source rectangle.

nHeightSrc

[in] Specifies the height, in logical units, of the source rectangle.

blendFunction

[in] Specifies the alpha-blending function for source and destination bitmaps, a global alpha value to be applied to the entire source bitmap, and format information for the source bitmap. The source and destination blend functions are currently limited to AC_SRC_OVER. See the **BLENDFUNCTION** and **EMRALPHABLEND** structures.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If the source rectangle and destination rectangle are not the same size, the source bitmap is stretched to match the destination rectangle. If the **SetStretchBltMode** function is used, the *iStretchMode* value is automatically converted to COLORONCOLOR for this function (that is, BLACKONWHITE, WHITEONBLACK, and HALFTONE are changed to COLORONCOLOR).

The destination coordinates are transformed by using the transformation currently specified for the destination device context. The source coordinates are transformed by using the transformation currently specified for the source device context.

An error occurs (and the function returns FALSE) if the source device context identifies an enhanced metafile device context.

If destination and source bitmaps do not have the same color format, **AlphaBlend** converts the source bitmap to match the destination bitmap.

AlphaBlend does not support mirroring. If either the width or height of the source or destination is negative, this call will fail.

If the source and destination are the same surface—that is, they are both the screen or the same memory bitmap—and the source and destination rectangles overlap, an error occurs and the function returns FALSE.

The source rectangle must lie completely within the source surface; otherwise, an error occurs and the function returns FALSE.

AlphaBlend fails if the width or height of the source or destination is negative.

Note The **SourceConstantaAlpha** member of **BLENDFUNCTION** specifies an alpha transparency value to be used on the entire source bitmap. The **SourceConstantAlpha** value is combined with any per-pixel alpha values. If **SourceConstantAlpha** is 0, it is assumed that the image is transparent. Set the **SourceConstantAlpha** value to 255 (which indicates that the image is opaque) when you want to use only per-pixel alpha values.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

+ See Also

Bitmaps Overview, Bitmap Functions

BitBlt

The **BitBlt** function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context.

```
BOOL BitBlt(
  HDC hdcDest, // handle to destination DC
  int nXDest,  // x-coord of destination upper-left corner
  int nYDest,  // y-coord of destination upper-left corner
  int nWidth,  // width of destination rectangle
  int nHeight, // height of destination rectangle
  HDC hdcSrc,  // handle to source DC
  int nXSrc,   // x-coordinate of source upper-left corner
  int nYSrc,   // y-coordinate of source upper-left corner
  DWORD dwRop  // raster operation code
);
```

Parameters

hdcDest
[in] Handle to the destination device context.

nXDest
[in] Specifies the logical x-coordinate of the upper-left corner of the destination rectangle.

nYDest
[in] Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.

nWidth
[in] Specifies the logical width of the source and destination rectangles.

nHeight
[in] Specifies the logical height of the source and the destination rectangles.

hdcSrc
[in] Handle to the source device context.

nXSrc
[in] Specifies the logical x-coordinate of the upper-left corner of the source rectangle.

nYSrc
[in] Specifies the logical y-coordinate of the upper-left corner of the source rectangle.

dwRop
[in] Specifies a raster-operation code. These codes define how the color data for the source rectangle is to be combined with the color data for the destination rectangle to achieve the final color.

The following list shows some common raster operation codes:

Value	Description
BLACKNESS	Fills the destination rectangle using the color associated with index 0 in the physical palette. (This color is black for the default physical palette.)
CAPTUREBLT	**Windows 98, Windows 2000:** Includes any windows that are layered on top of your window in the resulting image. By default, the image contains only your window.
DSTINVERT	Inverts the destination rectangle.
MERGECOPY	Merges the colors of the source rectangle with the specified pattern by using the Boolean AND operator.
MERGEPAINT	Merges the colors of the inverted source rectangle with the colors of the destination rectangle by using the Boolean OR operator.
NOMIRRORBITMAP	**Windows 98, Windows 2000:** Prevents the bitmap from being mirrored.
NOTSRCCOPY	Copies the inverted source rectangle to the destination.
NOTSRCERASE	Combines the colors of the source and destination rectangles by using the Boolean OR operator, and then inverts the resultant color.
PATCOPY	Copies the specified pattern into the destination bitmap.
PATINVERT	Combines the colors of the specified pattern with the colors of the destination rectangle by using the Boolean XOR operator.
PATPAINT	Combines the colors of the pattern with the colors of the inverted source rectangle by using the Boolean OR operator. The result of this operation is combined with the colors of the destination rectangle by using the Boolean OR operator.
SRCAND	Combines the colors of the source and destination rectangles by using the Boolean AND operator.
SRCCOPY	Copies the source rectangle directly to the destination rectangle.
SRCERASE	Combines the inverted colors of the destination rectangle with the colors of the source rectangle by using the Boolean AND operator.
SRCINVERT	Combines the colors of the source and destination rectangles by using the Boolean XOR operator.
SRCPAINT	Combines the colors of the source and destination rectangles by using the Boolean OR operator.
WHITENESS	Fills the destination rectangle using the color associated with index 1 in the physical palette. (This color is white for the default physical palette.)

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If a rotation or shear transformation is in effect in the source device context, **BitBlt** returns an error. If other transformations exist in the source device context (and a matching transformation is *not* in effect in the destination device context), the rectangle in the destination device context is stretched, compressed, or rotated, as necessary.

If the color formats of the source and destination device contexts do not match, the **BitBlt** function converts the source color format to match the destination format.

When an enhanced metafile is being recorded, an error occurs if the source device context identifies an enhanced-metafile device context.

Not all devices support the **BitBlt** function. For more information, see the RC_BITBLT raster capability entry in the **GetDeviceCaps** function, as well as the following functions: **MaskBlt**, **PlgBlt**, and **StretchBlt**.

BitBlt returns an error if the source and destination device contexts represent different devices.

ICM: No color management is performed when blits occur.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions

CreateBitmap

The **CreateBitmap** function creates a bitmap with the specified width, height, and color format (color planes and bits-per-pixel).

```
HBITMAP CreateBitmap(
  int nWidth,         // bitmap width, in pixels
  int nHeight,        // bitmap height, in pixels
  UINT cPlanes,       // number of color planes
  UINT cBitsPerPel,   // number of bits to identify color
  CONST VOID *lpvBits // color data array
);
```

Parameters

nWidth

[in] Specifies the bitmap width, in pixels.

nHeight

[in] Specifies the bitmap height, in pixels.

cPlanes

[in] Specifies the number of color planes used by the device.

cBitsPerPel

[in] Specifies the number of bits required to identify the color of a single pixel.

lpvBits

[in] Pointer to an array of color data used to set the colors in a rectangle of pixels. Each scan line in the rectangle must be word aligned (scan lines that are not word aligned must be padded with zeros). If this parameter is NULL, the new bitmap is undefined.

Return Values

If the function succeeds, the return value is a handle to a bitmap.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

After a bitmap is created, it can be selected into a device context by calling the **SelectObject** function.

While the **CreateBitmap** function can be used to create color bitmaps. For performance reasons applications should use **CreateBitmap** to create monochrome bitmaps and **CreateCompatibleBitmap** to create color bitmaps. When a color bitmap returned from **CreateBitmap** is selected into a device context, the system must ensure that the bitmap matches the format of the device context into which it is being selected. Since **CreateCompatibleBitmap** takes a device context, it returns a bitmap that has the same format as the specified device context. Because of this, subsequent calls to **SelectObject** are faster than with a color bitmap returned from **CreateBitmap**.

If the bitmap is monochrome, zeros represent the foreground color, and ones represent the background color for the destination device context.

If an application sets the *nWidth* or *nHeight* parameter to zero, **CreateBitmap** returns the handle to a 1-pixel-by-1-pixel, monochrome bitmap.

When you no longer need the bitmap, call the **DeleteObject** function to delete it.

Windows 95/98: The created bitmap cannot exceed 16 MB in size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, **CreateDIBitmap**, **DeleteObject**, **GetBitmapBits**, **SelectObject**, **SetBitmapBits**

CreateBitmapIndirect

The **CreateBitmapIndirect** function creates a bitmap with the specified width, height, and color format (color planes and bits-per-pixel).

```
HBITMAP CreateBitmapIndirect(
  CONST BITMAP *lpbm    // bitmap data
);
```

Parameters

lpbm

[in] Pointer to a **BITMAP** structure that contains information about the bitmap. If an application sets the **bmWidth** or **bmHeight** members to zero, **CreateBitmapIndirect** returns the handle to a 1-pixel-by-1-pixel, monochrome bitmap.

Return Values

If the function succeeds, the return value is a handle to the bitmap.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

After a bitmap is created, it can be selected into a device context by calling the **SelectObject** function.

While the **CreateBitmapIndirect** function can be used to create color bitmaps, for performance reasons applications should use **CreateBitmapIndirect** to create monochrome bitmaps and **CreateCompatibleBitmap** to create color bitmaps. When a color bitmap returned from **CreateBitmapIndirect** is selected into a device context, the system must ensure that the bitmap matches the format of the device context into which it is being selected. Since **CreateCompatibleBitmap** takes a device context, it returns a bitmap that has the same format as the specified device context. Because of this, subsequent calls to **SelectObject** are faster than with a color bitmap returned from **CreateBitmapIndirect**.

If the bitmap is monochrome, zeros represent the foreground color, and ones represent the background color for the destination device context.

When you no longer need the bitmap, call the **DeleteObject** function to delete it.

Windows 95/98: The created bitmap cannot exceed 16 MB in size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BitBlt**, **BITMAP**, **CreateBitmap**, **CreateCompatibleBitmap**, **CreateDIBitmap**, **DeleteObject**, **SelectObject**

CreateCompatibleBitmap

The **CreateCompatibleBitmap** function creates a bitmap that is compatible with the device that is associated with the specified device context.

```
HBITMAP CreateCompatibleBitmap(
  HDC hdc,        // handle to DC
  int nWidth,     // width of bitmap, in pixels
  int nHeight     // height of bitmap, in pixels
);
```

Parameters

hdc
[in] Handle to a device context.

nWidth
[in] Specifies the bitmap width, in pixels.

nHeight
[in] Specifies the bitmap height, in pixels.

Return Values

If the function succeeds, the return value is a handle to the bitmap.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The color format of the bitmap created by the **CreateCompatibleBitmap** function matches the color format of the device identified by the *hdc* parameter. This bitmap can be selected into any memory device context that is compatible with the original device.

Because memory device contexts allow both color and monochrome bitmaps, the format of the bitmap returned by the **CreateCompatibleBitmap** function differs when the specified device context is a memory device context. However, a compatible bitmap that was created for a nonmemory device context always possesses the same color format and uses the same color palette as the specified device context.

Note When a memory device context is created, it initially has a 1-pixel-by-1-pixel, monochrome bitmap selected into it. If this memory device context is used in **CreateCompatibleBitmap**, the bitmap that is created is a *monochrome* bitmap.

To create a color bitmap, use the *hDC* that was used to create the memory device context, as shown in the following code:

```
HDC memDC = CreateCompatibleDC ( hDC );
HBITMAP memBM = CreateCompatibleBitmap ( hDC );
SelectObject ( memDC, memBM );
```

If an application sets the *nWidth* or *nHeight* parameters to zero, **CreateCompatibleBitmap** returns the handle to a 1-pixel-by-1-pixel, monochrome bitmap.

If a DIB section, which is a bitmap created by the **CreateDIBSection** function, is selected into the device context identified by the *hdc* parameter, **CreateCompatibleBitmap** creates a DIB section.

When you no longer need the bitmap, call the **DeleteObject** function to delete it.

Windows 95/98: The created bitmap cannot exceed 16 MB in size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **CreateDIBSection**, **DeleteObject**, **SelectObject**

CreateDIBitmap

The **CreateDIBitmap** function creates a DDB from a DIB and, optionally, sets the bitmap bits.

```
HBITMAP CreateDIBitmap(
  HDC hdc,                           // handle to DC
  CONST BITMAPINFOHEADER *lpbmih,    // bitmap data
  DWORD fdwInit,                     // initialization option
  CONST VOID *lpbInit,               // initialization data
  CONST BITMAPINFO *lpbmi,           // color-format data
  UINT fuUsage                       // color-data usage
);
```

Parameters

hdc

[in] Handle to a device context.

lpbmih

[in] Pointer to a bitmap information header structure, which may be one of those shown in the following table:

Operating system	Bitmap information header
Windows NT 3.51 and earlier	**BITMAPINFOHEADER**
Windows NT 4.0 and Windows 95	**BITMAPV4HEADER**
Windows 2000 and Windows 98	**BITMAPV5HEADER**

If *fdwInit* is CBM_INIT, the function uses the bitmap information header structure to obtain the desired width and height of the bitmap, as well as other information. Note that a positive value for the height indicates a bottom-up DIB while a negative value for the height indicates a top-down DIB. Calling **CreateDIBitmap** with *fdwInit* as CBM_INIT is equivalent to calling the **CreateCompatibleBitmap** function to create a DDB in the format of the device, and then calling the **SetDIBits** function to translate the DIB bits to the DDB.

fdwInit

[in] Specifies how the system initializes the bitmap bits. The following value is defined:

Value	Meaning
CBM_INIT	If this flag is set, the system uses the data pointed to by the *lpbInit* and *lpbmi* parameters to initialize the bitmap's bits. If this flag is clear, the data pointed to by those parameters is not used.

If *fdwInit* is zero, the system does not initialize the bitmap's bits.

lpbInit

[in] Pointer to an array of bytes containing the initial bitmap data. The format of the data depends on the **biBitCount** member of the **BITMAPINFO** structure to which the *lpbmi* parameter points.

lpbmi

[in] Pointer to a **BITMAPINFO** structure that describes the dimensions and color format of the array pointed to by the *lpbInit* parameter.

fuUsage

[in] Specifies whether the **bmiColors** member of the **BITMAPINFO** structure was initialized and, if so, whether **bmiColors** contains explicit red, green, blue (RGB) values or palette indexes. The *fuUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	A color table is provided and consists of an array of 16-bit indexes into the logical palette of the device context into which the bitmap is to be selected.
DIB_RGB_COLORS	A color table is provided and contains literal RGB values.

Return Values

If the function succeeds, the return value is a handle to the bitmap.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

For a device to reach optimal bitmap-drawing speed, specify *fdwInit* as CBM_INIT. Then, use the same color depth DIB as the video mode. When the video is running 4 bpp or 8 bpp, use DIB_PAL_COLORS.

The CBM_CREATDIB flag for the *fdwInit* parameter is no longer supported.

When you no longer need the bitmap, call the **DeleteObject** function to delete it.

ICM: The *fuUsage* parameter specifies whether or not the **bmiColors** member of **BITMAPINFO** pointed at by the *lpbmi* parameter contains color information. If **bmiColors** does not contain color information, no color management is performed for the bitmap. The **bmiHeader** member of **BITMAPINFO** must contain either **BITMAPV4HEADER** or **BITMAPV5HEADER** for color management to be enabled. The contents of the resulting bitmap are not color matched after the bitmap has been created.

Windows 95/98: The created bitmap cannot exceed 16 MB in size.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFO**, **BITMAPINFOHEADER**, **CreateCompatibleBitmap**, **DeleteObject**, **GetDeviceCaps**, **GetSystemPaletteEntries**, **SelectObject**, **SetDIBits**

CreateDIBSection

The **CreateDIBSection** function creates a DIB to which applications can write directly. The function gives you a pointer to the location of the bitmap's bit values. You can supply a handle to a file-mapping object that the function will use to create the bitmap, or you can let the system allocate the memory for the bitmap.

```
HBITMAP CreateDIBSection(
  HDC hdc,                   // handle to DC
  CONST BITMAPINFO *pbmi,    // bitmap data
  UINT iUsage,               // data type indicator
  VOID **ppvBits,            // bit values
  HANDLE hSection,           // handle to file mapping object
  DWORD dwOffset             // offset to bitmap bit values
);
```

Parameters

hdc

[in] Handle to a device context. If the value of *iUsage* is DIB_PAL_COLORS, the function uses this device context's logical palette to initialize the DIB's colors.

pbmi

[in] Pointer to a **BITMAPINFO** structure that specifies various attributes of the DIB, including the bitmap's dimensions and colors.

iUsage

[in] Specifies the type of data contained in the **bmiColors** array member of the **BITMAPINFO** structure pointed to by *pbmi* (either logical palette indexes or literal RGB values). The following values are defined:

Value	Meaning
DIB_PAL_COLORS	The **bmiColors** member is an array of 16-bit indexes into the logical palette of the device context specified by *hdc*.
DIB_RGB_COLORS	The **BITMAPINFO** structure contains an array of literal RGB values.

ppvBits

[out] Pointer to a variable that receives a pointer to the location of the DIB's bit values.

hSection

[in] Handle to a file-mapping object that the function will use to create the DIB. This parameter can be NULL.

If *hSection* is not NULL, it must be a handle to a file-mapping object created by calling the **CreateFileMapping** function with the PAGE_READWRITE or PAGE_WRITECOPY flag. Read-only DIB sections are not supported. Handles created by other means will cause **CreateDIBSection** to fail.

If *hSection* is not NULL, the **CreateDIBSection** function locates the bitmap's bit values at offset *dwOffset* in the file-mapping object referred to by *hSection*. An application can later retrieve the *hSection* handle by calling the **GetObject** function with the **HBITMAP** returned by **CreateDIBSection**.

If *hSection* is NULL, the system allocates memory for the DIB. In this case, the **CreateDIBSection** function ignores the *dwOffset* parameter. An application cannot later obtain a handle to this memory. The **dshSection** member of the **DIBSECTION** structure filled in by calling the **GetObject** function will be NULL.

dwOffset

[in] Specifies the offset from the beginning of the file-mapping object referenced by *hSection* where storage for the bitmap's bit values is to begin. This value is ignored if *hSection* is NULL. The bitmap's bit values are aligned on doubleword boundaries, so *dwOffset* must be a multiple of the size of a **DWORD**.

Return Values

If the function succeeds, the return value is a handle to the newly created DIB, and **ppvBits* points to the bitmap's bit values.

If the function fails, the return value is NULL, and **ppvBits* is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

As noted above, if *hSection* is NULL, the system allocates memory for the DIB. The system closes the handle to that memory when you later delete the DIB by calling the **DeleteObject** function. If *hSection* is not NULL, you must close the *hSection* memory handle yourself after calling **DeleteObject** to delete the bitmap.

Windows NT/2000: You need to guarantee that the **GDI** subsystem has completed any drawing to a bitmap created by **CreateDIBSection** before you draw to the bitmap yourself. Access to the bitmap must be synchronized. Do this by calling the **GdiFlush** function. This applies to any use of the pointer to the bitmap's bit values, including passing the pointer in calls to functions such as **SetDIBits**.

ICM: If the **bmiHeader** member of **BITMAPINFO** (pointed to by *pbmi*) does not contain **BITMAPV4HEADER** or **BITMAPV5HEADER**, no color management is done. Otherwise, color management is enabled, and the specified color space is associated with the bitmap.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFO**, **CreateFileMapping**, **DeleteObject**, **DIBSECTION**, **GdiFlush**, **GetDIBColorTable**, **GetObject**, **SetDIBits**, **SetDIBColorTable**

ExtFloodFill

The **ExtFloodFill** function fills an area of the display surface with the current brush.

```
BOOL ExtFloodFill(
  HDC hdc,          // handle to DC
  int nXStart,      // starting x-coordinate
  int nYStart,      // starting y-coordinate
  COLORREF crColor, // fill color
  UINT fuFillType   // fill type
);
```

Parameters

hdc
  [in] Handle to a device context.

nXStart

[in] Specifies the logical x-coordinate of the point where filling is to start.

nYStart

[in] Specifies the logical y-coordinate of the point where filling is to start.

crColor

[in] Specifies the color of the boundary or of the area to be filled. The interpretation of *crColor* depends on the value of the *fuFillType* parameter. To create a **COLORREF** color value, use the **RGB** macro.

fuFillType

[in] Specifies the type of fill operation to be performed. This parameter must be one of the following values:

Value	Meaning
FLOODFILLBORDER	The fill area is bounded by the color specified by the *crColor* parameter. This style is identical to the filling performed by the **FloodFill** function.
FLOODFILLSURFACE	The fill area is defined by the color that is specified by *crColor*. Filling continues outward in all directions as long as the color is encountered. This style is useful for filling areas with multicolored boundaries.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The following are some of the reasons this function might fail:

- The filling could not be completed.
- The specified point has the boundary color specified by the *crColor* parameter (if FLOODFILLBORDER was requested).
- The specified point does not have the color specified by *crColor* (if FLOODFILLSURFACE was requested).
- The point is outside the clipping region—that is, it is not visible on the device.

If the *fuFillType* parameter is FLOODFILLBORDER, the system assumes that the area to be filled is completely bounded by the color specified by the *crColor* parameter. The function begins filling at the point specified by the *nXStart* and *nYStart* parameters and continues in all directions until it reaches the boundary.

If *fuFillType* is FLOODFILLSURFACE, the system assumes that the area to be filled is a single color. The function begins to fill the area at the point specified by *nXStart* and *nYStart* and continues in all directions, filling all adjacent regions containing the color specified by *crColor*.

Only memory device contexts and devices that support raster-display operations support the **ExtFloodFill** function. To determine whether a device supports this technology, use the **GetDeviceCaps** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **COLORREF**, **FloodFill**, **GetDeviceCaps**, **RGB**

GetBitmapDimensionEx

The **GetBitmapDimensionEx** function retrieves the dimensions of a bitmap. The retrieved dimensions must have been set by the **SetBitmapDimensionEx** function.

```
BOOL GetBitmapDimensionEx(
  HBITMAP hBitmap,     // handle to bitmap
  LPSIZE lpDimension   // dimensions
);
```

Parameters

hBitmap
[in] Handle to the bitmap.

lpDimension
[out] Pointer to a **SIZE** structure to receive the bitmap dimensions.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The function returns a data structure that contains fields for the height and width of the bitmap. If those dimensions have not yet been set, the structure that is returned will have zeros in those fields.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **SetBitmapDimensionEx**, **SIZE**

GetDIBColorTable

The **GetDIBColorTable** function retrieves RGB (red, green, blue) color values from a range of entries in the color table of the DIB section bitmap that is currently selected into a specified device context.

```
UINT GetDIBColorTable(
  HDC hdc,              // handle to DC
  UINT uStartIndex,     // color table index of first entry
  UINT cEntries,        // number of entries to retrieve
  RGBQUAD *pColors      // array of color table entries
);
```

Parameters

hdc
[in] Handle to a device context. A DIB section bitmap must be selected into this device context.

uStartIndex
[in] A zero-based color table index that specifies the first color table entry to retrieve.

cEntries
[in] Specifies the number of color table entries to retrieve.

pColors
[out] Pointer to a buffer that receives an array of **RGBQUAD** data structures containing color information from the DIB's color table. The buffer must be large enough to contain as many **RGBQUAD** data structures as the value of *cEntries*.

Return Values

If the function succeeds, the return value is the number of color table entries that the function retrieves.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **GetDIBColorTable** function should be called to retrieve the color table for DIB section bitmaps that use 1, 4, or 8 bpp. The **biBitCount** member of a bitmap's associated **BITMAPINFOHEADER** structure specifies the number of bits per pixel. DIB section bitmaps with a **biBitCount** value greater than eight do not have a color table, but they do have associated color masks. Call the **GetObject** function to retrieve those color masks.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFOHEADER**, **CreateDIBSection**, **DIBSECTION**, **GetObject**, **RGBQUAD**, **SetDIBColorTable**

GetDIBits

The **GetDIBits** function retrieves the bits of the specified bitmap and copies them into a buffer using the specified format.

```
int GetDIBits(
  HDC hdc,           // handle to DC
  HBITMAP hbmp,      // handle to bitmap
  UINT uStartScan,   // first scan line to set
  UINT cScanLines,   // number of scan lines to copy
  LPVOID lpvBits,    // array for bitmap bits
  LPBITMAPINFO lpbi, // bitmap data buffer
  UINT uUsage        // RGB or palette index
);
```

Parameters

hdc
[in] Handle to the device context.

hbmp
[in] Handle to the bitmap.

uStartScan
[in] Specifies the first scan line to retrieve.

cScanLines
[in] Specifies the number of scan lines to retrieve.

lpvBits
[out] Pointer to a buffer to receive the bitmap data. If this parameter is NULL, the function passes the dimensions and format of the bitmap to the **BITMAPINFO** structure pointed to by the *lpbi* parameter.

lpbi
[in/out] Pointer to a **BITMAPINFO** structure that specifies the desired format for the DIB data.

uUsage
[in] Specifies the format of the **bmiColors** member of the **BITMAPINFO** structure. It must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table should consist of an array of 16-bit indexes into the current logical palette.
DIB_RGB_COLORS	The color table should consist of literal red, green, blue (RGB) values.

Return Values

If the *lpvBits* parameter is non-NULL and the function succeeds, the return value is the number of scan lines copied from the bitmap.

Windows 95/98: If the *lpvBits* parameter is NULL and **GetDIBits** successfully fills the **BITMAPINFO** structure, the return value is the total number of scan lines in the bitmap.

Windows NT/2000: If the *lpvBits* parameter is NULL and **GetDIBits** successfully fills the **BITMAPINFO** structure, the return value is non-zero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If the requested format for the DIB matches its internal format, the RGB values for the bitmap are copied. If the requested format does not match the internal format, a color table is synthesized. The following table describes the color table synthesized for each format:

Value	Meaning
1_BPP	The color table consists of a black and a white entry.
4_BPP	The color table consists of a mix of colors identical to the standard VGA palette.
8_BPP	The color table consists of a general mix of 256 colors defined by GDI. (Included in these 256 colors are the 20 colors found in the default logical palette.)
24_BPP	No color table is returned.

If the *lpvBits* parameter is a valid pointer, the first six members of the bitmap information header structure must be initialized to specify the size and format of the DIB. The scan lines must be aligned on a **DWORD** except for RLE compressed bitmaps.

A bitmap information header structure may be one of the following:

Operating system	Bitmap information header
Windows NT 3.51 and earlier	**BITMAPINFOHEADER**
Windows NT 4.0 and Windows 95	**BITMAPV4HEADER**
Windows 2000 and Windows 98	**BITMAPV5HEADER**

A bottom-up DIB is specified by setting the height to a positive number, while a top-down DIB is specified by setting the height to a negative number. The bitmap's color table will be appended to the **BITMAPINFO** structure.

If *lpvBits* is NULL, **GetDIBits** examines the first member of the first structure pointed to by *lpbi.* This member must specify the size, in bytes, of a **BITMAPCOREHEADER** or a bitmap information header structure. The function uses the specified size to determine how the remaining members should be initialized.

If *lpvBits* is NULL and the bit count member of **BITMAPINFO** is initialized to zero, **GetDIBits** fills in a bitmap information header structure or **BITMAPCOREHEADER** without the color table. This technique can be used to query bitmap attributes.

The bitmap identified by the *hbmp* parameter must not be selected into a device context when the application calls this function.

The origin for a bottom-up DIB is the lower-left corner of the bitmap; the origin for a top-down DIB is the upper-left corner.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions, **SetDIBits**

GetPixel

The **GetPixel** function retrieves the red, green, blue (RGB) color value of the pixel at the specified coordinates.

```
COLORREF GetPixel(
  HDC hdc,      // handle to DC
  int nXPos,    // x-coordinate of pixel
  int nYPos     // y-coordinate of pixel
);
```

Parameters

hdc
  [in] Handle to the device context.

nXPos
  [in] Specifies the logical x-coordinate of the pixel to be examined.

nYPos
  [in] Specifies the logical y-coordinate of the pixel to be examined.

Return Values

The return value is the RGB value of the pixel. If the pixel is outside of the current clipping region, the return value is CLR_INVALID.

Remarks

The pixel must be within the boundaries of the current clipping region.

Not all devices support **GetPixel**. An application should call **GetDeviceCaps** to determine whether a specified device supports this function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions, **COLORREF**, **GetDeviceCaps**, **SetPixel**

GetStretchBltMode

The **GetStretchBltMode** function retrieves the current stretching mode. The stretching mode defines how color data is added to or removed from bitmaps that are stretched or compressed when the **StretchBlt** function is called.

```
int GetStretchBltMode(
  HDC hdc  // handle to DC
);
```

Parameters

hdc
[in] Handle to the device context.

Return Values

If the function succeeds, the return value is the current stretching mode.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions

GradientFill

The **GradientFill** function fills rectangle and triangle structures.

```
BOOL GradientFill(
  HDC hdc,                   // handle to DC
  PTRIVERTEX pVertex,        // array of vertices
  ULONG dwNumVertex,         // number of vertices
  PVOID pMesh,               // array of gradients
  ULONG dwNumMesh,           // size of gradient array
  ULONG dwMode               // gradient fill mode
);
```

Parameters

hdc

[in] Handle to the destination device context.

pVertex

[in] Pointer to an array of **TRIVERTEX** structures that each define a triangle vertex.

dwNumVertex

[in] The number of vertices in *pVertex*.

pMesh

[in] Array of **GRADIENT_TRIANGLE** structures in triangle mode, or an array of **GRADIENT_RECT** structures in rectangle mode.

dwNumMesh

[in] The number of elements (triangles or rectangles) in *pMesh*.

dwMode

[in] Specifies gradient fill mode. This parameter can be one of the following values:

Value	Meaning
GRADIENT_FILL_RECT_H	In this mode, two endpoints describe a rectangle. The rectangle is defined to have a constant color (specified by the **TRIVERTEX** structure) for the left and right edges. GDI interpolates the color from the top to bottom edge and fills the interior.
GRADIENT_FILL_RECT_V	In this mode, two endpoints describe a rectangle. The rectangle is defined to have a constant color (specified by the **TRIVERTEX** structure) for the top and bottom edges. GDI interpolates the color from the top to bottom edge and fills the interior.
GRADIENT_FILL_TRIANGLE	In this mode, an array of **TRIVERTEX** structures is passed to GDI along with a list of array indexes that describe separate triangles. GDI performs linear interpolation between triangle vertices and fills the interior. Drawing is done directly in 24-bpp and 32-bpp modes. Dithering is performed in 16-bpp, 8-bpp, 4-bpp, and 1-bpp mode.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

To add smooth shading to a triangle, call the **GradientFill** function with the three triangle endpoints. GDI will linearly interpolate and fill the triangle.

To add smooth shading to a rectangle, call **GradientFill** with the upper-left and lower-right coordinates of the rectangle. There are two shading modes used when drawing a rectangle. In horizontal mode, the rectangle is shaded from left to right. In vertical mode, the rectangle is shaded from top to bottom.

The **GradientFill** function uses a mesh method to specify the endpoints of the object to draw. All vertices are passed to **GradientFill** in the *pVertex* array. The *pMesh* parameter specifies how these vertices are connected to form an object. When filling a rectangle, *pMesh* points to an array of **GRADIENT_RECT** structures. Each **GRADIENT_RECT** structure specifies the index of two vertices in the *pVertex* array. These two vertices form the upper-left and lower-right boundary of one rectangle.

In the case of filling a triangle, *pMesh* points to an array of **GRADIENT_TRIANGLE** structures. Each **GRADIENT_TRIANGLE** structure specifies the index of three vertices in the *pVertex* array. These three vertices form one triangle.

In order to simplify hardware acceleration, this routine is not required to be pixel-perfect in the triangle interior.

For more information, see *Smooth Shading*, *Drawing a Shaded Triangle*, and *Drawing a Shaded Rectangle*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

See Also

Bitmaps Overview, Bitmap Functions, **EMRGRADIENTFILL**, **GRADIENT_RECT**, **GRADIENT_TRIANGLE**, **TRIVERTEX**

LoadBitmap

The **LoadBitmap** function loads the specified bitmap resource from a module's executable file. This function has been superseded by the **LoadImage** function.

```
HBITMAP LoadBitmap(
  HINSTANCE hInstance,  // handle to application instance
  LPCTSTR lpBitmapName  // name of bitmap resource
);
```

Parameters

hInstance

[in] Handle to the instance of the module whose executable file contains the bitmap to be loaded.

lpBitmapName

[in] Pointer to a null-terminated string that contains the name of the bitmap resource to be loaded. Alternatively, this parameter can consist of the resource identifier in the low-order word and zero in the high-order word. The **MAKEINTRESOURCE** macro can be used to create this value.

Return Values

If the function succeeds, the return value is the handle to the specified bitmap.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If the bitmap pointed to by the *lpBitmapName* parameter does not exist or there is insufficient memory to load the bitmap, the function fails.

An application can use the **LoadBitmap** function to access the predefined bitmaps used by the Win32 API. To do so, the application must set the *hInstance* parameter to NULL and the *lpBitmapName* parameter to one of the following values:

OBM_BTNCORNERS
OBM_BTSIZE
OBM_CHECK
OBM_CHECKBOXES
OBM_CLOSE
OBM_COMBO
OBM_DNARROW
OBM_DNARROWD
OBM_DNARROWI
OBM_LFARROW
OBM_LFARROWD
OBM_LFARROWI
OBM_MNARROW
OBM_OLD_CLOSE
OBM_OLD_DNARROW
OBM_OLD_LFARROW
OBM_OLD_REDUCE
OBM_OLD_RESTORE
OBM_OLD_RGARROW
OBM_OLD_UPARROW
OBM_OLD_ZOOM
OBM_REDUCE
OBM_REDUCED
OBM_RESTORE
OBM_RESTORED
OBM_RGARROW
OBM_RGARROWD
OBM_RGARROWI
OBM_SIZE
OBM_UPARROW
OBM_UPARROWD
OBM_UPARROWI
OBM_ZOOM
OBM_ZOOMD

Bitmap names that begin with OBM_OLD represent bitmaps used by 16-bit versions of Windows earlier than 3.0.

For an application to use any of the **OBM_** constants, the constant **OEMRESOURCE** must be defined before the Windows.h header file is included.

The application must call the **DeleteObject** function to delete each bitmap handle returned by the **LoadBitmap** function.

Windows 95 has a problem dealing with Win32 .exe or .dll files that contain resources whose size is 64 KB or larger. To retain Win16 compatibility, Windows 95 converts the 32-bit size into a 16-bit size and a shift count. When it does this conversion it rounds down instead of up, so some bytes can be lost. In addition, Win16 uses the same shift count for all resources, thus the shift required for a large resource can cause a small resource to be severely truncated, or even eliminated completely.

To avoid this problem, compute the scaling factor for the largest resource and pad all resources with zeros so each is a multiple of the scaling factor. For example, a resource of size 0x100065 is converted to 0x8003 * 32, which loses 5 bytes. To save the 5 bytes, you must pad the resource with 27 zeros so that it becomes size 0x100080 and is then converted to 0x8004 * 32. And any smaller resource must also be padded with zeros so it is a multiple of the scaling factor, which in this case is 32.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Bitmaps Overview, Bitmap Functions, **CreateBitmap**, **DeleteObject**, **LoadCursor**, **LoadIcon**, **LoadImage**, **MAKEINTRESOURCE**

MaskBlt

The **MaskBlt** function combines the color data for the source and destination bitmaps using the specified mask and raster operation.

```
BOOL MaskBlt(
  HDC hdcDest,         // handle to destination DC
  int nXDest,          // x-coord of destination upper-left corner
  int nYDest,          // y-coord of destination upper-left corner
  int nWidth,          // width of source and destination
  int nHeight,         // height of source and destination
  HDC hdcSrc,          // handle to source DC
  int nXSrc,           // x-coord of upper-left corner of source
  int nYSrc,           // y-coord of upper-left corner of source
  HBITMAP hbmMask,     // handle to monochrome bit mask
  int xMask,           // horizontal offset into mask bitmap
  int yMask,          // vertical offset into mask bitmap
  DWORD dwRop         // raster operation code
);
```

Parameters

hdcDest

[in] Handle to the destination device context.

nXDest

[in] Specifies the logical x-coordinate of the upper-left corner of the destination rectangle.

nYDest

[in] Specifies the logical y-coordinate of the upper-left corner of the destination rectangle.

nWidth

[in] Specifies the width, in logical units, of the destination rectangle and source bitmap.

nHeight

[in] Specifies the height, in logical units, of the destination rectangle and source bitmap.

hdcSrc

[in] Handle to the device context from which the bitmap is to be copied. It must be zero if the *dwRop* parameter specifies a raster operation that does not include a source.

nXSrc

[in] Specifies the logical x-coordinate of the upper-left corner of the source bitmap.

nYSrc

[in] Specifies the logical y-coordinate of the upper-left corner of the source bitmap.

hbmMask

[in] Handle to the monochrome mask bitmap combined with the color bitmap in the source device context.

xMask

[in] Specifies the horizontal pixel offset for the mask bitmap specified by the *hbmMask* parameter.

yMask

[in] Specifies the vertical pixel offset for the mask bitmap specified by the *hbmMask* parameter.

dwRop

[in] Specifies both foreground and background ternary raster operation codes that the function uses to control the combination of source and destination data. The background raster operation code is stored in the high-order byte of the high-order word of this value; the foreground raster operation code is stored in the low-order byte of the high-order word of this value; the low-order word of this value is ignored, and should be zero. The macro **MAKEROP4** creates such combinations of foreground and background raster operation codes.

For a discussion of foreground and background in the context of this function, see the following Remarks section.

For a list of common raster operation codes, see the **BitBlt** function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A value of 1 in the mask specified by *hbmMask* indicates that the foreground raster operation code specified by *dwRop* should be applied at that location. A value of 0 in the mask indicates that the background raster operation code specified by *dwRop* should be applied at that location.

If the raster operations require a source, the mask rectangle must cover the source rectangle. If it does not, the function will fail. If the raster operations do not require a source, the mask rectangle must cover the destination rectangle. If it does not, the function will fail.

If a rotation or shear transformation is in effect for the source device context when this function is called, an error occurs. However, other types of transformation are allowed.

If the color formats of the source, pattern, and destination bitmaps differ, this function converts the pattern or source format, or both, to match the destination format.

If the mask bitmap is not a monochrome bitmap, an error occurs.

When an enhanced metafile is being recorded, an error occurs (and the function returns FALSE) if the source device context identifies an enhanced-metafile device context.

Not all devices support the **MaskBlt** function. An application should call the **GetDeviceCaps** function to determine whether a device supports this function.

If no mask bitmap is supplied, this function behaves exactly like **BitBlt**, using the foreground raster operation code.

ICM: No color management is performed when blits occur.

Windows 98, Windows 2000: When used in a multimonitor system, both *hdcSrc* and *hdcDest* must refer to the same device or the function will fail.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BitBlt**, **GetDeviceCaps**, **PlgBlt**, **StretchBlt**

PlgBlt

The **PlgBlt** function performs a bit-block transfer of the bits of color data from the specified rectangle in the source device context to the specified parallelogram in the destination device context. If the given bitmask handle identifies a valid monochrome bitmap, the function uses this bitmap to mask the bits of color data from the source rectangle.

```
BOOL PlgBlt(
  HDC hdcDest,           // handle to destination DC
  CONST POINT *lpPoint,  // destination vertices
  HDC hdcSrc,            // handle to source DC
  int nXSrc,             // x-coord of source upper-left corner
  int nYSrc,             // y-coord of source upper-left corner
  int nWidth,            // width of source rectangle
  int nHeight,           // height of source rectangle
  HBITMAP hbmMask,       // handle to bitmask
  int xMask,             // x-coord of bitmask upper-left corner
  int yMask              // y-coord of bitmask upper-left corner
);
```

Parameters

hdcDest
[in] Handle to the destination device context.

lpPoint
[in] Pointer to an array of three points in logical space that identify three corners of the destination parallelogram. The upper-left corner of the source rectangle is mapped to the first point in this array, the upper-right corner to the second point in this array, and the lower-left corner to the third point. The lower-right corner of the source rectangle is mapped to the implicit fourth point in the parallelogram.

hdcSrc
[in] Handle to the source device context.

nXSrc
[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the source rectangle.

nYSrc
[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the source rectangle.

nWidth
[in] Specifies the width, in logical units, of the source rectangle.

nHeight
[in] Specifies the height, in logical units, of the source rectangle.

hbmMask
[in] Handle to an optional monochrome bitmap that is used to mask the colors of the source rectangle.

xMask
[in] Specifies the x-coordinate of the upper-left corner of the monochrome bitmap.

yMask
[in] Specifies the y-coordinate of the upper-left corner of the monochrome bitmap.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The fourth vertex of the parallelogram (*D*) is defined by treating the first three points (*A*, *B*, and *C*) as vectors and computing $D = B + C - A$.

If the bitmask exists, a value of one in the mask indicates that the source pixel color should be copied to the destination. A value of zero in the mask indicates that the destination pixel color is not to be changed. If the mask rectangle is smaller than the source and destination rectangles, the function replicates the mask pattern.

Scaling, translation, and reflection transformations are allowed in the source device context; however, rotation and shear transformations are not. If the mask bitmap is not a monochrome bitmap, an error occurs. The stretching mode for the destination device context is used to determine how to stretch or compress the pixels, if that is necessary.

When an enhanced metafile is being recorded, an error occurs if the source device context identifies an enhanced-metafile device context.

The destination coordinates are transformed according to the destination device context; the source coordinates are transformed according to the source device context. If the source transformation has a rotation or shear, an error is returned.

If the destination and source rectangles do not have the same color format, **PlgBlt** converts the source rectangle to match the destination rectangle.

Not all devices support the **PlgBlt** function. For more information, see the description of the RC_BITBLT raster capability in the **GetDeviceCaps** function.

If the source and destination device contexts represent incompatible devices, **PlgBlt** returns an error.

Windows 98, Windows 2000: When used in a multimonitor system, both *hdcSrc* and *hdcDest* must refer to the same device or the function will fail.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BitBlt**, **GetDeviceCaps**, **MaskBlt**, **SetStretchBltMode**, **StretchBlt**

SetBitmapDimensionEx

The **SetBitmapDimensionEx** function assigns preferred dimensions to a bitmap. These dimensions can be used by applications; however, they are not used by the system.

```
BOOL SetBitmapDimensionEx(
  HBITMAP hBitmap,  // handle to bitmap
  int nWidth,       // bitmap width in .01-mm units
  int nHeight,      // bitmap height in .01-mm units
  LPSIZE lpSize     // original dimensions
);
```

Parameters

hBitmap
  [in] Handle to the bitmap. The bitmap cannot be a DIB-section bitmap.

nWidth
  [in] Specifies the width, in 0.1-millimeter units, of the bitmap.

nHeight
  [in] Specifies the height, in 0.1-millimeter units, of the bitmap.

lpSize
  [out] Pointer to a **SIZE** structure to receive the previous dimensions of the bitmap. This pointer can be NULL.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can retrieve the dimensions assigned to a bitmap with the **SetBitmapDimensionEx** function by calling the **GetBitmapDimensionEx** function.

The bitmap identified by *hBitmap* cannot be a DIB section, which is a bitmap created by the **CreateDIBSection** function. If the bitmap is a DIB section, the **SetBitmapDimensionEx** function fails.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **CreateDIBSection**, **GetBitmapDimensionEx**, **SIZE**

SetDIBColorTable

The **SetDIBColorTable** function sets RGB (red, green, blue) color values in a range of entries in the color table of the DIB that is currently selected into a specified device context.

```
UINT SetDIBColorTable(
  HDC hdc,               // handle to DC
  UINT uStartIndex,      // color table index of first entry
  UINT cEntries,         // number of color table entries
  CONST RGBQUAD *pColors // array of color table entries
);
```

Parameters

hdc

[in] Specifies a device context. A DIB must be selected into this device context.

uStartIndex

[in] A zero-based color table index that specifies the first color table entry to set.

cEntries

[in] Specifies the number of color table entries to set.

pColors

[in] Pointer to an array of **RGBQUAD** structures containing new color information for the DIB's color table.

Return Values

If the function succeeds, the return value is the number of color table entries that the function sets.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This function should be called to set the color table for DIBs that use 1 bpp, 4 bpp, or 8 bpp. The **BitCount** member of a bitmap's associated bitmap information header structure.

A bitmap information header structure may be one of the following:

Operating system	Bitmap information header
Windows NT 3.51 and earlier	**BITMAPINFOHEADER**
Windows NT 4.0 and Windows 95	**BITMAPV4HEADER**
Windows 2000 and Windows 98	**BITMAPV5HEADER**

BITMAPINFOHEADER structure specifies the number of bits per pixel. Device-independent bitmaps with a **biBitCount** value greater than 8 do not have a color table.

Windows NT 4.0 and Windows 95: The **bV4BitCount** member of a bitmap's associated **BITMAPV4HEADER** structure specifies the number of bits per pixel. Device-independent bitmaps with a **bV4BitCount** value greater than 8 do not have a color table.

Windows 2000 and Windows 98: The **bV5BitCount** member of a bitmap's associated **BITMAPV5HEADER** structure specifies the number of bits per pixel. Device-independent bitmaps with a **bV5BitCount** value greater than 8 do not have a color table.

ICM: No color management is performed.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFOHEADER**, **CreateDIBSection**, **DIBSECTION**, **GetDIBColorTable**, **GetObject**, **RGBQUAD**

SetDIBits

The **SetDIBits** function sets the pixels in a bitmap using the color data found in the specified DIB.

```
int SetDIBits(
  HDC hdc,                  // handle to DC
  HBITMAP hbmp,             // handle to bitmap
  UINT uStartScan,          // starting scan line
  UINT cScanLines,          // number of scan lines
  CONST VOID *lpvBits,      // array of bitmap bits
  CONST BITMAPINFO *lpbmi,  // bitmap data
  UINT fuColorUse           // type of color indexes to use
);
```

Parameters

hdc

[in] Handle to a device context.

hbmp

[in] Handle to the bitmap that is to be altered using the color data from the specified DIB.

uStartScan

[in] Specifies the starting scan line for the device-independent color data in the array pointed to by the *lpvBits* parameter.

cScanLines

[in] Specifies the number of scan lines found in the array containing device-independent color data.

lpvBits

[in] Pointer to the DIB color data, stored as an array of bytes. The format of the bitmap values depends on the **biBitCount** member of the **BITMAPINFO** structure pointed to by the *lpbmi* parameter.

lpbmi

[in] Pointer to a **BITMAPINFO** structure that contains information about the DIB.

fuColorUse

[in] Specifies whether the **bmiColors** member of the **BITMAPINFO** structure was provided and, if so, whether **bmiColors** contains explicit red, green, blue (RGB) values or palette indexes. The *fuColorUse* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the logical palette of the device context identified by the *hdc* parameter.
DIB_RGB_COLORS	The color table is provided and contains literal RGB values.

Return Values

If the function succeeds, the return value is the number of scan lines copied.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Optimal bitmap drawing speed is obtained when the bitmap bits are indexes into the system palette.

Applications can retrieve the system palette colors and indexes by calling the **GetSystemPaletteEntries** function. After the colors and indexes are retrieved, the application can create the DIB. For more information, see *System Palette*.

The device context identified by the *hdc* parameter is used only if the DIB_PAL_COLORS constant is set for the *fuColorUse* parameter; otherwise it is ignored.

The bitmap identified by the *hbmp* parameter must not be selected into a device context when the application calls this function.

The scan lines must be aligned on a **DWORD** except for RLE-compressed bitmaps.

The origin for bottom-up DIBs is the lower-left corner of the bitmap; the origin for top-down DIBs is the upper-left corner of the bitmap.

ICM: Color management is performed. If the specified **BITMAPINFO** structure is not **BITMAPV4HEADER** or **BITMAPV5HEADER**, the color profile of the current device context is used as the source color space profile. If the **BITMAPINFO** structure is not **BITMAPV4HEADER** or **BITMAPV5HEADER**, the sRGB color space is used. If the specified **BITMAPINFO** structure is **BITMAPV4HEADER** or **BITMAPV5HEADER**, the color space profile associated with the bitmap is used as the source color space.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFO**, **GetDIBits**, **GetSystemPaletteEntries**

SetDIBitsToDevice

The **SetDIBitsToDevice** function sets the pixels in the specified rectangle on the device that is associated with the destination device context using color data from a DIB .

Windows 98 and Windows 2000: SetDIBitsToDevice has been extended to allow a JPEG or PNG image to be passed as the source image.

```
int SetDIBitsToDevice(
  HDC hdc,                      // handle to DC
  int XDest,                    // x-coord of destination upper-left corner
  int YDest,                    // y-coord of destination upper-left corner
  DWORD dwWidth,                // source rectangle width
  DWORD dwHeight,               // source rectangle height
  int XSrc,                     // x-coord of source lower-left corner
  int YSrc,                     // y-coord of source lower-left corner
  UINT uStartScan,              // first scan line in array
  UINT cScanLines,              // number of scan lines
  CONST VOID *lpvBits,          // array of DIB bits
  CONST BITMAPINFO *lpbmi,      // bitmap information
  UINT fuColorUse               // RGB or palette indexes
);
```

Parameters

hdc
: [in] Handle to the device context.

XDest
: [in] Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

YDest
: [in] Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

dwWidth
: [in] Specifies the width, in logical units, of the DIB.

dwHeight
: [in] Specifies the height, in logical units, of the DIB.

XSrc
: [in] Specifies the x-coordinate, in logical units, of the lower-left corner of the DIB.

YSrc
[in] Specifies the y-coordinate, in logical units, of the lower-left corner of the DIB.

uStartScan
[in] Specifies the starting scan line in the DIB.

cScanLines
[in] Specifies the number of DIB scan lines contained in the array pointed to by the *lpvBits* parameter.

lpvBits
[in] Pointer to DIB color data stored as an array of bytes. For more information, see the following Remarks section.

lpbmi
[in] Pointer to a **BITMAPINFO** structure that contains information about the DIB.

fuColorUse
[in] Specifies whether the **bmiColors** member of the **BITMAPINFO** structure contains explicit red, green, blue (RGB) values or indexes into a palette. For more information, see the following Remarks section.

The *fuColorUse* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indexes into the currently selected logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

Return Values

If the function succeeds, the return value is the number of scan lines set.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Windows 98 and Windows 2000: If the driver cannot support the JPEG or PNG file image passed to **SetDIBitsToDevice**, the function will fail and return GDI_ERROR. If failure does occur, the application must fall back on its own JPEG or PNG support to decompress the image into a bitmap, and then pass the bitmap to **SetDIBitsToDevice**.

Remarks

Optimal bitmap drawing speed is obtained when the bitmap bits are indexes into the system palette.

Applications can retrieve the system palette colors and indexes by calling the **GetSystemPaletteEntries** function. After the colors and indexes are retrieved, the application can create the DIB. For more information about the system palette, see *Colors*.

The origin of a bottom-up DIB is the lower-left corner of the bitmap; the origin of a top-down DIB is the upper-left corner.

To reduce the amount of memory required to set bits from a large DIB on a device surface, an application can band the output by repeatedly calling **SetDIBitsToDevice**, placing a different portion of the bitmap into the *lpvBits* array each time. The values of the *uStartScan* and *cScanLines* parameters identify the portion of the bitmap contained in the *lpvBits* array.

The **SetDIBitsToDevice** function returns an error if it is called by a process that is running in the background while a full-screen MS-DOS session runs in the foreground.

Windows 98, Windows 2000:

- If the **biCompression** member of **BITMAPINFOHEADER** is BI_JPEG or BI_PNG, *lpvBits* points to a buffer containing a JPEG or PNG image. The **biSizeImage** member of specifies the size of the buffer. The *fuColorUse* parameter must be set to DIB_RGB_COLORS.
- If the **bV4Compression** member of **BITMAPV4HEADER** is BI_JPEG or BI_PNG, *lpvBits* points to a buffer containing a JPEG or PNG image. The **bV4SizeImage** member of **BITMAPV4HEADER** specifies the size of the buffer. The *fuColorUse* parameter must be set to DIB_RGB_COLORS.
- If the **bV5Compression** member of **BITMAPV5HEADER** is BI_JPEG or BI_PNG, *lpvBits* points to a buffer containing a JPEG or PNG image. The **bV5SizeImage** member of **BITMAPV5HEADER** specifies the size of the buffer. The *fuColorUse* parameter must be set to DIB_RGB_COLORS.
- To ensure proper metafile spooling while printing, applications must call the CHECKJPEGFORMAT or CHECKPNGFORMAT escape to verify that the printer recognizes the JPEG or PNG image, respectively, before calling **SetDIBitsToDevice**.

ICM: Color management is performed. If the specified **BITMAPINFO** structure is not **BITMAPV4HEADER** or **BITMAPV5HEADER**, the color profile of the current device context is used as the source color space profile. If the **BITMAPINFO** structure is not **BITMAPV4HEADER** or **BITMAPV5HEADER**, the sRGB color space is used. If the specified **BITMAPINFO** structure is **BITMAPV4HEADER** or **BITMAPV5HEADER**, the color space profile associated with the bitmap is used as the source color space.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFO**, **GetSystemPaletteEntries**, **SetDIBits**, **StretchDIBits**

SetPixel

The **SetPixel** function sets the pixel at the specified coordinates to the specified color.

```
COLORREF SetPixel(
  HDC hdc,            // handle to DC
  int X,              // x-coordinate of pixel
  int Y,              // y-coordinate of pixel
  COLORREF crColor    // pixel color
);
```

Parameters

hdc

[in] Handle to the device context.

X

[in] Specifies the x-coordinate, in logical units, of the point to be set.

Y

[in] Specifies the y-coordinate, in logical units, of the point to be set.

crColor

[in] Specifies the color to be used to paint the point. To create a **COLORREF** color value, use the **RGB** macro.

Return Values

If the function succeeds, the return value is the RGB value that the function sets the pixel to. This value may differ from the color specified by *crColor*; that occurs when an exact match for the specified color cannot be found.

If the function fails, the return value is –1.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The function fails if the pixel coordinates lie outside of the current clipping region.

Not all devices support the **SetPixel** function. For more information, see **GetDeviceCaps**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.

Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Bitmaps Overview, Bitmap Functions, **COLORREF**, **GetDeviceCaps**, **GetPixel**, **RGB**, **SetPixelV**

SetPixelV

The **SetPixelV** function sets the pixel at the specified coordinates to the closest approximation of the specified color. The point must be in the clipping region and the visible part of the device surface.

```
BOOL SetPixelV(
  HDC hdc,           // handle to device context
  int X,             // x-coordinate of pixel
  int Y,             // y-coordinate of pixel
  COLORREF crColor   // new pixel color
);
```

Parameters

hdc

[in] Handle to the device context.

X

[in] Specifies the x-coordinate, in logical units, of the point to be set.

Y

[in] Specifies the y-coordinate, in logical units, of the point to be set.

crColor

[in] Specifies the color to be used to paint the point. To create a **COLORREF** color value, use the **RGB** macro.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Not all devices support the **SetPixelV** function. For more information, see the description of the RC_BITBLT capability in the **GetDeviceCaps** function.

SetPixelV is faster than **SetPixel** because it does not need to return the color value of the point actually painted.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **COLORREF**, **GetDeviceCaps**, **RGB**, **SetPixel**

SetStretchBltMode

The **SetStretchBltMode** function sets the bitmap stretching mode in the specified device context.

```
int SetStretchBltMode(
  HDC hdc,           // handle to DC
  int iStretchMode   // bitmap stretching mode
);
```

Parameters

hdc

[in] Handle to the device context.

iStretchMode

[in] Specifies the stretching mode. This parameter can be one of the following values:

Value	Description
BLACKONWHITE	Performs a Boolean AND operation using the color values for the eliminated and existing pixels. If the bitmap is a monochrome bitmap, this mode preserves black pixels at the expense of white pixels.
COLORONCOLOR	Deletes the pixels. This mode deletes all eliminated lines of pixels without trying to preserve their information.

(continued)

(continued)

Value	Description
HALFTONE	Maps pixels from the source rectangle into blocks of pixels in the destination rectangle. The average color over the destination block of pixels approximates the color of the source pixels. After setting the HALFTONE stretching mode, an application must call the **SetBrushOrgEx** function to set the brush origin. If it fails to do so, brush misalignment occurs. This is not supported on Windows 95/98.
STRETCH_ANDSCANS	Same as BLACKONWHITE.
STRETCH_DELETESCANS	Same as COLORONCOLOR.
STRETCH_HALFTONE	Same as HALFTONE.
STRETCH_ORSCANS	Same as WHITEONBLACK.
WHITEONBLACK	Performs a Boolean OR operation using the color values for the eliminated and existing pixels. If the bitmap is a monochrome bitmap, this mode preserves white pixels at the expense of black pixels.

Return Values

If the function succeeds, the return value is the previous stretching mode.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call **GetLastError**.

Remarks

The stretching mode defines how the system combines rows or columns of a bitmap with existing pixels on a display device when an application calls the **StretchBlt** function.

The BLACKONWHITE (STRETCH_ANDSCANS) and WHITEONBLACK (STRETCH_ORSCANS) modes are typically used to preserve foreground pixels in monochrome bitmaps. The COLORONCOLOR (STRETCH_DELETESCANS) mode is typically used to preserve color in color bitmaps.

The HALFTONE mode is slower and requires more processing of the source image than the other three modes; but produces higher quality images. Also note that **SetBrushOrgEx** must be called after setting the HALFTONE mode to avoid brush misalignment.

Additional stretching modes might also be available depending on the capabilities of the device driver.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **GetStretchBltMode**, **SetBrushOrgEx**, **StretchBlt**

StretchBlt

The **StretchBlt** function copies a bitmap from a source rectangle into a destination rectangle, stretching or compressing the bitmap to fit the dimensions of the destination rectangle, if necessary. The system stretches or compresses the bitmap according to the stretching mode currently set in the destination device context.

```
BOOL StretchBlt(
  HDC hdcDest,      // handle to destination DC
  int nXOriginDest, // x-coord of destination upper-left corner
  int nYOriginDest, // y-coord of destination upper-left corner
  int nWidthDest,   // width of destination rectangle
  int nHeightDest,  // height of destination rectangle
  HDC hdcSrc,       // handle to source DC
  int nXOriginSrc,  // x-coord of source upper-left corner
  int nYOriginSrc,  // y-coord of source upper-left corner
  int nWidthSrc,    // width of source rectangle
  int nHeightSrc,   // height of source rectangle
  DWORD dwRop       // raster operation code
);
```

Parameters

hdcDest
[in] Handle to the destination device context.

nXOriginDest
[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nYOriginDest
[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nWidthDest
[in] Specifies the width, in logical units, of the destination rectangle.

nHeightDest
[in] Specifies the height, in logical units, of the destination rectangle.

hdcSrc
[in] Handle to the source device context.

nXOriginSrc
[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the source rectangle.

nYOriginSrc
[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the source rectangle.

nWidthSrc
[in] Specifies the width, in logical units, of the source rectangle.

nHeightSrc
[in] Specifies the height, in logical units, of the source rectangle.

dwRop
[in] Specifies the raster operation to be performed. Raster operation codes define how the system combines colors in output operations that involve a brush, a source bitmap, and a destination bitmap.

See **BitBlt** for a list of common raster operation codes.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

StretchBlt stretches or compresses the source bitmap in memory and then copies the result to the destination rectangle. The color data for pattern or destination pixels is merged after the stretching or compression occurs.

When an enhanced metafile is being recorded, an error occurs (and the function returns FALSE) if the source device context identifies an enhanced-metafile device context.

If the specified raster operation requires a brush, the system uses the brush currently selected into the destination device context.

The destination coordinates are transformed by using the transformation currently specified for the destination device context; the source coordinates are transformed by using the transformation currently specified for the source device context.

If the source transformation has a rotation or shear, an error occurs.

If destination, source, and pattern bitmaps do not have the same color format, **StretchBlt** converts the source and pattern bitmaps to match the destination bitmap.

If **StretchBlt** must convert a monochrome bitmap to a color bitmap, it sets white bits (1) to the background color and black bits (0) to the foreground color. To convert a color bitmap to a monochrome bitmap, it sets pixels that match the background color to white (1) and sets all other pixels to black (0). The foreground and background colors of the device context with color are used.

StretchBlt creates a mirror image of a bitmap if the signs of the *nWidthSrc* and *nWidthDest* parameters or of the *nHeightSrc* and *nHeightDest* parameters differ. If *nWidthSrc* and *nWidthDest* have different signs, the function creates a mirror image of the bitmap along the x-axis. If *nHeightSrc* and *nHeightDest* have different signs, the function creates a mirror image of the bitmap along the y-axis.

Not all devices support the **StretchBlt** function. For more information, see ***GetDeviceCaps***.

ICM: No color management is performed when a blit operation occurs.

Windows 98, Windows 2000: When used in a multimonitor system, both *hdcSrc* and *hdcDest* must refer to the same device or the function will fail.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BitBlt**, **GetDeviceCaps**, **MaskBlt**, **PlgBlt**, **SetStretchBltMode**

StretchDIBits

The **StretchDIBits** function copies the color data for a rectangle of pixels in a DIB to the specified destination rectangle. If the destination rectangle is larger than the source rectangle, this function stretches the rows and columns of color data to fit the destination rectangle. If the destination rectangle is smaller than the source rectangle, this function compresses the rows and columns by using the specified raster operation.

Windows 98 and Windows 2000: StretchDIBits has been extended to allow a JPEG or PNG image to be passed as the source image.

```
int StretchDIBits(
  HDC hdc,                      // handle to DC
  int XDest,                    // x-coord of destination upper-left
                                // corner
```

(continued)

(continued)

```
  int YDest,                    // y-coord of destination upper-left
                                // corner
  int nDestWidth,               // width of destination rectangle
  int nDestHeight,              // height of destination rectangle
  int XSrc,                     // x-coord of source upper-left corner
  int YSrc,                     // y-coord of source upper-left corner
  int nSrcWidth,                // width of source rectangle
  int nSrcHeight,               // height of source rectangle
  CONST VOID *lpBits,           // bitmap bits
  CONST BITMAPINFO *lpBitsInfo, // bitmap data
  UINT iUsage,                  // usage options
  DWORD dwRop                   // raster operation code
);
```

Parameters

hdc

[in] Handle to the destination device context.

XDest

[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

YDest

[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nDestWidth

[in] Specifies the width, in logical units, of the destination rectangle.

nDestHeight

[in] Specifies the height, in logical units, of the destination rectangle.

XSrc

[in] Specifies the x-coordinate, in pixels, of the source rectangle in the DIB.

YSrc

[in] Specifies the y-coordinate, in pixels, of the source rectangle in the DIB.

nSrcWidth

[in] Specifies the width, in pixels, of the source rectangle in the DIB.

nSrcHeight

[in] Specifies the height, in pixels, of the source rectangle in the DIB.

lpBits

[in] Pointer to the DIB bits, which are stored as an array of bytes. For more information, see the Remarks section.

lpBitsInfo

[in] Pointer to a **BITMAPINFO** structure that contains information about the DIB.

iUsage

[in] Specifies whether the **bmiColors** member of the **BITMAPINFO** structure was provided and, if so, whether **bmiColors** contains explicit red, green, blue (RGB) values or indexes. The *iUsage* parameter must be one of the following values:

Value	Meaning
DIB_PAL_COLORS	The array contains 16-bit indexes into the logical palette of the source device context.
DIB_RGB_COLORS	The color table contains literal RGB values.

For more information, see the Remarks section.

dwRop

[in] Specifies how the source pixels, the destination device context's current brush, and the destination pixels are to be combined to form the new image. For more information, see the following Remarks section.

Return Values

If the function succeeds, the return value is the number of scan lines copied.

If the function fails, the return value is GDI_ERROR.

Windows NT/2000: To get extended error information, call **GetLastError**.

Windows 98/Windows 2000: If the driver cannot support the JPEG or PNG file image passed to **StretchDIBits**, the function will fail and return GDI_ERROR. If failure does occur, the application must fall back on its own JPEG or PNG support to decompress the image into a bitmap, and then pass the bitmap to **StretchDIBits**.

Remarks

The origin of a bottom-up DIB is the bottom-left corner; the origin of a top-down DIB is the upper-left corner.

StretchDIBits creates a mirror image of a bitmap if the signs of the *nSrcWidth* and *nDestWidth* parameters, or if the *nSrcHeight* and *nDestHeight* parameters differ. If *nSrcWidth* and *nDestWidth* have different signs, the function creates a mirror image of the bitmap along the x-axis. If *nSrcHeight* and *nDestHeight* have different signs, the function creates a mirror image of the bitmap along the y-axis.

Windows 98/Windows 2000: This function allows a JPEG or PNG image to be passed as the source image. How each parameter is used remains the same, except as follows:

- If the **biCompression** member of **BITMAPINFOHEADER** is BI_JPEG or BI_PNG, *lpBits* points to a buffer containing a JPEG or PNG image, respectively. The *biSizeImage* member of **BITMAPINFOHEADER**specifies the size of the buffer. The *iUsage* parameter must be set to DIB_RGB_COLORS. The *dwRop parameter* must be set to SRCCOPY.

- If the **bV4Compression** member of **BITMAPV4HEADER** is BI_JPEG or BI_PNG, *lpBits* points to a buffer containing a JPEG or PNG image, respectively. The **BITMAPV4HEADER**'s **bV4SizeImage** member specifies the size of the buffer. The *iUsage* parameter must be set to DIB_RGB_COLORS. The *dwRop* parameter must be set to SRCCOPY.
- If the **bV5Compression** member of **BITMAPV5HEADER** is BI_JPEG or BI_PNG, *lpBits* points to a buffer containing a JPEG or PNG image, respectively. The **BITMAPV5HEADER**'s **bV5SizeImage** member specifies the size of the buffer. The *iUsage* parameter must be set to DIB_RGB_COLORS. The *dwRop* parameter must be set to SRCCOPY.
- To ensure proper metafile spooling while printing, applications must call the CHECKJPEGFORMAT or CHECKPNGFORMAT escape to verify that the printer recognizes the JPEG or PNG image, respectively, before calling **StretchDIBits**.

ICM: Color management is performed. If the specified **BITMAPINFO**'s **bmiHeader** does not contain **BITMAPV4HEADER** or **BITMAPV5HEADER**, the color profile of the current device context is used as the source color space profile. If it does not have a color profile, the sRGB space is used. If the specified **BITMAPINFO**'s **bmiHeader** contains **BITMAPV4HEADER** or **BITMAPV5HEADER**, the color space profile specified in the bitmap header is used as the source of color space profile.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Bitmaps Overview, Bitmap Functions, **BITMAPINFO**, **SetMapMode**, **SetStretchBltMode**

TransparentBlt

The **TransparentBlt** function performs a bit-block transfer of the color data corresponding to a rectangle of pixels from the specified source device context into a destination device context.

```
BOOL TransparentBlt(
  HDC hdcDest,          // handle to destination DC
  int nXOriginDest,     // x-coord of destination upper-left
                        // corner
  int nYOriginDest,     // y-coord of destination upper-left
```

```
                      // corner
  int nWidthDest,     // width of destination rectangle
  int hHeightDest,    // height of destination rectangle
  HDC hdcSrc,         // handle to source DC
  int nXOriginSrc,    // x-coord of source upper-left corner
  int nYOriginSrc,    // y-coord of source upper-left corner
  int nWidthSrc,      // width of source rectangle
  int nHeightSrc,     // height of source rectangle
  UINT crTransparent  // color to make transparent
);
```

Parameters

hdcDest

[in] Handle to the destination device context.

*nXOriginDes*t

[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nYOriginDest

[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the destination rectangle.

nWidthDest

[in] Specifies the width, in logical units, of the destination rectangle.

hHeightDest

[in] Handle to the height, in logical units, of the destination rectangle.

hdcSrc

[in] Handle to the source device context.

nXOriginSrc

[in] Specifies the x-coordinate, in logical units, of the source rectangle.

nYOriginSrc

[in] Specifies the y-coordinate, in logical units, of the source rectangle.

nWidthSrc

[in] Specifies the width, in logical units, of the source rectangle.

nHeightSrc

[in] Specifies the height, in logical units, of the source rectangle.

crTransparent

[in] The RGB color in the source bitmap to treat as transparent.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **TransparentBlt** function supports all formats of source bitmaps. However, for 32 bpp bitmaps, it just copies the alpha value over. Use **AlphaBlend** to specify 32 bits-per-pixel bitmaps with transparency.

If the source and destination rectangles are not the same size, the source bitmap is stretched to match the destination rectangle. When the **SetStretchBltMode** function is used, the *iStretchMode* modes of BLACKONWHITE and WHITEONBLACK are converted to COLORONCOLOR for the **TransparentBlt** function.

The destination device context specifies the transformation type for the destination coordinates. The source device context specifies the transformation type for the source coordinates.

TransparentBlt does not mirror a bitmap if either the width or height, of either the source or destination, is negative.

Windows 98/Windows 2000: When used in a multimonitor system, both *hdcSrc* and *hdcDest* must refer to the same device or the function will fail.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

+ See Also

Bitmaps Overview, Bitmap Functions, **AlphaBlend**, **SetStretchBltMode**

Bitmap Structures

BITMAP

The **BITMAP** structure defines the type, width, height, color format, and bit values of a bitmap.

```
typedef struct tagBITMAP {
  LONG   bmType;
  LONG   bmWidth;
  LONG   bmHeight;
  LONG   bmWidthBytes;
  WORD   bmPlanes;
```

```
    WORD    bmBitsPixel;
    LPVOID  bmBits;
} BITMAP, *PBITMAP;
```

Members

bmType

Specifies the bitmap type. This member must be zero.

bmWidth

Specifies the width, in pixels, of the bitmap. The width must be greater than zero.

bmHeight

Specifies the height, in pixels, of the bitmap. The height must be greater than zero.

bmWidthBytes

Specifies the number of bytes in each scan line. This value must be divisible by two, because the system assumes that the bit values of a bitmap form an array that is word aligned.

bmPlanes

Specifies the count of color planes.

bmBitsPixel

Specifies the number of bits required to indicate the color of a pixel.

bmBits

Pointer to the location of the bit values for the bitmap. The **bmBits** member must be a long pointer to an array of character (1-byte) values.

Remarks

The bitmap formats currently used are monochrome and color. The monochrome bitmap uses a one-bit, one-plane format. Each scan is a multiple of 32 bits.

Scans are organized as follows for a monochrome bitmap of height *n*:

```
    Scan 0
    Scan 1
    .
    .
    .
    Scan n-2
    Scan n-1
```

The pixels on a monochrome device are either black or white. If the corresponding bit in the bitmap is 1, the pixel is set to the foreground color; if the corresponding bit in the bitmap is zero, the pixel is set to the background color.

All devices that have the RC_BITBLT device capability support bitmaps. For more information, see **GetDeviceCaps**.

Each device has a unique color format. To transfer a bitmap from one device to another, use the **GetDIBits** and **SetDIBits** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **CreateBitmapIndirect**, **GetObject**

BITMAPCOREHEADER

The **BITMAPCOREHEADER** structure contains information about the dimensions and color format of a DIB.

```
typedef struct tagBITMAPCOREHEADER {
  DWORD   bcSize;
  WORD    bcWidth;
  WORD    bcHeight;
  WORD    bcPlanes;
  WORD    bcBitCount;
} BITMAPCOREHEADER, *PBITMAPCOREHEADER;
```

Members

bcSize
Specifies the number of bytes required by the structure.

bcWidth
Specifies the width of the bitmap, in pixels.

bcHeight
Specifies the height of the bitmap, in pixels.

bcPlanes
Specifies the number of planes for the target device. This value must be 1.

bcBitCount
Specifies the number of bits-per-pixel. This value must be 1, 4, 8, or 24.

Remarks

The **BITMAPCOREINFO** structure combines the **BITMAPCOREHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a DIB. For more information about specifying a DIB, see **BITMAPCOREINFO**.

An application should use the information stored in the **bcSize** member to locate the color table in a **BITMAPCOREINFO** structure, using a method such as the following:

```
pColor = ((LPBYTE) pBitmapCoreInfo +
        (WORD) (pBitmapCoreInfo -> bcSize))
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **BITMAPCOREINFO**

BITMAPCOREINFO

The **BITMAPCOREINFO** structure defines the dimensions and color information for a DIB.

```
typedef struct _BITMAPCOREINFO {
  BITMAPCOREHEADER  bmciHeader;
  RGBTRIPLE         bmciColors[1];
} BITMAPCOREINFO, *PBITMAPCOREINFO;
```

Members

bmciHeader

Specifies a **BITMAPCOREHEADER** structure that contains information about the dimensions and color format of a DIB.

bmciColors

Specifies an array of **RGBTRIPLE** structures that define the colors in the bitmap.

Remarks

A DIB consists of two parts: a **BITMAPCOREINFO** structure describing the dimensions and colors of the bitmap, and an array of bytes defining the pixels of the bitmap. The bits in the array are packed together, but each scan line must be padded with zeros to end on a **LONG** boundary. The origin of the bitmap is the lower-left corner.

The **bcBitCount** member of the **BITMAPCOREHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member can be one of the following values:

Value	Meaning
1	The bitmap is monochrome, and the **bmciColors** member contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmciColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmciColors** member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmciColors** member contains up to 256 entries. In this case, each byte in the array represents a single pixel.
24	The bitmap has a maximum of 2^{24} colors, and the **bmciColors** member is NULL. Each three-byte triplet in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel.

The colors in the **bmciColors** table should appear in order of importance.

Alternatively, for functions that use DIBs, the **bmciColors** member can be an array of 16-bit unsigned integers that specify indexes into the currently realized logical palette, instead of explicit RGB values. In this case, an application using the bitmap must call the DIB functions (**CreateDIBitmap**, **CreateDIBPatternBrush**, and **CreateDIBSection**) with the *iUsage* parameter set to DIB_PAL_COLORS.

Note The **bmciColors** member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application. Unless the application has exclusive use and control of the bitmap, the bitmap color table should contain explicit RGB values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **BITMAPCOREHEADER**, **CreateDIBitmap**, **CreateDIBPatternBrush**, **CreateDIBSection**, **RGBTRIPLE**

BITMAPFILEHEADER

The **BITMAPFILEHEADER** structure contains information about the type, size, and layout of a file that contains a DIB.

```
typedef struct tagBITMAPFILEHEADER {
  WORD    bfType;
  DWORD   bfSize;
  WORD    bfReserved1;
  WORD    bfReserved2;
  DWORD   bfOffBits;
} BITMAPFILEHEADER, *PBITMAPFILEHEADER;
```

Members

bfType

Specifies the file type; must be BM.

bfSize

Specifies the size, in bytes, of the bitmap file.

bfReserved1

Reserved; must be zero.

bfReserved2

Reserved; must be zero.

bfOffBits

Specifies the offset, in bytes, from the **BITMAPFILEHEADER** structure to the bitmap bits.

Remarks

A **BITMAPINFO** or **BITMAPCOREINFO** structure immediately follows the **BITMAPFILEHEADER** structure in the DIB file.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures, **BITMAPCOREINFO**, **BITMAPINFO**

BITMAPINFO

The **BITMAPINFO** structure defines the dimensions and color information for a Win32 DIB.

```
typedef struct tagBITMAPINFO {
  BITMAPINFOHEADER bmiHeader;
  RGBQUAD          bmiColors[1];
} BITMAPINFO, *PBITMAPINFO;
```

Members

bmiHeader

Specifies a bitmap information header structure that contains information about the dimensions of color format. The bitmap information header structure is version-related:

Windows NT 3.51 and earlier: Use the **BITMAPINFOHEADER** structure.

Windows 95 and Windows NT 4.0: Use the **BITMAPV4HEADER** structure.

Windows 98 and Windows 2000: Use the **BITMAPV5HEADER** structure.

bmiColors

The **bmiColors** member contains one of the following:

- An array of **RGBQUAD**. The elements of the array that make up the color table.
- An array of 16-bit unsigned integers that specifies indexes into the currently realized logical palette. This use of **bmiColors** is allowed for functions that use DIBs. When **bmiColors** elements contain indexes to a realized logical palette, they must also call the following bitmap functions:

  CreateDIBitmap

  CreateDIBPatternBrush

  CreateDIBSection

  The i*Usage* parameter of **CreateDIBSection** must be set to DIB_PAL_COLORS.

Platform differences are listed in the following:

Windows NT 3.51 and earlier: Use of the number of entries in the array depends on the values of the **biBitCount** and **biClrUsed** members of the **BITMAPINFOHEADER** structure.

Windows 95 and Windows NT 4.0: Use of the number of entries in the array depends on the values of the **bV4BitCount** and **bV4ClrUsed** members of the **BITMAPV4HEADER** structure.

Windows 98 and Windows 2000: Use of the number of entries in the array depends on the values of the **bV5BitCount** and **bV5ClrUsed** members of the **BITMAPV5HEADER** structure.

The colors in the **bmiColors** table appear in order of importance. For more information, see the Remarks section.

Remarks

A DIB consists of two distinct parts: a **BITMAPINFO** structure describing the dimensions and colors of the bitmap, and an array of bytes defining the pixels of the bitmap. The bits in the array are packed together, but each scan line must be padded with zeros to end on a **LONG** data-type boundary. If the height of the bitmap is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If the height is negative, the bitmap is a top-down DIB and its origin is the upper left corner.

A bitmap is packed when the bitmap array immediately follows the **BITMAPINFO** header. Packed bitmaps are referenced by a single pointer. For packed bitmaps, the **ClrUsed** member must be set to an even number when using the DIB_PAL_COLORS mode so that the DIB bitmap array starts on a **DWORD** boundary.

Note The **bmiColors** member should not contain palette indexes if the bitmap is to be stored in a file or transferred to another application.

Unless the application has exclusive use and control of the bitmap, the bitmap color table should contain explicit RGB values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **CreateDIBitmap**, **CreateDIBPatternBrush**, **CreateDIBSection**, **RGBQUAD**

BITMAPINFOHEADER

The **BITMAPINFOHEADER** structure contains information about the dimensions and color format of a DIB.

Applications developed for Windows NT 4.0 and Windows 95 may use the **BITMAPV4HEADER** structure. Applications developed for Windows 2000 and Windows 98 may use the **BITMAPV5HEADER** structure for increased functionality.

```
typedef struct tagBITMAPINFOHEADER{
  DWORD  biSize;
  LONG   biWidth;
  LONG   biHeight;
  WORD   biPlanes;
  WORD   biBitCount
  DWORD  biCompression;
```

(continued)

(continued)

```
  DWORD  biSizeImage;
  LONG   biXPelsPerMeter;
  LONG   biYPelsPerMeter;
  DWORD  biClrUsed;
  DWORD  biClrImportant;
} BITMAPINFOHEADER, *PBITMAPINFOHEADER;
```

Members

biSize

Specifies the number of bytes required by the structure.

biWidth

Specifies the width of the bitmap, in pixels.

Windows 98, Windows 2000: If **biCompression** is BI_JPEG or BI_PNG, the **biWidth** member specifies the width of the decompressed JPEG or PNG image file, respectively.

biHeight

Specifies the height of the bitmap, in pixels. If **biHeight** is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If **biHeight** is negative, the bitmap is a top-down DIB and its origin is the upper-left corner.

If **biHeight** is negative, indicating a top-down DIB, **biCompression** must be either BI_RGB or BI_BITFIELDS. Top-down DIBs cannot be compressed.

Windows 98, Windows 2000: If **biCompression** is BI_JPEG or BI_PNG, the **biHeight** member specifies the height of the decompressed JPEG or PNG image file, respectively.

biPlanes

Specifies the number of planes for the target device. This value must be set to 1.

biBitCount

Specifies the number of bits-per-pixel. The **biBitCount** member of the **BITMAPINFOHEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member must be one of the following values:

Value	Meaning
0	**Windows 98, Windows 2000:** The number of bits per pixel is specified or implied by the JPEG or PNG format.
1	The bitmap is monochrome, and the **bmiColors** member contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** table; if the bit is set, the pixel has the color of the second entry in the table.

Value	Meaning
4	The bitmap has a maximum of 16 colors, and the **bmiColors** member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** member contains up to 256 entries. In this case, each byte in the array represents a single pixel.
16	The bitmap has a maximum of 2^16 colors. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_RGB, the **bmiColors** member is NULL. Each **WORD** in the bitmap array represents a single pixel. The relative intensities of red, green, and blue are represented with five bits for each color component. The value for blue is in the least significant five bits, followed by five bits each for green and red. The most significant bit is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPINFOHEADER**. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. Each **WORD** in the bitmap array represents a single pixel. **Windows NT/Windows 2000:** When the **biCompression** member is BI_BITFIELDS, bits set in each **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not have to be used. **Windows 95/98:** When the **biCompression** member is BI_BITFIELDS, the system supports only the following 16bpp color masks: A 5-5-5 16-bit image, where the blue mask is 0x001F, the green mask is 0x03E0, and the red mask is 0x7C00; and a 5-6-5 16-bit image, where the blue mask is 0x001F, the green mask is 0x07E0, and the red mask is 0xF800.
24	The bitmap has a maximum of 2^24 colors, and the **bmiColors** member is NULL. Each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPINFOHEADER**.

(continued)

(continued)

Value	Meaning
32	The bitmap has a maximum of 2^32 colors. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_RGB, the **bmiColors** member is NULL. Each **DWORD** in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. The high byte in each **DWORD** is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPINFOHEADER**. If the **biCompression** member of the **BITMAPINFOHEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. Each **DWORD** in the bitmap array represents a single pixel. **Windows NT/2000:** When the **biCompression** member is BI_BITFIELDS, bits set in each **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not need to be used. **Windows 95/98:** When the **biCompression** member is BI_BITFIELDS, the system supports only the following 32-bpp color mask: The blue mask is 0x000000FF, the green mask is 0x0000FF00, and the red mask is 0x00FF0000.

biCompression

Specifies the type of compression for a compressed bottom-up bitmap (top-down DIBs cannot be compressed). This member can be one of the following values:

Value	Description
BI_RGB	An uncompressed format.
BI_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bpp. The compression format is a 2-byte format consisting of a count byte followed by a byte containing a color index. For more information, see *Bitmap Compression.*
BI_RLE4	An RLE format for bitmaps with 4 bpp. The compression format is a 2-byte format consisting of a count byte followed by two word-length color indexes. For more information, see *Bitmap Compression.*
BI_BITFIELDS	Specifies that the bitmap is not compressed and that the color table consists of three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. This is valid when used with 16-bpp and 32-bpp bitmaps.
BI_JPEG	**Windows 98, Windows 2000:** Indicates that the image is a JPEG image.
BI_PNG	**Windows 98, Windows 2000:** Indicates that the image is a PNG image.

biSizeImage

Specifies the size, in bytes, of the image. This may be set to zero for BI_RGB bitmaps.

Windows 98, Windows 2000: If **biCompression** is BI_JPEG or BI_PNG, **biSizeImage** indicates the size of the JPEG or PNG image buffer, respectively.

biXPelsPerMeter

Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

biYPelsPerMeter

Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.

biClrUsed

Specifies the number of color indexes in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **biBitCount** member for the compression mode specified by **biCompression**.

If **biClrUsed** is nonzero and the **biBitCount** member is less than 16, the **biClrUsed** member specifies the actual number of colors the graphics engine or device driver accesses. If **biBitCount** is 16 or greater, the **biClrUsed** member specifies the size of the color table used to optimize performance of the system color palettes. If **biBitCount** equals 16 or 32, the optimal color palette starts immediately following the three **DWORD** masks.

If the bitmap is a packed bitmap (a bitmap in which the bitmap array immediately follows the **BITMAPINFO** header and is referenced by a single pointer), the **biClrUsed** member must be either zero or the actual size of the color table.

biClrImportant

Specifies the number of color indexes that are required for displaying the bitmap. If this value is zero, all colors are required.

Remarks

The **BITMAPINFO** structure combines the **BITMAPINFOHEADER** structure and a color table to provide a complete definition of the dimensions and colors of a DIB. For more information about DIBs, see *Device-Independent Bitmaps* and ***BITMAPINFO***.

An application should use the information stored in the **biSize** member to locate the color table in a **BITMAPINFO** structure, as follows:

```
pColor = ((LPSTR)pBitmapInfo +
    (WORD)(pBitmapInfo->bmiHeader.biSize));
```

Windows 98, Windows 2000: The **BITMAPINFOHEADER** structure is extended to allow a JPEG or PNG image to be passed as the source image to **StretchDIBits**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures

BITMAPV4HEADER

The **BITMAPV4HEADER** structure is the Windows 95 and Windows NT 4.0 bitmap information header file. Applications written for earlier versions of Windows NT should continue to use **BITMAPINFOHEADER**. Applications written for Windows 2000 and Windows 98 can use **BITMAPV5HEADER**.

```
typedef struct {
  DWORD        bV4Size;
  LONG         bV4Width;
  LONG         bV4Height;
  WORD         bV4Planes;
  WORD         bV4BitCount;
  DWORD        bV4Compression;
  DWORD        bV4SizeImage;
  LONG         bV4XPelsPerMeter;
  LONG         bV4YPelsPerMeter;
  DWORD        bV4ClrUsed;
  DWORD        bV4ClrImportant;
  DWORD        bV4RedMask;
  DWORD        bV4GreenMask;
  DWORD        bV4BlueMask;
  DWORD        bV4AlphaMask;
  DWORD        bV4CSType;
  CIEXYZTRIPLE bV4EndPoints;
  DWORD        bV4GammaRed;
  DWORD        bV4GammaGreen;
  DWORD        bV4GammaBlue;
} BITMAPV4HEADER, *PBITMAPV4HEADER;
```

Members

bV4Size

Specifies the number of bytes required by the structure. Applications should use this member to determine which bitmap information header structure is being used.

bV4Width

Specifies the width of the bitmap, in pixels.

Windows 98, Windows 2000: If **bV4Compression** is BI_JPEG or BI_PNG, **bV4Width** specifies the width of the JPEG or PNG image in pixels.

bV4Height

Specifies the height of the bitmap, in pixels. If **bV4Height** is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If **bV4Height** is negative, the bitmap is a top-down DIB and its origin is the upper-left corner.

If **bV4Height** is negative, indicating a top-down DIB, **bV4Compression** must be either BI_RGB or BI_BITFIELDS. Top-down DIBs cannot be compressed.

Windows 98, Windows 2000: If **bV4Compression** is BI_JPEG or BI_PNG, **bV4Height** specifies the height of the JPEG or PNG image in pixels.

bV4Planes

Specifies the number of planes for the target device. This value must be set to 1.

bV4BitCount

Specifies the number of bits per pixel. The **bV4BitCount** member of the **BITMAPV4HEADER** structure determines the number of bits that define each pixel and the maximum number of colors in the bitmap. This member must be one of the following values:

Value	Meaning
0	**Windows 98, Windows 2000:** The number of bits-per-pixel is specified or is implied by the JPEG or PNG file format.
1	The bitmap is monochrome, and the **bmiColors** member contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** table; if the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmiColors** member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** member contains up to 256 entries. In this case, each byte in the array represents a single pixel.

(continued)

(continued)

Value	Meaning
16	The bitmap has a maximum of 2^16 colors. If the **bV4Compression** member of the **BITMAPINFOHEADER** is BI_RGB, the **bmiColors** member is NULL. Each **WORD** in the bitmap array represents a single pixel. The relative intensities of red, green, and blue are represented with five bits for each color component. The value for blue is in the least significant five bits, followed by five bits each for green and red, respectively. The most significant bit is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **bV4ClrUsed** member of the **BITMAPV4HEADER**. If the **bV4Compression** member of the **BITMAPV4HEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components of each pixel. Each **WORD** in the bitmap array represents a single pixel.
24	The bitmap has a maximum of 2^24 colors, and the **bmiColors** member is NULL. Each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red for a pixel. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **bV4ClrUsed** member of the **BITMAPV4HEADER**.
32	The bitmap has a maximum of 2^32 colors. If the **biCompression** member of the **BITMAPV4HEADER** is BI_RGB, the **bmiColors** member is NULL. Each **DWORD** in the bitmap array represents the relative intensities of blue, green, and red for a pixel. The high byte in each **DWORD** is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPV4HEADER**. If the **bV4Compression** member of the **BITMAPV4HEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components of each pixel. Each **DWORD** in the bitmap array represents a single pixel.

bV4Compression

Specifies the type of compression for a compressed bottom-up bitmap (top-down DIBs cannot be compressed). This member can be one of the following values:

Value	Description
BI_RGB	An uncompressed format.
BI_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bpp. The compression format is a 2-byte format consisting of a count byte followed by a byte containing a color index. For more information, see *Bitmap Compression*.
BI_RLE4	An RLE format for bitmaps with 4 bpp. The compression format is a 2-byte format consisting of a count byte followed by two word-length color indexes. For more information, see *Bitmap Compression*.
BI_BITFIELDS	Specifies that the bitmap is not compressed. The members **bV4RedMask**, **bV4GreenMask**, and **bV4BlueMask** specify the red, green, and blue components for each pixel. This is valid when used with 16-bpp and 32-bpp bitmaps.
BI_JPEG	**Windows 98, Windows 2000:** Specifies that the image is compressed using the JPEG file interchange format. JPEG compression trades off compression against loss; it can achieve a compression ratio of 20:1 with little noticeable loss.
BI_PNG	**Windows 98, Windows 2000:** Specifies that the image is compressed using the PNG file interchange format.

bV4SizeImage

Specifies the size, in bytes, of the image. This may be set to zero for BI_RGB bitmaps.

Windows 98, Windows 2000: If **biCompression** is BI_JPEG or BI_PNG, **bV4SizeImage** is the size of the JPEG or PNG image buffer.

bV4XPelsPerMeter

Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

bV4YPelsPerMeter

Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.

bV4ClrUsed

Specifies the number of color indexes in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **bV4BitCount** member for the compression mode specified by **bV4Compression**.

If **bV4ClrUsed** is nonzero and the **bV4BitCount** member is less than 16, the **bV4ClrUsed** member specifies the actual number of colors the graphics engine or device driver accesses. If **bV4BitCount** is 16 or greater, the **bV4ClrUsed** member specifies the size of the color table used to optimize performance of the system color palettes. If **bV4BitCount** equals 16 or 32, the optimal color palette starts immediately following the **BITMAPV4 HEADER**.

When the bitmap array immediately follows the **BITMAPINFO** header, it is a packed bitmap. Packed bitmaps are referenced by a single pointer. Packed bitmaps require that the **bV4ClrUsed** member be either zero or the actual size of the color table.

bV4ClrImportant
Specifies the number of color indexes that are required for displaying the bitmap. If this value is zero, all colors are important.

bV4RedMask
Color mask that specifies the red component of each pixel, valid only if **bV4Compression** is set to BI_BITFIELDS.

bV4GreenMask
Color mask that specifies the green component of each pixel, valid only if **bV4Compression** is set to BI_BITFIELDS.

bV4BlueMask
Color mask that specifies the blue component of each pixel, valid only if **bV4Compression** is set to BI_BITFIELDS.

bV4AlphaMask
Color mask that specifies the alpha component of each pixel.

bV4CSType
Specifies the color space of the DIB. The following table lists the value for **bV4CSType**:

Value	Meaning
LCS_CALIBRATED_RGB	This value indicates that endpoints and gamma values are given in the appropriate fields.

See the **LOGCOLORSPACE** structure for information that defines a logical color space.

bV4EndPoints
A **CIEXYZTRIPLE** structure that specifies the x, y, and z coordinates of the three colors that correspond to the red, green, and blue endpoints for the logical color space associated with the bitmap. This member is ignored unless the **bV4CSType** member specifies LCS_CALIBRATED_RGB.

Note A *color space* is a model for representing color numerically in terms of three or more coordinates. For example, the RGB color space represents colors in terms of the red, green, and blue coordinates.

bV4GammaRed
Toned response curve for red. This member is ignored unless color values are calibrated RGB values and **bV4CSType** is set to LCS_CALIBRATED_RGB. Specified in 16^16 format.

bV4GammaGreen
Toned response curve for green. Used if **bV4CSType** is set to LCS_CALIBRATED_RGB. Specified as 16^16 format.

bV4GammaBlue
Toned response curve for blue. Used if **bV4CSType** is set to LCS_CALIBRATED_RGB. Specified as 16^16 format.

Remarks

The **BITMAPINFO** structure combines the **BITMAPV4HEADER** structure and a color table to provide a complete definition of the dimensions and colors of a DIB. For more information about DIBs, see *Device-Independent Bitmaps* and ***BITMAPINFO***.

An application should use the information stored in the **bV4Size** member to locate the color table in a **BITMAPINFO** structure, as follows:

```
pColor = ((LPSTR)pBitmapInfo +
    (WORD)(pBitmapInfo->bmiHeader.biSize));
```

Windows 98, Windows 2000: The **BITMAPV4HEADER** structure is extended to allow a JPEG or PNG image to be passed as the source image to **StretchDIBits**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures

BITMAPV5HEADER

The **BITMAPV5HEADER** structure is the Windows 2000 and Windows 98 bitmap information header file. The Independent Color Management interface (ICM) 2.0 allows International Color Consortium (ICC) color profiles to be linked or embedded in DIBs (DIBs). See *Using Structures in ICM 2.0* for more information.

Applications written for Windows NT 4.0 and Windows 95 can use the **BITMAPV4HEADER** structure. Applications written for earlier versions of Windows NT should continue to use the **BITMAPINFOHEADER** structure.

The **BITMAPV5HEADER** is an extended version of **BITMAPINFOHEADER** and allows a JPEG or PNG image to be passed as the source image to **StretchDIBits**.

```
typedef struct {
  DWORD        bV5Size;
  LONG         bV5Width;
  LONG         bV5Height;
```

(continued)

(continued)

```
WORD          bV5Planes;
  WORD          bV5BitCount;
  DWORD         bV5Compression;
  DWORD         bV5SizeImage;
  LONG          bV5XPelsPerMeter;
  LONG          bV5YPelsPerMeter;
  DWORD         bV5ClrUsed;
  DWORD         bV5ClrImportant;
  DWORD         bV5RedMask;
  DWORD         bV5GreenMask;
  DWORD         bV5BlueMask;
  DWORD         bV5AlphaMask;
  DWORD         bV5CSType;
  CIEXYZTRIPLE bV5EndPoints;
  DWORD         bV5GammaRed;
  DWORD         bV5GammaGreen;
  DWORD         bV5GammaBlue;
  DWORD         bV5Intent;
  DWORD         bV5ProfileData;
  DWORD         bV5ProfileSize;
  DWORD         bV5Reserved;
} BITMAPV5HEADER, *PBITMAPV5HEADER;
```

Members

bV5Size

Specifies the number of bytes required by the structure. Applications should use this member to determine which bitmap information header structure is being used.

bV5Width

Specifies the width of the bitmap, in pixels.

If **bV5Compression** is BI_JPEG or BI_PNG, the **bV5Width** member specifies the width of the decompressed JPEG or PNG image in pixels.

bV5Height

Specifies the height of the bitmap, in pixels. If the value of **bV5Height** is positive, the bitmap is a bottom-up DIB and its origin is the lower-left corner. If **bV5Height** value is negative, the bitmap is a top-down DIB and its origin is the upper-left corner.

If **bV5Height** is negative, indicating a top-down DIB, **bV5Compression** must be either BI_RGB or BI_BITFIELDS. Top-down DIBs cannot be compressed.

If **bV5Compression** is BI_JPEG or BI_PNG, the **bV5Height** member specifies the height of the decompressed JPEG or PNG image in pixels.

bV5Planes

Specifies the number of planes for the target device. This value must be set to 1.

bV5BitCount

Specifies the number of bits that define each pixel and the maximum number of colors in the bitmap.

This member can be one of the following values:

Value	Meaning
0	The number of bits per pixel is specified or is implied by the JPEG or PNG file format.
1	The bitmap is monochrome, and the **bmiColors** member contains two entries. Each bit in the bitmap array represents a pixel. If the bit is clear, the pixel is displayed with the color of the first entry in the **bmiColors** color table. If the bit is set, the pixel has the color of the second entry in the table.
4	The bitmap has a maximum of 16 colors, and the **bmiColors** member contains up to 16 entries. Each pixel in the bitmap is represented by a 4-bit index into the color table. For example, if the first byte in the bitmap is 0x1F, the byte represents two pixels. The first pixel contains the color in the second table entry, and the second pixel contains the color in the sixteenth table entry.
8	The bitmap has a maximum of 256 colors, and the **bmiColors** member contains up to 256 entries. In this case, each byte in the array represents a single pixel.
16	The bitmap has a maximum of 2^16 colors. If the **biCompression** member of the **BITMAPV5HEADER** structure is BI_RGB, the **bmiColors** member is NULL. Each **WORD** in the bitmap array represents a single pixel. The relative intensities of red, green, and blue are represented with five bits for each color component. The value for blue is in the least significant five bits, followed by five bits each for green and red. The most significant bit is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPV5HEADER**. If the **biCompression** member of the **BITMAPV5HEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components, respectively, of each pixel. Each **WORD** in the bitmap array represents a single pixel. When the **biCompression** member is BI_BITFIELDS, bits set in each **DWORD** mask must be contiguous and should not overlap the bits of another mask. All the bits in the pixel do not need to be used.
24	The bitmap has a maximum of 2^24 colors, and the **bmiColors** member is NULL. Each 3-byte triplet in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPV5HEADER** structure.

(continued)

(continued)

Value	Meaning
32	The bitmap has a maximum of 2^32 colors. If the **biCompression** member of the **BITMAPV5HEADER** is BI_RGB, the **bmiColors** member is NULL. Each **DWORD** in the bitmap array represents the relative intensities of blue, green, and red, respectively, for a pixel. The high byte in each **DWORD** is not used. The **bmiColors** color table is used for optimizing colors used on palette-based devices, and must contain the number of entries specified by the **biClrUsed** member of the **BITMAPV5HEADER**. If the **biCompression** member of the **BITMAPV5HEADER** is BI_BITFIELDS, the **bmiColors** member contains three **DWORD** color masks that specify the red, green, and blue components of each pixel. Each **DWORD** in the bitmap array represents a single pixel.

bV5Compression

Specifies that the bitmap is not compressed. The **bV5RedMask**, **bV5GreenMask**, and **bV5BlueMask** members specify the red, green, and blue components of each pixel. This is valid when used with 16-bpp and 32-bpp bitmaps. This member can be one of the following values:

Value	Meaning
BI_RGB	An uncompressed format.
BI_RLE8	A run-length encoded (RLE) format for bitmaps with 8 bpp. The compression format is a two-byte format consisting of a count byte followed by a byte containing a color index. If **bV5Compression** is BI_RGB and the **bV5BitCount** member is 16, 24, or 32, the bitmap array specifies the actual intensities of blue, green, and red rather than using color table indexes. For more information, see *Bitmap Compression*.
BI_RLE4	An RLE format for bitmaps with 4 bpp. The compression format is a two-byte format consisting of a count byte followed by two word-length color indexes. For more information, see *Bitmap Compression*.
BI_BITFIELDS	Specifies that the bitmap is not compressed and that the color table consists of three **DWORD** color masks that specify the red, green, and blue components of each pixel. Valid when used with 16-bpp and 32-bpp bitmaps.
BI_JPEG	Specifies that the image is compressed using the JPEG file Interchange Format. JPEG compression trades off compression against loss; it can achieve a compression ratio of 20:1 with little noticeable loss.
BI_PNG	Specifies that the image is compressed using the PNG file Interchange Format.

bV5SizeImage

Specifies the size, in bytes, of the image. This may be set to zero for BI_RGB bitmaps.

If **bV5Compression** is BI_JPEG or BI_PNG, **bVSizeImage** is the size of the JPEG or PNG image buffer.

bV5XPelsPerMeter

Specifies the horizontal resolution, in pixels per meter, of the target device for the bitmap. An application can use this value to select a bitmap from a resource group that best matches the characteristics of the current device.

bV5YPelsPerMeter

Specifies the vertical resolution, in pixels per meter, of the target device for the bitmap.

bV5ClrUsed

Specifies the number of color indexes in the color table that are actually used by the bitmap. If this value is zero, the bitmap uses the maximum number of colors corresponding to the value of the **bV5BitCount** member for the compression mode specified by **bV5Compression**.

If **bV5ClrUsed** is nonzero and **bV5iBitCount** is less than 16, the **bV5ClrUsed** member specifies the actual number of colors the graphics engine or device driver accesses. If **bV5BitCount** is 16 or greater, the **bV5ClrUsed** member specifies the size of the color table used to optimize performance of the system color palettes. If **bV5BitCount** equals 16 or 32, the optimal color palette starts immediately following the BITMAPV5HEADER. If **BV5ClrUsed** is nonzero, the color table is used on palettized devices, and **bV5ClrUsed** specifies the number of entries.

When the bitmap array immediately follows the **BITMAPINFO** header, it is a packed bitmap. Packed bitmaps are referenced by a single pointer. Packed bitmaps require that the **bV5ClrUsed** member must be either zero or the actual size of the color table.

bV5ClrImportant

Specifies the number of color indexes that are required for displaying the bitmap. If this value is zero, all colors are required.

bV5RedMask

Color mask that specifies the red component of each pixel, valid only if **bV5Compression** is set to BI_BITFIELDS.

bV5GreenMask

Color mask that specifies the green component of each pixel, valid only if **bV5Compression** is set to BI_BITFIELDS.

bV5BlueMask

Color mask that specifies the blue component of each pixel, valid only if **bV5Compression** is set to BI_BITFIELDS.

bV5AlphaMask

Color mask that specifies the alpha component of each pixel.

bV5CSType

Specifies the color space of the DIB.

The following table specifies the values for **bV5CSType**:

Value	Meaning
LCS_CALIBRATED_RGB	This value implies that endpoints and gamma values are given in the appropriate fields.
LCS_sRGB	Specifies that the bitmap is in sRGB color space.
LCS_WINDOWS_COLOR_SPACE	This value indicates that the bitmap is in the system default color space, sRGB.
PROFILE_LINKED	This value indicates that **bV5ProfileDat**a points to the file name of the profile to use (gamma and endpoints values are ignored).
PROFILE_EMBEDDED	This value indicates that **bV5ProfileData** points to a memory buffer that contains the profile to be used (gamma and endpoints values are ignored).

See the **LOGCOLORSPACE** structure for information that defines a logical color space.

bV5EndPoints

A **CIEXYZTRIPLE** structure that specifies the x-, y-, and z-coordinates of the three colors that correspond to the red, green, and blue endpoints for the logical color space associated with the bitmap. This member is ignored unless the **bV5CSType** member specifies LCS_CALIBRATED_RGB.

bV5GammaRed

Toned response curve for red. Used if **bV5CSType** is set to LCS_CALIBRATED_RGB. Specified in 16^16 format.

bV5GammaGreen

Toned response curve for green. Used if **bV5CSType** is set to LCS_CALIBRATED_RGB. Specified in 16^16 format.

bV5GammaBlue

Toned response curve for blue. Used if **bV5CSType** is set to LCS_CALIBRATED_RGB. Specified in 16^16 format.

bV5ProfileSize

Size, in bytes, of embedded profile data.

bV5Intent

Rendering intent for bitmap. This can be one of the following values:

Value	Intent	ICC name	Meaning
LCS_GM_ABS_COLORIMETRIC	Match	Absolute Colorimetric	Maintains the white point. Matches the colors to their nearest color in the destination gamut.
LCS_GM_BUSINESS	Graphic	Saturation	Maintains saturation. Used for business charts and other situations in which undithered colors are required.
LCS_GM_GRAPHICS	Proof	Relative Colorimetric	Maintains colorimetric match. Used for graphic designs and named colors.
LCS_GM_IMAGES	Picture	Perceptual	Maintains contrast. Used for photographs and natural images.

bV5ProfileData

The offset, in bytes, from the beginning of the **BITMAPV5HEADER** structure to the start of the profile data. If the profile is embedded, profile data is the actual profile, and it is linked. (The profile data is the null-terminated file name of the profile.) This cannot be a Unicode string. It must be composed exclusively of characters from the Windows character set (code page 1252). These profile members are ignored unless the **bV5CSType** member specifies PROFILE_LINKED or PROFILE_EMBEDDED.

bV5Reserved

This member has been reserved for future use. Its value should be set to zero.

Remarks

The **BITMAPINFO** structure combines the **BITMAPV5HEADER** structure and a color table to provide a complete definition of the dimensions and colors of a DIB. For more information about DIBs, see *Device-Independent Bitmaps* and ***BITMAPINFO***.

An application should use the information stored in the **bV5Size** member to locate the color table in a **BITMAPINFO** structure, as follows:

```
pColor = ((LPSTR)pBitmapInfo +
    (WORD)(pBitmapInfo->bmiHeader.biSize));
```

If **bV5Height** is negative, indicating a top-down DIB, **bV5Compression** must be either BI_RGB or BI_BITFIELDS. Top-down DIBs cannot be compressed.

When a DIB is loaded into memory, the profile data (if present) should follow the color table, and the **bV5ProfileData** should provide the offset of the profile data from the beginning of the **BITMAPV5HEADER** structure. The value stored in **bV5ProfileDate** will be different from the value returned by the **sizeof** operator given the **BITMAPV5HEADER** argument, because **bV5ProfileData** is the offset in bytes from thebeginning of the **BITMAPV5HEADER** structure to the start of the profile data. (Bitmap bits do not follow the color table in memory). Applications should modify the **bV5ProfileData** member after loading the DIB into memory.

For packed DIBs, the profile data should follow the bitmap bits similar to the file format. The **bV5ProfileData** member should still give the offset of the profile data from the beginning of the **BITMAPV5HEADER**.

Applications should access the profile data only when **bV5Size** equals the size of the **BITMAPB5HEADER** and **bV5CSType** equals PROFILE_EMBEDDED or PROFILE_LINKED.

If a profile is linked, the path of the profile can be any fully qualified name (including a network path) that can be opened using the **CreateFile** function.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures

BLENDFUNCTION

The **BLENDFUNCTION** structure controls blending by specifying the blending functions for source and destination bitmaps.

```
typedef struct _BLENDFUNCTION {
  BYTE     BlendOp;
  BYTE     BlendFlags;
  BYTE     SourceConstantAlpha;
  BYTE     AlphaFormat;
}BLENDFUNCTION, *PBLENDFUNCTION, *LPBLENDFUNCTION;
```

Members

BlendOp
Specifies the source blend operation. Currently, the only source and destination blend operation that has been defined is AC_SRC_OVER. For details, see the following Remarks section.

BlendFlags
Must be zero.

SourceConstantAlpha
Specifies an alpha transparency value to be used on the entire source bitmap. The **SourceConstantAlpha** value is combined with any per-pixel alpha values in the source bitmap. If you set **SourceConstantAlpha** to 0, it is assumed that your image

is transparent. Set the **SourceConstantAlpha** value to 255 (opaque) when you only want to use per-pixel alpha values.

AlphaFormat

This member controls the way the source and destination bitmaps are interpreted. **AlphaFormat** has the following value:

Value	Meaning
AC_SRC_ALPHA	This flag is set when the bitmap has an Alpha channel (that is, per-pixel alpha). Note that the APIs use premultiplied alpha, which means that the red, green and blue channel values in the bitmap must be premultiplied with the alpha channel value. For example, if the alpha channel value is x, the red, green and blue channels must be multiplied by x and divided by 0xff prior to the call.

Remarks

When the AC_SRC_OVER operation is used, the source bitmap is placed over the destination bitmap based on the alpha values of the source pixels.

If the source bitmap has no per-pixel alpha value, the blend is based on the **SourceConstantAlpha** value, as shown in the following table:

Dst.Red	=	Src.Red	* SourceConstantAlpha + (1 - SourceConstantAlpha) * Dst.Red
Dst.Green	=	Src.Green	* SourceConstantAlpha + (1 - SourceConstantAlpha) * Dst.Green
Dst.Blue	=	Src.Blue	* SourceConstantAlpha + (1 - SourceConstantAlpha) * Dst.Blue

If the source bitmap has per-pixel alpha and the **SourceConstantAlpha** is *not* used (that is, it equals 0xff), the blend is based on the per-pixel alpha, as shown in the following table:

Dst.Red	=	Src.Red	+ (1 - Src.Alpha) * Dst.Red
Dst.Green	=	Src.Green	+ (1 - Src.Alpha) * Dst.Green
Dst.Blue	=	Src.Blue	+ (1 - Src.Alpha) * Dst.Blue

If the destination bitmap has an alpha channel, then:

Dst.alpha	=	Src.Alpha	+ (1 - SrcAlpha) * Dst.Alpha

If the source has per-pixel alpha and the **SourceConstantAlpha** is used (that is, it is not 0xff), the source is pre-multiplied by the **SourceConstantAlpha** and then the blend is based on the per-pixel alpha. The following tables show this:

```
Src.Red    = Src.Red      * SourceConstantAlpha;
Src.Green  = Src.Green    * SourceConstantAlpha;
Src.Blue   = Src.Blue     * SourceConstantAlpha;
Src.Alpha  = Src.Alpha    * SourceConstantAlpha;
Dst.Red    = Src.Red      + (1 - Src.Alpha) * Dst.Red
Dst.Green  = Src.Green    + (1 - Src.Alpha) * Dst.Green
Dst.Blue   = Src.Blue     + (1 - Src.Alpha) * Dst.Blue
Dst.Alpha  = Src.Alpha    + (1 - Src.Alpha) * Dst.Alpha
```

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures

COLORADJUSTMENT

The **COLORADJUSTMENT** structure defines the color adjustment values used by the **StretchBlt** and **StretchDIBits** functions when the stretch mode is HALFTONE. You can set the color adjustment values by calling the **SetColorAdjustment** function.

```
typedef struct  tagCOLORADJUSTMENT {
  WORD  caSize;
  WORD  caFlags;
  WORD  caIlluminantIndex;
  WORD  caRedGamma;
  WORD  caGreenGamma;
  WORD  caBlueGamma;
  WORD  caReferenceBlack;
  WORD  caReferenceWhite;
  SHORT caContrast;
  SHORT caBrightness;
  SHORT caColorfulness;
  SHORT caRedGreenTint;
} COLORADJUSTMENT, *PCOLORADJUSTMENT;
```

Members

caSize

Specifies the size, in bytes, of the structure.

caFlags

Specifies how the output image should be prepared. This member may be set to NULL or any combination of the following values:

Value	Meaning
CA_LOG_FILTER	Specifies that a logarithmic function should be applied to the final density of the output colors. This will increase the color contrast when the luminance is low.
CA_NEGATIVE	Specifies that the negative of the original image should be displayed.

caIlluminantIndex

Specifies the type of standard light source under which the image is viewed. This member may be set to one of the following values:

Value	Meaning
ILLUMINANT_DEVICE_DEFAULT	Device's default. Standard used by output devices.
ILLUMINANT_A	Tungsten lamp.
ILLUMINANT_B	Noon sunlight.
ILLUMINANT_C	NTSC daylight.
ILLUMINANT_D50	Normal print.
ILLUMINANT_D55	Bond paper print.
ILLUMINANT_D65	Standard daylight. Standard for CRTs and pictures.
ILLUMINANT_D75	Northern daylight.
ILLUMINANT_DAYLIGHT	Same as ILLUMINANT_C.
ILLUMINANT_F2	Cool white lamp.
ILLUMINANT_FLUORESCENT	Same as ILLUMINANT_F2.
ILLUMINANT_NTSC	Same as ILLUMINANT_C.
ILLUMINANT_TUNGSTEN	Same as ILLUMINANT_A.

caRedGamma

Specifies the *n*th power gamma-correction value for the red primary of the source colors. The value must be in the range from 2500 to 65,000. A value of 10,000 means no gamma correction.

caGreenGamma

Specifies the *n*th power gamma-correction value for the green primary of the source colors. The value must be in the range from 2500 to 65,000. A value of 10,000 means no gamma correction.

caBlueGamma

Specifies the *n*th power gamma-correction value for the blue primary of the source colors. The value must be in the range from 2500 to 65,000. A value of 10,000 means no gamma correction.

caReferenceBlack

Specifies the black reference for the source colors. Any colors that are darker than this are treated as black. The value must be in the range from 0 to 4000.

caReferenceWhite

Specifies the white reference for the source colors. Any colors that are lighter than this are treated as white. The value must be in the range from 6000 to 10,000.

caContrast

Specifies the amount of contrast to be applied to the source object. The value must be in the range from –100 to 100. A value of 0 means no contrast adjustment.

caBrightness

Specifies the amount of brightness to be applied to the source object. The value must be in the range from –100 to 100. A value of 0 means no brightness adjustment.

caColorfulness

Specifies the amount of colorfulness to be applied to the source object. The value must be in the range from –100 to 100. A value of 0 means no colorfulness adjustment.

caRedGreenTint

Specifies the amount of red or green tint adjustment to be applied to the source object. The value must be in the range from –100 to 100. Positive numbers adjust towards red and negative numbers adjust towards green. Zero means no tint adjustment.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures, **GetColorAdjustment**, **SetColorAdjustment**, **SetStretchBltMode**, **StretchBlt**, **StretchDIBits**

DIBSECTION

The **DIBSECTION** structure contains information about a DIB created by calling the **CreateDIBSection** function. A **DIBSECTION** structure includes information about the bitmap's dimensions, color format, color masks, optional file mapping object, and optional bit values storage offset. An application can obtain a filled-in **DIBSECTION** structure for a given DIB by calling the **GetObject** function.

```
typedef struct tagDIBSECTION {
  BITMAP              dsBm;
  BITMAPINFOHEADER    dsBmih;
  DWORD               dsBitfields[3];
  HANDLE              dshSection;
  DWORD               dsOffset;
} DIBSECTION, *PDIBSECTION;
```

Members

dsBm

A **BITMAP** data structure that contains information about the DIB: its type, its dimensions, its color capacities, and a pointer to its bit values.

dsBmih

A bitmap information header structure that contains information about the color format of the DIB.

A bitmap information header structure may be one of the following:

Operating system	Bitmap information header
Windows NT 3.51 and earlier	**BITMAPINFOHEADER**
Windows NT 4.0 and Windows 95	**BITMAPV4HEADER**
Windows 2000 and Windows 98	**BITMAPV5HEADER**.

dsBitfields

Specifies three **DWORD** color masks for the DIB. This field is only valid when the **BitCount** member of the ***Bitmap Information Header*** structure has a value greater than 8. Each color mask indicates the bits within a **DWORD** that are used to encode one of the three color channels (red, green, and blue).

dshSection

Contains a handle to the file mapping object that the **CreateDIBSection** function used to create the DIB. If **CreateDIBSection** was called with a NULL value for its *hSection* parameter, causing the system to allocate memory for the bitmap, the **dshSection** member will be NULL.

dsOffset

Specifies the offset to the bitmap's bit values within the file mapping object referenced by **dshSection**. If **dshSection** is NULL, the **dsOffset value** has no meaning.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **BITMAP**, **BITMAPINFOHEADER**, **CreateDIBSection**, **GetDIBColorTable**, **GetObject**

GRADIENT_RECT

The **GRADIENT_RECT** structure specifies the index of two vertices in the *pVertex* array. These two vertices form the upper-left and lower-right boundaries of a rectangle.

```
typedef struct _GRADIENT_RECT {
  ULONG    UpperLeft;
  ULONG    LowerRight;
}GRADIENT_RECT, *PGRADIENT_RECT;
```

Members

UpperLeft
Specifies the upper-left corner of a rectangle.

LowerRight
Specifies the lower-right corner of a rectangle.

Remarks

The **GRADIENT_RECT** structure contains the values used in the *dwMode* parameter of the **GradientFill** function. For related **GradientFill** structures, see **GRADIENT_TRIANGLE** and **TRIVERTEX**.

For an example of the use of this structure, see *Drawing a Shaded Rectangle*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures

GRADIENT_TRIANGLE

The **GRADIENT_TRIANGLE** structure specifies the index of three vertices in the *pVertex* array. These three vertices form one triangle.

```
typedef struct _GRADIENT_TRIANGLE {
  ULONG    Vertex1;
  ULONG    Vertex2;
  ULONG    Vertex3;
}GRADIENT_TRIANGLE, *PGRADIENT_TRIANGLE;
```

Members

Vertex1
First point of the triangle where sides intersect.

Vertex2
Second point of the triangle where sides intersect.

Vertex3
Third point of the triangle where sides intersect.

Remarks

The **GRADIENT_TRIANGLE** structure contains the values used in the *dwMode* parameter of the **GradientFill** function. For related **GradientFill** structures, see **GRADIENT_RECT** and **TRIVERTEX**.

For an example of this function, see *Drawing a Shaded Triangle.*

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **GradientFill**, **GRADIENT_RECT**, **TRIVERTEX**

RGBQUAD

The **RGBQUAD** structure describes a color consisting of relative intensities of red, green, and blue.

```
typedef struct tagRGBQUAD {
  BYTE    rgbBlue;
  BYTE    rgbGreen;
  BYTE    rgbRed;
  BYTE    rgbReserved;
} RGBQUAD;
```

Members

rgbBlue
Specifies the intensity of blue in the color.

rgbGreen
Specifies the intensity of green in the color.

rgbRed
Specifies the intensity of red in the color.

rgbReserved
Reserved; must be zero.

Remarks

The **bmiColors** member of the **BITMAPINFO** structure consists of an array of **RGBQUAD** structures.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures, **BITMAPINFO**, **CreateDIBitmap**, **CreateDIBSection**, **GetDIBits**, **SetDIBits**, **SetDIBitsToDevice**, **StretchDIBits**

RGBTRIPLE

The **RGBTRIPLE** structure describes a color consisting of relative intensities of red, green, and blue. The **bmciColors** member of the **BITMAPCOREINFO** structure consists of an array of **RGBTRIPLE** structures.

```
typedef struct tagRGBTRIPLE {
  BYTE rgbtBlue;
  BYTE rgbtGreen;
  BYTE rgbtRed;
} RGBTRIPLE;
```

Members

rgbtBlue
Specifies the intensity of blue in the color.

rgbtGreen
Specifies the intensity of green in the color.

rgbtRed
Specifies the intensity of red in the color.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Bitmaps Overview, Bitmap Structures, **BITMAPCOREINFO**

SIZE

The **SIZE** structure specifies the width and height of a rectangle.

```
typedef struct tagSIZE {
  LONG cx;
  LONG cy;
} SIZE, *PSIZE;
```

Members

cx

Specifies the rectangle's width.

cy

Specifies the rectangle's height.

Remarks

The rectangle dimensions stored in this structure may correspond to viewport extents, window extents, text extents, bitmap dimensions, or the aspect-ratio filter for some extended functions.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in windef.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures, **GetAspectRatioFilterEx**, **GetBitmapDimensionEx**, **GetTextExtentPoint32**, **GetViewportExtEx**, **GetWindowExtEx**, **ScaleViewportExtEx**, **ScaleWindowExtEx**, **SetBitmapDimensionEx**, **SetViewportExtEx**, **SetWindowExtEx**

TRIVERTEX

The **TRIVERTEX** structure contains color information and position information.

```
typedef struct _TRIVERTEX {
  LONG        x;
  Long        y;
  COLOR16     Red;
  COLOR16     Green;
  COLOR16     Blue;
  COLOR16     Alpha;
}TRIVERTEX, *PTRIVERTEX;
```

Members

x

Specifies the x-coordinate, in logical units, of the upper-left corner of the rectangle.

y

Specifies the y-coordinate, in logical units, of the upper-left corner of the rectangle.

Red

Indicates color information at the point of x, y.

Green

Indicates color information at the point of x, y.

Blue

Indicates color information at the point of x, y.

Alpha

Indicates color information at the point of x, y.

Remarks

In the **TRIVERTEX** structure, x and y indicate position in the same manner as in the **POINTL** structure contained in the wtypes.h header file. **Red**, **Green**, **Blue**, and **Alpha** members indicate color information at the point x, y. The color information of each channel is specified as a value from 0x0000 to 0xff00. This allows higher color resolution for an object that has been split into small triangles for display. The **TRIVERTEX** structure contains information needed by the *pVertex* parameter of **GradientFill**. For an example of the use of this structure, see *Drawing a Shaded Triangle* and *Drawing a Shaded Rectangle*.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Structures

Bitmap Macros

MAKEROP4

The **MAKEROP4** macro creates a quaternary raster operation code for use with the **MaskBlt** function. The macro takes two ternary raster operation codes as input, one for the foreground and one for the background, and packs their Boolean operation indexes into the high-order word of a 32-bit value. The low-order word of this value will be ignored.

```
DWORD MAKEROP4(
  DWORD fore, // foreground ternary raster operation code
  DWORD back  // background ternary raster operation code
);
```

Parameters

fore

Specifies a foreground ternary raster operation code.

back

Specifies a background ternary raster operation code

Return Values

The return value is a **DWORD** quaternary raster operation code for use with the **MaskBlt** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Bitmaps Overview, Bitmap Macros, **MaskBlt**

CHAPTER 7

Brushes

A *brush* is a graphics tool that a Win32-based application uses to paint the interior of polygons, ellipses, and paths. Drawing applications use brushes to paint shapes; word processing applications use brushes to paint rules; computer-aided design (CAD) applications use brushes to paint the interiors of cross-section views; and spreadsheet applications use brushes to paint the sections of pie charts and the bars in bar graphs.

About Brushes

There are two types of brushes: logical and physical. A *logical brush* is a description of the ideal bitmap that an application uses to paint shapes. A *physical brush* is the actual bitmap that a device driver creates based on an application's logical-brush definition. For more information about bitmaps, see *Bitmaps*.

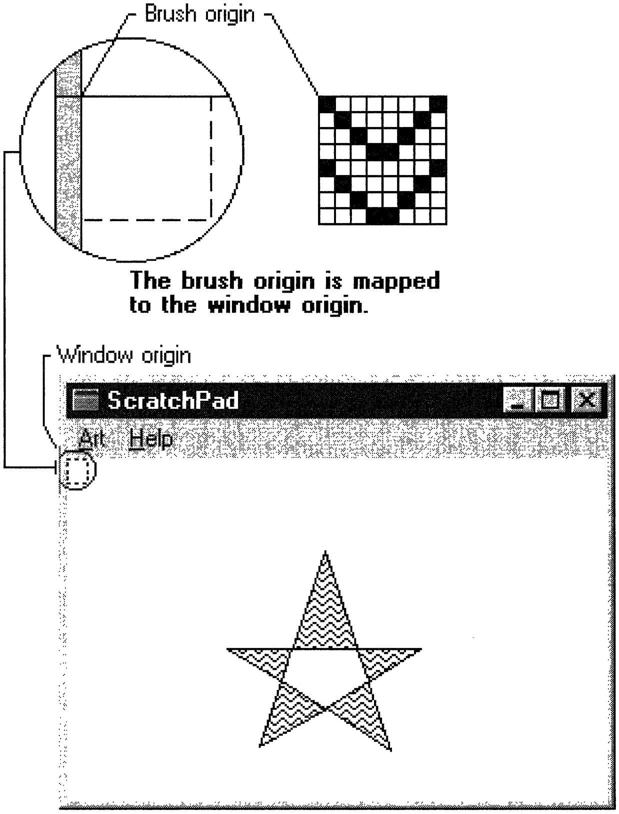
When an application calls one of the functions that creates a brush, it retrieves a handle that identifies a logical brush. When the application passes this handle to the **SelectObject** function, the device driver for the corresponding display or printer creates the physical brush.

Brush Origin

When an application calls a drawing function to paint a shape, the system positions a brush at the start of the paint operation and maps a pixel in the brush bitmap to the client area at the *window origin*, which is the upper-left corner of the window. The coordinates of the pixel that the system maps are called the *brush origin*. The default brush origin is located in the upper-left corner of the brush bitmap, at the coordinates (0,0). The system then copies the brush across the client area, forming a pattern that is as tall as the bitmap. The copy operation continues, row by row, until the entire client area is filled. However, the brush pattern is visible only within the boundaries of the specified shape.

There are instances when the default brush origin should not be used. For example, it may be necessary for an application to use the same brush to paint the backgrounds of its parent and child windows and blend a child window's background with that of the parent window. To do this, the application should reset the brush origin by calling the **SetBrushOrgEx** function and shifting the origin the required number of pixels. (An application can retrieve the current brush origin by calling the **GetBrushOrgEx** function.)

Figure 7-1 shows a five-pointed star filled by using an application-defined brush. The illustration shows a zoomed image of the brush, as well as the location to which it was mapped at the beginning of the paint operation.

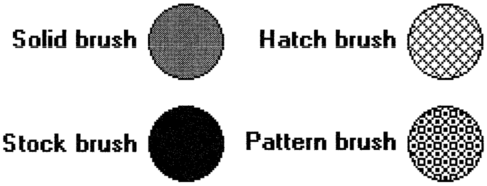


The brush origin is mapped to the window origin.

Figure 7-1: Using an application-defined brush to fill a figure.

Logical Brush Types

There are four types of logical brushes: **solid**, **stock**, **hatch**, and **pattern**. These brushes are shown in the following illustration.



The stock and hatch types each have several predefined brushes.

The **CreateBrushIndirect** function creates a logical brush with a specified style, color, and pattern.

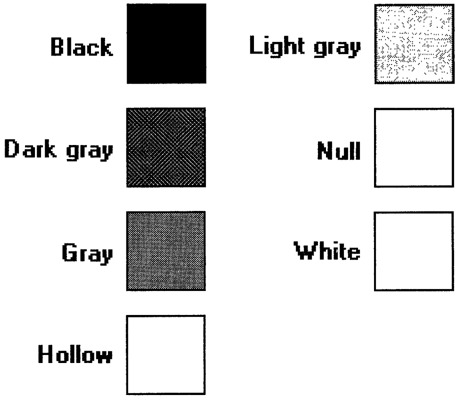
Solid Brush

A *solid brush* is a logical brush that contains 64 pixels of the same color. An application can create a solid logical brush by calling the **CreateSolidBrush** function, specifying the color of the brush required. After creating the solid brush, the application can select it into its device context and use it to paint filled shapes.

Stock Brush

There are seven predefined logical stock brushes maintained by the graphics device interface (GDI). There are also 21 predefined logical stock brushes maintained by the window management interface (USER).

The following rectangles were painted by using the seven predefined stock brushes.

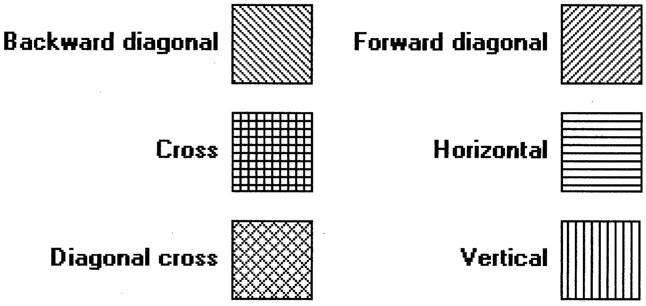


An application can retrieve a handle identifying one of the seven stock brushes by calling the **GetStockObject** function, specifying the brush type.

The 21 stock brushes maintained by the window management interface correspond to the colors of window elements such as menus, scroll bars, and buttons. An application can obtain a handle identifying one of these brushes by calling the **GetSysColorBrush** function and specifying a system-color value. An application can retrieve the color corresponding to a particular window element by calling the **GetSysColor** function. An application can set the color corresponding to a window element by calling the **SetSysColors** function.

Hatch Brush

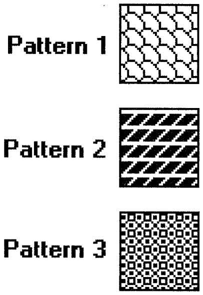
There are six predefined logical hatch brushes maintained by GDI. The following rectangles were painted by using the six predefined hatch brushes.



An application can create a hatch brush by calling the **CreateHatchBrush** function, specifying one of the six hatch styles.

Pattern Brush

A pattern (or custom) brush is created from an application-defined bitmap or device-independent bitmap (DIB). The following rectangles were painted by using different pattern brushes.



To create a logical pattern brush, an application must first create a bitmap. After creating the bitmap, the application can create the logical pattern brush by calling the **CreatePatternBrush** or **CreateDIBPatternBrushPt** function, supplying a handle that identifies the bitmap (or DIB). The brushes that appear in the preceding illustration were created from monochrome bitmaps. For a description of bitmaps, DIBs, and the functions that create them, see *Bitmaps*.

Pattern Block Transfer

The name of the **PatBlt** function (an abbreviation for pattern block transfer) implies that this function simply replicates the brush (or pattern) until it fills a specified rectangle. However, the function is actually much more powerful. Before replicating the brush, it combines the color data for the pattern with the color data for the existing pixels on the video display by using a raster operation (ROP). An ROP is a bitwise operation that is applied to the bits of color data for the replicated brush and the bits of color data for the target rectangle on the display device. There are 256 ROPs; however, the **PatBlt** function recognizes only those that require a pattern and a destination (not those that require a source). The following table identifies the most common ROPs.

ROP	Description
PATCOPY	Copies the pattern to the destination bitmap.
PATINVERT	Combines the destination bitmap with the pattern by using the Boolean XOR operator.
DSTINVERT	Inverts the destination bitmap.
BLACKNESS	Turns all output to binary zeroes.
WHITENESS	Turns all output to binary ones.

For more information, see *Raster Operation Codes.*

ICM-Enabled Brush Functions

Microsoft Windows 98 and Microsoft Windows 2000 have been designed to work with Microsoft Image Color Management (ICM). ICM technology ensures that a color image, graphic, or text object is rendered as close as possible to its original intent on any device, despite differences in imaging technologies and color capabilities between devices. Whether you are scanning an image or other graphic on a color scanner, downloading it over the Internet, viewing or editing it on the screen, or outputting it to paper, film, or other media, ICM 2.0 helps you keep its colors consistent and accurate. For more information on ICM, see *About Image Color Management Version 2.0.*

The following brush functions are enabled for use with ICM:

- **CreateBrushIndirect**
- **CreateDIBPatternBrush**
- **CreateDIBPatternBrushPt**
- **CreateHatchBrush**
- **CreatePatternBrush**
- **CreateSolidBrush**

Brush Reference

Brush Functions

CreateBrushIndirect

The **CreateBrushIndirect** function creates a logical brush that has the specified style, color, and pattern.

```
HBRUSH CreateBrushIndirect(
  CONST LOGBRUSH *lplb   // brush information
);
```

Parameters

lplb

[in] Pointer to a **LOGBRUSH** structure that contains information about the brush.

Return Values

If the function succeeds, the return value identifies a logical brush.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes.

After an application creates a brush by calling **CreateBrushIndirect**, it can select it into any device context by calling the **SelectObject** function.

A brush created by using a monochrome bitmap (one color plane, one bit per pixel) is drawn using the current text and background colors. Pixels represented by a bit set to 0 are drawn with the current text color; pixels represented by a bit set to 1 are drawn with the current background color.

If the **lbStyle** member of the **LOGBRUSH** structure pointed to by *lplb* is BS_PATTERN, the bitmap pointed to by the **lbHatch** member of that structure cannot be a DIB section. A DIB section is a bitmap created by the **CreateDIBSection** function. If **lbStyle** is BS_PATTERN and the bitmap is a DIB section, the **CreateBrushIndirect** function fails.

When you no longer need the brush, call the **DeleteObject** function to delete it.

ICM: No color is done at brush creation. However, color management is performed when the brush is selected into an ICM-enabled device context.

Windows 95: Creating brushes from bitmaps or DIBs larger than 8 by 8 pixels is not supported. If a larger bitmap is specified, only a portion of the bitmap is used.

Windows NT/2000 and Windows 98: Brushes can be created from bitmaps or DIBs larger than 8 by 8 pixels.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Brushes Overview, Brush Functions, **CreateDIBSection**, **DeleteObject**, **GetBrushOrgEx**, **LOGBRUSH**, **SelectObject**, **SetBrushOrgEx**

CreateDIBPatternBrushPt

The **CreateDIBPatternBrushPt** function creates a logical brush that has the pattern specified by the device-independent bitmap (DIB).

```
HBRUSH CreateDIBPatternBrushPt(
  CONST VOID *lpPackedDIB,  // bitmap bits
  UINT iUsage               // usage
);
```

Parameters

lpPackedDIB

[in] Pointer to a packed DIB consisting of a **BITMAPINFO** structure immediately followed by an array of bytes defining the pixels of the bitmap.

Windows 95: Creating brushes from bitmaps or DIBs larger than 8 by 8 pixels is not supported. If a larger bitmap is specified, only a portion of the bitmap is used.

Windows NT/2000 and Windows 98: Brushes can be created from bitmaps or DIBs larger than 8 by 8 pixels.

iUsage

[in] Specifies whether the **bmiColors** member of the **BITMAPINFO** structure contains a valid color table and, if so, whether the entries in this color table contain explicit red, green, blue (RGB) values or palette indices. The *iUsage* parameter must be one of the following values.

Value	Meaning
DIB_PAL_COLORS	A color table is provided and consists of an array of 16-bit indices into the logical palette of the device context into which the brush is to be selected.
DIB_RGB_COLORS	A color table is provided and contains literal RGB values.

Return Values

If the function succeeds, the return value identifies a logical brush.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes.

After an application creates a brush by calling **CreateDIBPatternBrushPt**, it can select that brush into any device context by calling the **SelectObject** function.

When you no longer need the brush, call the **DeleteObject** function to delete it.

ICM: No color is done at brush creation. However, color management is performed when the brush is selected into an ICM-enabled device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Brushes Overview, Brush Functions, **BITMAPINFO**, **CreateDIBPatternBrush**, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**, **DeleteObject**, **GetBrushOrgEx**, **SelectObject**, **SetBrushOrgEx**

CreateHatchBrush

The **CreateHatchBrush** function creates a logical brush that has the specified hatch pattern and color.

```
HBRUSH CreateHatchBrush(
  int fnStyle,       // hatch style
  COLORREF clrref    // foreground color
);
```

Parameters

fnStyle

[in] Specifies the hatch style of the brush. This parameter can be one of the following values.

Value	Meaning
HS_BDIAGONAL	45-degree downward left-to-right hatch
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch

Value	Meaning
HS_FDIAGONAL	45-degree upward left-to-right hatch
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

clrref

[in] Specifies the foreground color of the brush that is used for the hatches. To create a **COLORREF** color value, use the **RGB** macro.

Return Values

If the function succeeds, the return value identifies a logical brush.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes.

After an application creates a brush by calling **CreateHatchBrush**, it can select that brush into any device context by calling the **SelectObject** function.

If an application uses a hatch brush to fill the backgrounds of both a parent and a child window with matching color, it may be necessary to set the brush origin before painting the background of the child window. You can do this by having your application call the **SetBrushOrgEx** function. Your application can retrieve the current brush origin by calling the **GetBrushOrgEx** function.

When you no longer need the brush, call the **DeleteObject** function to delete it.

ICM: No color is done at brush creation. However, color management is performed when the brush is selected into an ICM-enabled device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Brushes Overview, Brush Functions, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreatePatternBrush**, **CreateSolidBrush**, **DeleteObject**, **GetBrushOrgEx**, **SelectObject**, **SetBrushOrgEx**, **COLORREF**, **RGB**

CreatePatternBrush

The **CreatePatternBrush** function creates a logical brush with the specified bitmap pattern. The bitmap can be a DIB section bitmap, which is created by the **CreateDIBSection** function.

```
HBRUSH CreatePatternBrush(
  HBITMAP hbmp   // handle to bitmap
);
```

Parameters

hbmp

[in] Handle to the bitmap to be used to create the logical brush.

Windows 95: Creating brushes from bitmaps or DIBs larger than 8 by 8 pixels is not supported. If a larger bitmap is specified, only a portion of the bitmap is used.

Windows NT/2000 and Windows 98: Brushes can be created from bitmaps or DIBs larger than 8 by 8 pixels.

Return Values

If the function succeeds, the return value identifies a logical brush.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A pattern brush is a bitmap that the system uses to paint the interiors of filled shapes.

After an application creates a brush by calling **CreatePatternBrush**, it can select that brush into any device context by calling the **SelectObject** function.

You can delete a pattern brush without affecting the associated bitmap by using the **DeleteObject** function. Therefore, you can then use this bitmap to create any number of pattern brushes.

A brush created by using a monochrome (1 bit per pixel) bitmap has the text and background colors of the device context to which it is drawn. Pixels represented by a 0 bit are drawn with the current text color; pixels represented by a 1 bit are drawn with the current background color.

ICM: No color is done at brush creation. However, color management is performed when the brush is selected into an ICM-enabled device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.

Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Brushes Overview, Brush Functions, **CreateBitmap**, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreateDIBSection**, **CreateHatchBrush**, **DeleteObject**, **GetBrushOrgEx**, **LoadBitmap**, **SelectObject**, **SetBrushOrgEx**

CreateSolidBrush

The **CreateSolidBrush** function creates a logical brush that has the specified solid color.

```
HBRUSH CreateSolidBrush(
  COLORREF crColor   // brush color value
);
```

Parameters

crColor
[in] Specifies the color of the brush. To create a **COLORREF** color value, use the **RGB** macro.

Return Values

If the function succeeds, the return value identifies a logical brush.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A solid brush is a bitmap that the system uses to paint the interiors of filled shapes.

After an application creates a brush by calling **CreateSolidBrush**, it can select that brush into any device context by calling the **SelectObject** function.

ICM: No color is done at brush creation. However, color management is performed when the brush is selected into an ICM-enabled device context.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Brushes Overview, Brush Functions, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreateHatchBrush**, **CreatePatternBrush**, **DeleteObject**, **SelectObject**, **COLORREF**, **RGB**

GetBrushOrgEx

The **GetBrushOrgEx** function retrieves the current brush origin for the specified device context. This function replaces the **GetBrushOrg** function.

```
BOOL GetBrushOrgEx(
  HDC hdc,        // handle to DC
  LPPOINT lppt    // coordinates of origin
);
```

Parameters

hdc

[in] Handle to the device context.

lppt

[out] Pointer to a **POINT** structure that receives the brush origin, in device coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes.

The brush origin is a set of coordinates with values between 0 and 7, specifying the location of one pixel in the bitmap. The default brush origin coordinates are (0,0). For horizontal coordinates, the value 0 corresponds to the leftmost column of pixels; the value 7 corresponds to the rightmost column. For vertical coordinates, the value 0 corresponds to the uppermost row of pixels; the value 7 corresponds to the lowermost row. When the system positions the brush at the start of any painting operation, it maps the origin of the brush to the location in the window's client area specified by the brush origin. For example, if the origin is set to (2,3), the system maps the origin of the brush (0,0) to the location (2,3) on the window's client area.

If an application uses a brush to fill the backgrounds of both a parent and a child window with matching colors, it may be necessary to set the brush origin after painting the parent window but before painting the child window.

Windows NT/2000: The system automatically tracks the origin of all window-managed device contexts and adjusts their brushes as necessary to maintain an alignment of patterns on the surface.

Windows 95/98: Automatic tracking of the brush origin is not supported. Applications must use the **UnrealizeObject**, **SetBrushOrgEx**, and **SelectObject** functions to align the brush before using it.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Brushes Overview, Brush Functions, **POINT**, **SelectObject**, **SetBrushOrgEx**, **UnrealizeObject**

GetSysColorBrush

The **GetSysColorBrush** function retrieves a handle identifying a logical brush that corresponds to the specified color index.

```
HBRUSH GetSysColorBrush(
  int nIndex  // system color index
);
```

Parameters

nIndex

[in] Specifies a color index. This value corresponds to the color used to paint one of the 21 window elements.

Return Values

The return value identifies a logical brush.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes. An application can retrieve the current system colors by calling the **GetSysColor** function. An application can set the current system colors by calling the **SetSysColors** function.

An application must not register a window class for a window using a system brush.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Brushes Overview, Brush Functions, **GetSysColor**, **SetSysColors**

PatBlt

The **PatBlt** function paints the specified rectangle using the brush that is currently selected into the specified device context. The brush color and the surface color or colors are combined by using the specified raster operation.

```
BOOL PatBlt(
  HDC hdc,        // handle to DC
  int nXLeft,     // x-coord of upper-left rectangle corner
  int nYLeft,     // y-coord of upper-left rectangle corner
  int nWidth,     // width of rectangle
  int nHeight,    // height of rectangle
  DWORD dwRop     // raster operation code
);
```

Parameters

hdc
[in] Handle to the device context.

nXLeft
[in] Specifies the x-coordinate, in logical units, of the upper-left corner of the rectangle to be filled.

nYLeft
[in] Specifies the y-coordinate, in logical units, of the upper-left corner of the rectangle to be filled.

nWidth
[in] Specifies the width, in logical units, of the rectangle.

nHeight

[in] Specifies the height, in logical units, of the rectangle.

dwRop

[in] Specifies the raster operation code. This code can be one of the following values.

Value	Meaning
PATCOPY	Copies the specified pattern into the destination bitmap.
PATINVERT	Combines the colors of the specified pattern with the colors of the destination rectangle by using the Boolean XOR operator.
DSTINVERT	Inverts the destination rectangle.
BLACKNESS	Fills the destination rectangle using the color associated with index 0 in the physical palette. (This color is black for the default physical palette.)
WHITENESS	Fills the destination rectangle using the color associated with index 1 in the physical palette. (This color is white for the default physical palette.)

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The values of the *dwRop* parameter for this function are a limited subset of the full 256 ternary raster-operation codes; in particular, an operation code that refers to a source rectangle cannot be used.

Not all devices support the **PatBlt** function. For more information, see the description of the RC_BITBLT capability in the **GetDeviceCaps** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Brushes Overview, Brush Functions, **GetDeviceCaps**

SetBrushOrgEx

The **SetBrushOrgEx** function sets the brush origin that GDI assigns to the *next* brush an application selects into the specified device context.

```
BOOL SetBrushOrgEx(
  HDC hdc,         // handle to device context
  int nXOrg,       // x-coord of new origin
  int nYOrg,       // y-coord of new origin
  LPPOINT lppt     // points to previous brush origin
);
```

Parameters

hdc

[in] Handle to the device context.

nXOrg

[in] Specifies the x-coordinate, in device units, of the new brush origin. If this value is greater than the brush width, its value is reduced using the modulus operator (*nXOrg* **mod** brush width).

nYOrg

[in] Specifies the y-coordinate, in device units, of the new brush origin. If this value is greater than the brush height, its value is reduced using the modulus operator (*nYOrg* **mod** brush height).

lppt

[out] Pointer to a **POINT** structure that receives the previous brush origin.

This parameter can be NULL if the previous brush origin is not required.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A brush is a bitmap that the system uses to paint the interiors of filled shapes.

The brush origin is a pair of coordinates specifying the location of one pixel in the bitmap. The default brush origin coordinates are (0,0). For horizontal coordinates, the value 0 corresponds to the leftmost column of pixels; the width corresponds to the rightmost column. For vertical coordinates, the value 0 corresponds to the uppermost row of pixels; the height corresponds to the lowermost row.

The system automatically tracks the origin of all window-managed device contexts and adjusts their brushes as necessary to maintain an alignment of patterns on the surface. The brush origin that is set with this call is relative to the upper-left corner of the client area.

An application should call **SetBrushOrgEx** after setting the bitmap stretching mode to HALFTONE by using **SetStretchBltMode**. This must be done to avoid brush misalignment.

Windows NT/2000: The system automatically tracks the origin of all window-managed device contexts and adjusts their brushes as necessary to maintain an alignment of patterns on the surface.

Windows 95/98: Automatic tracking of the brush origin is not supported. Applications must use the **UnrealizeObject**, **SetBrushOrgEx**, and **SelectObject** functions to align the brush before using it.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Brushes Overview, Brush Functions, **GetBrushOrgEx**, **POINT**, **SelectObject**, **SetStretchBltMode**, **UnrealizeObject**

Brush Structures

LOGBRUSH

The **LOGBRUSH** structure defines the style, color, and pattern of a physical brush. It is used by the **CreateBrushIndirect** and **ExtCreatePen** functions.

```
typedef struct tagLOGBRUSH {
  UINT      lbStyle;
  COLORREF lbColor;
  LONG      lbHatch;
} LOGBRUSH, *PLOGBRUSH;
```

Members

lbStyle

Specifies the brush style. The **lbStyle** member must be one of the following styles.

Value	Meaning
BS_DIBPATTERN	A pattern brush defined by a device-independent bitmap (DIB) specification. If **lbStyle** is BS_DIBPATTERN, the **lbHatch** member contains a handle to a packed DIB. For more information, see discussion in **lbHatch**. **Windows 95:** Creating brushes from bitmaps or DIBs larger than 8 by 8 pixels is not supported. If a larger bitmap is specified, only a portion of the bitmap is used.
BS_DIBPATTERN8X8	Same as BS_DIBPATTERN.
BS_DIBPATTERNPT	A pattern brush defined by a device-independent bitmap (DIB) specification. If **lbStyle** is BS_DIBPATTERNPT, the **lbHatch** member contains a pointer to a packed DIB. For more information, see discussion in **lbHatch**.
BS_HATCHED	Hatched brush.
BS_HOLLOW	Hollow brush.
BS_NULL	Same as BS_HOLLOW.
BS_PATTERN	Pattern brush defined by a memory bitmap.
BS_PATTERN8X8	Same as BS_PATTERN.
BS_SOLID	Solid brush.

lbColor

Specifies the color in which the brush is to be drawn. If **lbStyle** is the BS_HOLLOW or BS_PATTERN style, **lbColor** is ignored.

If **lbStyle** is BS_DIBPATTERN or BS_DIBPATTERNPT, the low-order word of **lbColor** specifies whether the **bmiColors** members of the **BITMAPINFO** structure contain explicit red, green, blue (RGB) values or indices into the currently realized logical palette. The **lbColor** member must be one of the following values.

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indices into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

If **lbStyle** is BS_HATCHED or BS_SOLID, **lbColor** is a **COLORREF** color value. To create a **COLORREF** color value, use the **RGB** macro.

lbHatch

Specifies a hatch style. The meaning depends on the brush style defined by **lbStyle**.

If **lbStyle** is BS_DIBPATTERN, the **lbHatch** member contains a handle to a packed DIB. To obtain this handle, an application calls the **GlobalAlloc** function with GMEM_MOVEABLE (or **LocalAlloc** with LMEM_MOVEABLE) to allocate a block of memory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes that define the pixels of the bitmap.

If **lbStyle** is BS_DIBPATTERNPT, the **lbHatch** member contains a pointer to a packed DIB. The pointer derives from the memory block created by **LocalAlloc** with LMEM_FIXED set or by **GlobalAlloc** with GMEM_FIXED set, or it is the pointer returned by a call like **LocalLock** (handle_to_the_dib). A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes that define the pixels of the bitmap.

If **lbStyle** is BS_HATCHED, the **lbHatch** member specifies the orientation of the lines used to create the hatch. It can be one of the following values.

Value	Meaning
HS_BDIAGONAL	A 45-degree upward, left-to-right hatch
HS_CROSS	Horizontal and vertical cross-hatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	A 45-degree downward, left-to-right hatch
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

If **lbStyle** is BS_PATTERN, **lbHatch** is a handle to the bitmap that defines the pattern. The bitmap cannot be a DIB section bitmap, which is created by the **CreateDIBSection** function.

If **lbStyle** is BS_SOLID or BS_HOLLOW, **lbHatch** is ignored.

Remarks

Although **lbColor** controls the foreground color of a hatch brush, the **SetBkMode** and **SetBkColor** functions control the background color.

Windows 95: Creating brushes from bitmaps or DIBs larger than 8 by 8 pixels is not supported. If a larger bitmap is specified, only a portion of the bitmap is used.

Windows NT/2000 and Windows 98: Brushes can be created from bitmaps or DIBs larger than 8 by 8 pixels.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

See Also

Brushes Overview, Brush Structures, **BITMAPINFO**, **CreateBrushIndirect**, **CreateDIBSection**, **ExtCreatePen**, **LOGBRUSH32**, **SetBkColor**, **SetBkMode**, **COLORREF**, **RGB**

LOGBRUSH32

The **LOGBRUSH32** structure defines the style, color, and pattern of a physical brush. It is similar to **LOGBRUSH**, but it is used to maintain compatibility between 32-bit platforms and 64-bit platforms when we record the metafile record on one platform and then play it on another. Thus, it is only used in **EMRCREATEBRUSHINDIRECT**. If the code will only be on one platform, **LOGBRUSH** is sufficient.

```
typedef struct tagLOGBRUSH32 {
  UINT      lbStyle;
  COLORREF lbColor;
  ULONG     lbHatch;
} LOGBRUSH32, *PLOGBRUSH32, NEAR *NPLOGBRUSH32,
FAR *LPLOGBRUSH32;
```

Members

lbStyle

Specifies the brush style. The **lbStyle** member must be one of the following styles.

Value	Meaning
BS_DIBPATTERN	A pattern brush defined by a device-independent bitmap (DIB) specification. If **lbStyle** is BS_DIBPATTERN, the **lbHatch** member contains a handle to a packed DIB. For more information, see discussion in **lbHatch**.
BS_DIBPATTERN8X8	Same as BS_DIBPATTERN.

Value	Meaning
BS_DIBPATTERNPT	A pattern brush defined by a device-independent bitmap (DIB) specification. If **lbStyle** is BS_DIBPATTERNPT, the **lbHatch** member contains a pointer to a packed DIB. For more information, see discussion in **lbHatch**.
BS_HATCHED	Hatched brush.
BS_HOLLOW	Hollow brush.
BS_NULL	Same as BS_HOLLOW.
BS_PATTERN	Pattern brush defined by a memory bitmap.
BS_PATTERN8X8	Same as BS_PATTERN.
BS_SOLID	Solid brush.

lbColor

Specifies the color in which the brush is to be drawn. If **lbStyle** is the BS_HOLLOW or BS_PATTERN style, **lbColor** is ignored.

If **lbStyle** is BS_DIBPATTERN or BS_DIBPATTERNPT, the low-order word of **lbColor** specifies whether the **bmiColors** members of the **BITMAPINFO** structure contain explicit red, green, blue (RGB) values or indices into the currently realized logical palette. The **lbColor** member must be one of the following values.

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indices into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

If **lbStyle** is BS_HATCHED or BS_SOLID, **lbColor** is a **COLORREF** color value. To create a **COLORREF** color value, use the **RGB** macro.

lbHatch

Specifies a hatch style. The meaning depends on the brush style defined by **lbStyle**.

If **lbStyle** is BS_DIBPATTERN, the **lbHatch** member contains a handle to a packed DIB. To obtain this handle, an application calls the **GlobalAlloc** function with GMEM_MOVEABLE (or **LocalAlloc** with LMEM_MOVEABLE) to allocate a block of brmemory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes that define the pixels of the bitmap.

If **lbStyle** is BS_DIBPATTERNPT, the **lbHatch** member contains a pointer to a packed DIB. The pointer derives from the memory block created by **LocalAlloc** with LMEM_FIXED set or by **GlobalAlloc** with GMEM_FIXED set, or it is the pointer returned by a call like **LocalLock** (handle_to_the_dib). A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes that define the pixels of the bitmap.

If **lbStyle** is BS_HATCHED, the **lbHatch** member specifies the orientation of the lines used to create the hatch. It can be one of the following values.

Value	Meaning
HS_BDIAGONAL	A 45-degree upward, left-to-right hatch
HS_CROSS	Horizontal and vertical cross-hatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	A 45-degree downward, left-to-right hatch
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

If **lbStyle** is BS_PATTERN, **lbHatch** is a handle to the bitmap that defines the pattern. The bitmap cannot be a DIB section bitmap, which is created by the **CreateDIBSection** function.

If **lbStyle** is BS_SOLID or BS_HOLLOW, **lbHatch** is ignored.

Remarks

Although **lbColor** controls the foreground color of a hatch brush, the **SetBkMode** and **SetBkColor** functions control the background color.

Brushes can be created from bitmaps or DIBs larger than 8 by 8 pixels.

! Requirements

Windows NT/2000: Requires Windows 2000 or later.
Windows 95/98: Not supported.
Windows CE: Not supported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Brushes Overview, Brush Structures, **BITMAPINFO**, **CreateDIBSection**, **EMRCREATEBRUSHINDIRECT**, **LOGBRUSH**, **SetBkColor**, **SetBkMode**, **COLORREF**, **RGB**

CHAPTER 8

Clipping

About Clipping

Clipping is the process of limiting output to a region or path within the client area of an application's window.

Clipping is used by Win32-based applications in a variety of ways. Word processing and spreadsheet applications clip keyboard input to keep it from appearing in the margins of a page or spreadsheet. Computer-aided design (CAD) and drawing applications clip graphics output to keep it from overwriting the edges of a drawing or picture.

A *clipping region* is a region with edges that are either straight lines or curves. A *clip path* is a region with edges that are straight lines, Bézier curves, or combinations of both. For more information about regions, see *Regions*. For more information about paths, see *Paths*.

Clipping Regions

A clipping region is one of the graphic objects that an application can select into a device context (DC). It is typically rectangular. Some device contexts provide a predefined or default clipping region while others do not. For example, 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. However, when you obtain a device context handle by calling the **GetDC** function with a NULL *hWnd* parameter, or by calling the **CreateDC** function, the DC does not contain a default clipping region. For more information about device contexts returned by the **BeginPaint** function, see ***Painting and Drawing***. For more information about device contexts returned by the **CreateDC** and **GetDC** functions, see *Device Contexts*.

Applications can perform a variety of operations on clipping regions. Some of these operations require a handle identifying the region and some do not. For example, an application can perform the following operations directly on a device context's clipping region:

- Determine whether graphics output appears within the region's borders by passing coordinates of the corresponding line, arc, bitmap, text, or filled shape to the **PtVisible** function.
- Determine whether part of the client area intersects a region by calling the **RectVisible** function.
- Move the existing region by a specified offset by calling the **OffsetClipRgn** function.

- Exclude a rectangular part of the client area from the current clipping region by calling the **ExcludeClipRect** function.
- Combine a rectangular part of the client area with the current clipping region by calling the **IntersectClipRect** function.

After obtaining a handle identifying the clipping region, an application can perform any operation that is common with regions, such as:

- Combining a copy of the current clipping region with a second region by calling the **CombineRgn** function.
- Compare a copy of the current clipping region to a second region by calling the **EqualRgn** function.
- Determine whether a point lies within the interior of a copy of the current clipping region by calling the **PtInRegion** function.

Clip Paths

Like a clipping region, a clip path is another graphics object that an application can select into a device context. Unlike a clipping region, a clip path is always created by an application and it is used for clipping to one or more irregular shapes. For example, an application can use the lines and curves that form the outlines of characters in a string of text to define a clip path.

To create a clip path, it's first necessary to create a path that describes the required irregular shape. Paths are created by calling the appropriate graphical device interface (GDI) drawing functions after calling the **BeginPath** function and before calling the **EndPath** function. This collection of functions is called a path bracket. For more information about paths and path brackets, see ***Paths.***

After the path is created, it can be converted to a clip path by calling the **SelectClipPath** function, identifying a device context, and specifying a usage mode. The usage mode determines how the system combines the new clip path with the device context's original clipping region. The following table describes the usage modes:

Mode	Description
RGN_AND	The clip path includes the intersection (overlapping areas) of the device context's clipping region and the current path.
RGN_COPY	The clip path is the current path.
RGN_DIFF	The clip path includes the device context's clipping region with any intersecting parts of the current path excluded.
RGN_OR	The clip path includes the union (combined areas) of the device context's clipping region and the current path.
RGN_XOR	The clip path includes the union of the device context's clipping region and the current path but excludes the intersection.

Clipping Reference

Clipping Functions

ExcludeClipRect

The **ExcludeClipRect** function creates a new clipping region that consists of the existing clipping region minus the specified rectangle.

```
int ExcludeClipRect(
    HDC hdc,            // handle to DC
    int nLeftRect,      // x-coord of upper-left corner
    int nTopRect,       // y-coord of upper-left corner
    int nRightRect,     // x-coord of lower-right corner
    int nBottomRect     // y-coord of lower-right corner
);
```

Parameters

hdc
[in] Handle to the device context.

nLeftRect
[in] Specifies the logical x-coordinate of the upper-left corner of the rectangle.

nTopRect
[in] Specifies the logical y-coordinate of the upper-left corner of the rectangle.

nRightRect
[in] Specifies the logical x-coordinate of the lower-right corner of the rectangle.

nBottomRect
[in] Specifies the logical y-coordinate of the lower-right corner of the rectangle.

Return Values

The return value specifies the new clipping region's complexity; it can be one of the following values:

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	No region was created.

Remarks

The lower and right edges of the specified rectangle are not excluded from the clipping region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **IntersectClipRect**

ExtSelectClipRgn

The **ExtSelectClipRgn** function combines the specified region with the current clipping region by using the specified mode.

```
int ExtSelectClipRgn(
    HDC hdc,          // handle to DC
    HRGN hrgn,        // handle to region
    int fnMode        // region-selection mode
);
```

Parameters

hdc
[in] Handle to the device context.

hrgn
[in] Handle to the region to be selected. This handle can only be NULL when the RGN_COPY mode is specified.

fnMode
[in] Specifies the operation to be performed. It must be one of the following values:

Value	Meaning
RGN_AND	The new clipping region combines the overlapping areas of the current clipping region and the region identified by *hrgn*.
RGN_COPY	The new clipping region is a copy of the region identified by *hrgn*. This is identical to **SelectClipRgn**. If the region identified by *hrgn* is NULL, the new clipping region is the default clipping region (the default clipping region is a null region).
RGN_DIFF	The new clipping region combines the areas of the current clipping region with those areas excluded from the region identified by *hrgn*.
RGN_OR	The new clipping region combines the current clipping region and the region identified by *hrgn*.
RGN_XOR	The new clipping region combines the current clipping region and the region identified by *hrgn* but excludes any overlapping areas.

Return Values

The return value specifies the new clipping region's complexity; it can be one of the following values:

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred.

Remarks

If an error occurs when this function is called, the previous clipping region for the specified device context is not affected.

The **ExtSelectClipRgn** function assumes that the coordinates for the specified region are specified in device units.

Only a copy of the region identified by the *hrgn* parameter is used. The region itself can be reused after this call or it can be deleted.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Clipping Overview, Clipping Functions, **SelectClipRgn**

GetClipBox

The **GetClipBox** function retrieves the dimensions of the tightest bounding rectangle that can be drawn around the current visible area on the device. The visible area is defined by the current clipping region or clip path, as well as any overlapping windows.

```
int GetClipBox(
  HDC hdc,      // handle to DC
  LPRECT lprc   // rectangle
);
```

Parameters

hdc

[in] Handle to the device context.

lprc

[out] Pointer to a **RECT** structure that is to receive the rectangle dimensions.

Return Values

If the function succeeds, the return value specifies the clipping box's complexity and can be one of the following values:

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred.

GetClipBox returns logical coordinates based on the given device context.

Windows NT/ 2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **RECT**

GetClipRgn

The **GetClipRgn** function retrieves a handle identifying the current application-defined clipping region for the specified device context.

```
int GetClipRgn(
   HDC hdc,          // handle to DC
   HRGN hrgn         // handle to region
);
```

Parameters

hdc
   [in] Handle to the device context.

hrgn
   [in] Handle to an existing region before the function is called. After the function returns, this parameter is a handle to a copy of the current clipping region.

Return Values

If the function succeeds and there is no clipping region for the given device context, the return value is zero. If the function succeeds and there is a clipping region for the given device context, the return value is 1. If an error occurs, the return value is –1.

Windows NT/ 2000: To get extended error information, call **GetLastError**.

Remarks

An application-defined clipping region is a clipping region identified by the **SelectClipRgn** function. It is not a clipping region created when the application calls the **BeginPaint** function.

If the function succeeds, the *hrgn* parameter is a handle to a copy of the current clipping region. Subsequent changes to this copy will not affect the current clipping region.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Clipping Overview, Clipping Functions, **BeginPaint**, **SelectClipRgn**

GetMetaRgn

The **GetMetaRgn** function retrieves the current metaregion for the specified device context.

```
int GetMetaRgn(
  HDC hdc,     // handle to DC
  HRGN hrgn    // handle to region
);
```

Parameters

hdc

[in] Handle to the device context.

hrgn

[in] Handle to an existing region before the function is called. After the function returns, this parameter is a handle to a copy of the current metaregion.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If the function succeeds, *hrgn* is a handle to a copy of the current metaregion. Subsequent changes to this copy will not affect the current metaregion.

The current clipping region of a device context is defined by the intersection of its clipping region and its metaregion.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **SetMetaRgn**

GetRandomRgn

The **GetRandomRgn** function copies the system clipping region of a specified device context to a specific region.

```
int GetRandomRgn(
    HDC  hdc,     // handle to DC
    HRGN hrgn,    // handle to region
    INT  iNum     // must be SYSRGN
);
```

Parameters

hdc

[in] Handle to the device context.

hrgn

[in] Handle to a region. Before the function is called, this identifies an existing region. After the function returns, this identifies a copy of the current system region. The old region identified by *hrgn* is overwritten.

iNum

[in] This parameter must be SYSRGN.

Return Values

If the function succeeds, the return value is 1. If the function fails, the return value is –1. If the region to be retrieved is NULL, the return value is 0.

Remarks

When using the SYSRGN flag, note that the system clipping region might not be current because of window movements. Nonetheless, it is safe to retrieve and use the system clipping region within the **BeginPaint/EndPaint** bracket during **WM_PAINT** processing. In this case, the system region is the intersection of the update region and the current visible area of the window. Any window movement following the return of **GetRandomRgn** and before **EndPaint** will result in a new **WM_PAINT** message. Any other use of the SYSRGN flag may result in painting errors in your application.

In Windows NT/2000, the region returned is in screen coordinates. In Windows 95/98, the region returned is in window coordinates.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **BeginPaint**, **EndPaint**, **ExtSelectClipRgn**, **GetClipRgn**, **GetClipBox**, **GetRegionData**, **OffsetRgn**

IntersectClipRect

The **IntersectClipRect** function creates a new clipping region from the intersection of the current clipping region and the specified rectangle.

```
int IntersectClipRect(
  HDC hdc,           // handle to DC
  int nLeftRect,     // x-coord of upper-left corner
  int nTopRect,      // y-coord of upper-left corner
  int nRightRect,    // x-coord of lower-right corner
  int nBottomRect    // y-coord of lower-right corner
);
```

Parameters

hdc
[in] Handle to the device context.

nLeftRect
[in] Specifies the logical x-coordinate of the upper-left corner of the rectangle.

nTopRect
[in] Specifies the logical y-coordinate of the upper-left corner of the rectangle.

nRightRect
[in] Specifies the logical x-coordinate of the lower-right corner of the rectangle.

nBottomRect
[in] Specifies the logical y-coordinate of the lower-right corner of the rectangle.

Return Values

The return value specifies the new clipping region's type and can be one of the following values.

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred. (The current clipping region is unaffected.)

Remarks

The lower and rightmost edges of the given rectangle are excluded from the clipping region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **ExcludeClipRect**

OffsetClipRgn

The **OffsetClipRgn** function moves the clipping region of a device context by the specified offsets.

```
int OffsetClipRgn(
  HDC hdc,         // handle to DC
  int nXOffset,    // offset along x-axis
  int nYOffset     // offset along y-axis
);
```

Parameters

hdc
  [in] Handle to the device context.

nXOffset
  [in] Specifies the number of logical units to move left or right.

nYOffset
  [in] Specifies the number of logical units to move up or down.

Return Values

The return value specifies the new region's complexity and can be one of the following values.

Value	**Meaning**
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred. (The current clipping region is unaffected.)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **SelectClipRgn**

PtVisible

The **PtVisible** function indicates whether the specified point is within the clipping region of a device context.

```
BOOL PtVisible(
    HDC hdc,    // handle to DC
    int X,      // x-coordinate of point
    int Y       // y-coordinate of point
);
```

Parameters

hdc
[in] Handle to the device context.

X
[in] Specifies the logical x-coordinate of the point.

Y
[in] Specifies the logical y-coordinate of the point.

Return Values

If the specified point is within the clipping region of the device context, the return value is nonzero.

If the specified point is not within the clipping region of the device context, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **RectVisible**

RectVisible

The **RectVisible** function determines whether any part of the specified rectangle lies within the clipping region of a device context.

```
BOOL RectVisible(
  HDC hdc,           // handle to DC
  CONST RECT *lprc   // rectangle
);
```

Parameters

hdc

[in] Handle to the device context.

lprc

[in] Pointer to a **RECT** structure that contains the logical coordinates of the specified rectangle.

Return Values

If some portion of the specified rectangle lies within the clipping region, the return value is nonzero.

If no portion of the specified rectangle lies within the clipping region, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **CreateRectRgn**, **PtVisible**, **RECT**, **SelectClipRgn**

SelectClipPath

The **SelectClipPath** function selects the current path as a clipping region for a device context, combining the new region with any existing clipping region by using the specified mode.

```
BOOL SelectClipPath(
  HDC hdc,      // handle to DC
  int iMode     // clipping mode
);
```

Parameters

hdc

[in] Handle to the device context of the path.

iMode

[in] Specifies the way to use the path. This parameter can be one of the following values.

Value	Meaning
RGN_AND	The new clipping region includes the intersection (overlapping areas) of the current clipping region and the current path.
RGN_COPY	The new clipping region is the current path.
RGN_DIFF	The new clipping region includes the areas of the current clipping region with those of the current path excluded.
RGN_OR	The new clipping region includes the union (combined areas) of the current clipping region and the current path.
RGN_XOR	The new clipping region includes the union of the current clipping region and the current path but without the overlapping areas.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/Windows 2000: To get extended error Information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Remarks

The device context identified by the *hdc* parameter must contain a closed path.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **BeginPath**, **EndPath**

SelectClipRgn

The **SelectClipRgn** function selects a region as the current clipping region for the specified device context.

```
int SelectClipRgn(
  HDC hdc,     // handle to DC
  HRGN hrgn    // handle to region
);
```

Parameters

hdc
[in] Handle to the device context.

hrgn
[in] Handle to the region to be selected.

Return Values

The return value specifies the region's complexity and can be one of the following values.

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred. (The previous clipping region is unaffected.)

Windows NT/Windows 2000: To get extended error Information, call **GetLastError**.

Remarks

Only a copy of the selected region is used. The region itself can be selected for any number of other device contexts or it can be deleted.

The **SelectClipRgn** function assumes that the coordinates for a region are specified in device units.

To remove a device-context's clipping region, specify a NULL region handle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Clipping Overview, Clipping Functions, **ExtSelectClipRgn**

SetMetaRgn

The **SetMetaRgn** function intersects the current clipping region for the specified device context with the current metaregion and saves the combined region as the new metaregion for the specified device context. The clipping region is reset to a null region.

```
int SetMetaRgn(
    HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

The return value specifies the new clipping region's complexity and can be one of the following values.

Value	Meaning
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred. (The previous clipping region is unaffected.)

Remarks

The current clipping region of a device context is defined by the intersection of its clipping region and its metaregion.

The **SetMetaRgn** function should only be called after an application's original device context was saved by calling the **SaveDC** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Clipping Overview, Clipping Functions, **GetMetaRgn**, **SaveDC**

CHAPTER 9

Colors

Color is an important element in the pictures and images generated by Win32-based applications. This overview describes how Win32-based applications can manage and use colors with pens, brushes, text, or bitmaps.

About Colors

Color can be used to communicate ideas, show relationships between items, and improve the appeal and quality of output. The Win32 API enables applications to discover the color capabilities of given devices and to choose from the available colors those that best suit their needs.

Although not described in this overview, image color matching is an important feature of color management that helps ensure that color images look the same whether displayed on screen or printed on paper. For more information, see *About Image Color Management Version 2.0*.

Color Basics

The color capabilities of devices, such as displays and printers, can range from monochrome to thousands of colors. Because an application might need to generate output for devices throughout this range, it should be prepared to handle varying color capabilities.

An application can discover the number of colors available for a given device by using the **GetDeviceCaps** function to retrieve the NUMCOLORS value. This value specifies the count of colors available for use by the application. Usually, this count corresponds to a physical property of the output device, such as the number of inks in the printer or the number of distinct color signals the display adapter can transmit to the monitor.

Although the NUMCOLORS value specifies the count of colors, it does not identify what the available colors are. An application can discover what colors are available by enumerating all pens having the PS_SOLID type. Because the device driver that supports a given device usually has a full range of solid pens, and because the system requires that solid pens have only colors that the device can generate, enumerating these pens is often equivalent to enumerating the colors. An application can enumerate the pens by using the **EnumObjects** function.

Color Values

Color is defined as a combination of three primary colors—red, green, and blue. The system identifies a color by giving it a color value (sometimes called an RGB triplet), which consists of three 8-bit values specifying the intensities of its color components. Black has the minimum intensity for red, green, and blue, so the color value for black is (0, 0, 0). White has the maximum intensity for red, green, and blue, so its color value is (255, 255, 255).

Note If image color matching is enabled, the definition of color and the meaning of a color value depends on the type of color space that is set currently for the device context.

The system and applications use parameters and variables having the **COLORREF** type to pass and store color values. For example, the **EnumObjects** function identifies the color of each pen by setting the **lopnColor** member in a **LOGPEN** structure to a color value. Applications can extract the individual values of the red, green, and blue components from a color value by using the **GetRValue**, **GetGValue**, and **GetBValue** macros, respectively. Applications can create a color value from individual component values by using the **RGB** macro. When creating or examining a logical palette, an application uses the **RGBQUAD** structure to define color values and to examine individual component values.

Color Approximations and Dithering

Although an application can use color without regard to the color capabilities of the device, the resulting output might not be as informative and pleasing as output for which color is chosen carefully. Few, if any, devices guarantee an exact match for every possible color value; therefore, if an application requests a color that the device cannot generate, the system approximates that color by using a color that the device can generate. For example, if an application attempts to create a red pen for a black and white printer, it will receive a black pen instead—the system uses black as the approximation for red.

An application can discover whether the system will approximate a given color by using the **GetNearestColor** function. The function takes a color value and returns the color value of the closest matching color the device can generate. The method the system uses to determine this approximation depends on the device driver and its color capabilities. In most cases, the approximated color's overall intensity is closest to that of the requested color.

When an application creates a pen or sets the color for text, the system always approximates a color if no exact match exists. When an application creates a solid brush, the system may attempt to simulate the requested color by dithering. *Dithering* simulates a color by alternating two or more colors in a pattern. For example, different shades of pink can be simulated by alternating different combinations of red and white. Depending on the colors and the pattern, dithering can produce reasonable simulations. It is most useful for monochrome devices, because it expands the number of available "colors" well beyond black and white.

The method used to create dithered colors depends on the device driver. Most device drivers use a standard dithering algorithm, which generates a pattern based on the intensity values of the requested red, green, and blue colors. In general, any requested color that cannot be generated by the device is subject to simulation, but an application is not notified when the system simulates a color. Furthermore, an application cannot modify or change the dithering algorithm of the device driver. An application, however, can bypass the algorithm by creating and using pattern brushes. In this way, the application creates its own dithered colors by combining solid colors in the bitmap that it uses to create the brush.

Color in Bitmaps

The system handles colors in bitmaps differently than colors in pens, brushes, and text. Compatible bitmaps, created by using the **CreateBitmap** or **CreateCompatibleBitmap** function, are device-specific and retain color information in a device-dependent format. No color values are used, and the colors are not subject to approximations and dithering.

Device-independent bitmaps (DIBs) retain color information either as color values or color palette indexes. If color values are used, the colors are subject to approximation, but not dithering. Color palette indexes can only be used with devices that support color palettes. Although the system does not approximate or dither colors identified by indexes, the resulting color may be different than that intended, because the indexes yield valid results only in the context of the color palette that was current at the time the bitmap was created. If the palette changes, so do the colors in the bitmap.

Color Mixing

Color mixing lets an application create new colors by combining the pen or brush color with colors in the existing image. The application can choose either to draw the pen or brush color as is (effectively drawing over any existing image) or to mix the color with the colors already present.

The foreground mix mode, sometimes called the binary raster operation, determines how these colors are mixed. An application can merge colors, preserving all components of both colors; mask colors, removing or moderating components that are not common; or exclusively mask colors, removing or moderating components that are common. There are several variations on these basic mixing operations.

Color mixing is subject to color approximation. If the result of color mixing is a color that the device cannot generate, the system approximates the result, using a color it can generate. If an application mixes dithered colors, the individual colors used to create the dithered color are mixed, and the results are subject to color approximation.

An application sets the foreground mix mode by using the **SetROP2** function and retrieves the current mode by using the **GetROP2** function.

Although there is a background mix mode, that mode does not control the mixing of colors. Instead, it specifies whether a background color is used when drawing styled lines, hatched brushes, and text.

Color Palettes

A color palette is an array that contains color values identifying the colors that can currently be displayed or drawn on the output device. Color palettes are used by devices that are capable of generating many colors but that can only display or draw a subset of these at any given time. For such devices, the system maintains a *system palette* to track and manage the current colors of the device. Applications do not have direct access to the system palette. Instead, the system associates a default palette with each device context. Applications can use the colors in the default palette or define their own colors by creating *logical palettes* and associating them with individual device contexts.

An application can determine whether a device supports color palettes by checking for the RC_PALETTE bit in the RASTERCAPS value returned by the **GetDeviceCaps** function.

Default Palette

The *default palette* is an array of color values identifying the colors that can be used with a device context by default. The system associates the default palette with a context whenever an application creates a context for a device that supports color palettes. The default palette ensures that colors are available for use by an application without any further action.

The default palette typically has 20 entries (colors), but the exact number of entries may vary from device to device. This number is equal to the NUMCOLORS value returned by the **GetDeviceCaps** function. An application can retrieve the color values for colors in the default palette by enumerating solid pens, the same technique used to discover the colors available on nonpalette devices. The colors in the default palette depend on the device. Display devices, for example, often use the 16 standard colors of the VGA display and 4 other colors defined by the Win32 API. Print devices can use other default colors.

When using the default palette, applications use color values to specify pen and text colors. If the requested color is not in the palette, the system approximates the color by using the closest color in the palette. If an application requests a solid brush color that is not in the palette, the system simulates the color by dithering with colors that are in the palette.

To avoid approximations and dithering, applications can specify also pen, brush, and text colors by using color palette indexes rather than color values. A color palette index is an integer value that identifies a specific palette entry. Applications can use color palette indexes in place of color values but must use the **PALETTEINDEX** macro to create the indexes.

Color palette indexes are only useful for devices that support color palettes. To avoid this device dependence, applications that use the same code to draw to both palette and nonpalette devices should use palette-relative color values to specify pen, brush, and text colors. These values are identical to color values except when creating solid brushes. (On palette devices, a solid brush color specified by a palette-relative color value is subject to color approximation instead of dithering.) Applications must use the **PALETTERGB** macro to create palette-relative color values.

The system does not allow an application to change the entries in the default palette. To use colors other than those in the default palette, an application must create its own logical palette and select the palette into the device context.

Logical Palette

A *logical palette* is a color palette that an application creates and associates with a given device context. Logical palettes let applications define and use colors that meet their specific needs. Applications can create any number of logical palettes, using them for separate device contexts or switching between them for a single device context. The maximum number of palettes that an application can create depends on the resources of the system.

An application creates a logical palette by using the **CreatePalette** function. The application fills a **LOGPALETTE** structure, which specifies the number of entries and the color values for each entry, and then the application passes the structure to **CreatePalette**. The function returns a palette handle that the application uses in all subsequent operations to identify the palette. To use colors in the logical palette, the application selects the palette into a device context by using the **SelectPalette** function and then realizes the palette by using the **RealizePalette** function. The colors in the palette are available as soon as the logical palette is realized.

An application should limit the size of its logical palettes to just enough entries to represent the colors needed. Applications cannot create logical palettes larger than the maximum palette size, a device-dependent value. Applications can obtain the maximum size by using the **GetDeviceCaps** function to retrieve the SIZEPALETTE value.

Although an application can specify any color value for a given entry in a logical palette, not all colors can be generated by the given device. The system does not provide a way to discover which colors are supported, but the application can discover the total number of these colors by retrieving the color resolution of the device. The color resolution, specified in color bits per pixel, is equal to the COLORRES value returned by the **GetDeviceCaps** function. A device that has a color resolution of 18 has 262,144 possible colors. If an application requests a color that is not supported, the system chooses an appropriate approximation.

Once a logical palette is created, an application can change colors in the palette by using the **SetPaletteEntries** function. If the logical palette has been selected and realized, changing the palette does not affect immediately the colors being displayed. The application must use the **UnrealizeObject** and **RealizePalette** functions to update the colors. In some cases, the application might need to deselect, unrealize, select, and realize the logical palette to ensure that the colors are updated exactly as requested. If an application selects a logical palette into more than one device context, changes to the logical palette affect all device contexts for which it is selected.

An application can change the number of entries in a logical palette by using the **ResizePalette** function. If the application reduces the size, the remaining entries are unchanged. If the application extends the size, the system sets the color for each new entry to black (0, 0, 0) and the flag to zero.

An application can retrieve the color and flag values for entries in a given logical palette by using the **GetPaletteEntries** function. An application can retrieve the index for the entry in a given logical palette that most closely matches a specified color value by using the **GetNearestPaletteIndex** function.

When an application no longer needs a logical palette, it can delete it by using the **DeleteObject** function. The application must make sure the logical palette is no longer selected into a device context before deleting the palette.

Palette Animation

Palette animation is a technique to simulate motion by rapidly changing the colors of selected entries in a color palette. An application can carry out palette animation by creating a logical palette that contains "reserved" entries and then using the **AnimatePalette** function to change colors in those reserved entries.

An application creates a reserved entry in a logical palette by setting the **peFlags** member of the **PALETTEENTRY** structure to the PC_RESERVED flag. Once this logical palette is selected and realized, the application can call the **AnimatePalette** function to change one or more reserved entries. If the given palette is associated with the active window, the system updates the colors on the screen immediately.

System Palette

The system maintains a *system palette* for each device that uses palettes. The system palette contains the color values for all colors that currently can be displayed or drawn by the device. Other than viewing the contents of the system palette, applications cannot access the system palette directly. Instead, the system has complete control of the system palette and permits access only through the use of logical palettes.

An application can view the contents of the system palette by using the **GetSystemPaletteEntries** function. This function retrieves the contents of one or more entries, up to the total number of entries in the system palette. The total is always equal to the number returned for the SIZEPALETTE value by the **GetDeviceCaps** function, and is the same as the maximum size for any given logical palette.

Although applications cannot change colors in the system palette directly, they can cause changes when realizing logical palettes. To realize a palette, the system examines each requested color and attempts to find an entry in the system palette that contains an exact match. If the system finds a matching color, it maps the logical palette index to the corresponding system palette index. If the system does not find an exact match, it copies the requested color to an unused system palette entry before mapping the indexes. If all system palette entries are in use, the system maps the logical palette index to the system palette entry whose color most closely matches the requested color. Once this mapping is set, applications cannot override it. For example, applications cannot use system palette indexes to specify colors; only logical palette indexes are permitted.

Applications can modify the way indexes are mapped by setting the **peFlags** member of the **PALETTEENTRY** structure to selected values when creating the logical palette. For example, the PC_NOCOLLAPSE flag directs the system to immediately copy the requested color to an unused system palette entry regardless of whether a system palette entry already contains that color. Also, the PC_EXPLICIT flag directs the system to map the logical palette index to an explicitly given system palette index. (The application gives the system palette index in the low-order word of the **PALETTEENTRY** structure.)

Palettes can be realized as either a background palette or a foreground palette by specifying TRUE or FALSE, respectively, for the *bForceBackground* parameter in the **SelectPalette** function. There can be only one foreground palette in the system at a time. If the window is the currently active window or a descendent of the currently active window, it can realize a foreground palette. Otherwise the palette is realized as a background palette regardless of the value of the *bForceBackground* parameter. The critical property of a foreground palette is that, when realized, it can overwrite all entries (except for the static entries) in the system palette. The system accomplishes this by marking all of the entries that are not static in the system palette as unused before the realization of a foreground palette, thereby eliminating all of the used entries. No preprocessing occurs on the system palette for a background palette realization. The foreground palette sets all of the possible nonstatic colors. Background palettes can set only what remains open, and are prioritized in a first-come, first-serve manner. Typically, applications use background palettes for child windows that realize their own individual palettes. This helps minimize the number of changes that occur to the system palette.

An unused system palette entry is any entry that is not reserved and does not contain a static color. Reserved entries are marked explicitly with the PC_RESERVED value. These entries are created when an application realizes a logical palette for palette animation. Static-color entries are created by the system and correspond to the colors in the default palette. The **GetDeviceCaps** function can be used to retrieve the NUMRESERVED value, which specifies the number of system palette entries reserved for static colors.

Because the system palette has a limited number of entries, selecting and realizing a logical palette for a given device might affect the colors associated with other logical palettes for the same device. These color changes are especially dramatic when they occur on the display. An application can make sure that reasonable colors are used for its currently selected logical palette by resetting the palette before each use. An application resets the palette by calling the **UnrealizeObject** and **RealizePalette** functions. Using these functions causes the system to remap the colors in the logical palette to reasonable colors in the system palette.

System Palette and Static Colors

Ordinarily, the system palette entries that the system reserves for static colors cannot be changed. An application can override this default behavior by using the **SetSystemPaletteUse** function to reduce the number of static-color entries and, thereby, increase the number of unused system palette entries. However, because changing the static colors can have an immediate and dramatic effect on all windows on the display, an application should not call **SetSystemPaletteUse**, unless it has a maximized window and the input focus.

When an application calls **SetSystemPaletteUse** with the SYSPAL_NOSTATIC value, the system frees all but two of the reserved entries, allowing those entries to receive new color values when the application subsequently realizes its logical palette. The two remaining static-color entries remain reserved and are set to white and black. An application can restore the reserved entries by calling **SetSystemPaletteUse** with the SYSPAL_STATIC value. It can discover the current system palette usage by using the **GetSystemPaletteUse** function.

Furthermore, after setting the system palette usage to SYSPAL_NOSTATIC, the application must realize immediately its logical palette, call the **GetSysColor** function to save the current system color settings, call the **SetSysColors** function to set the system colors to reasonable values using black and white, and finally send the **WM_SYSCOLORCHANGE** message to other top-level windows to allow them to be redrawn with the new system colors. When setting system colors using black and white, the application should make sure adjacent or overlapping items, such as window frames and borders, are set to black and white, respectively.

Before the application loses the input focus, closes its window, or terminates, it must immediately call **SetSystemPaletteUse** with the SYSPAL_STATIC value, realize its logical palette, restore the system colors to their previous values, and send the **WM_SYSCOLORCHANGE** message. The system sends a **WM_PAINT** message to any window that is affected by a system color change. Applications that have brushes using the existing system colors should delete those brushes and recreate them using the new system colors.

Palette Messages

Changes to the system palette for the display device can have dramatic and sometimes undesirable effects on the colors used in windows on the desktop. To minimize the impact of these changes, the system provides a set of messages that help applications manage their logical palettes while ensuring that colors in the active window are as close as possible to the colors intended.

The system sends a **WM_QUERYNEWPALETTE** message to a top-level or overlapped window just before activating the window. This message gives an application the opportunity to select and realize its logical palette so that it receives the best possible mapping of colors for its logical palette. When the application receives the message, it should use the **SelectPalette**, **UnrealizeObject**, and **RealizePalette** functions to select and realize the logical palette. Doing so directs the system to update colors in the system palette so that its colors match as many colors in the logical palette as possible.

When an application causes changes to the system palette as a result of realizing its logical palette, the system sends a **WM_PALETTECHANGED** message to all top-level and overlapped windows. This message gives applications the opportunity to update the colors in the client areas of their windows, replacing colors that have changed with colors that more closely match the intended colors. An application that receives the **WM_PALETTECHANGED** message should use **UnrealizeObject** and **RealizePalette** to reset the logical palettes associated with all inactive windows, and then update the colors in the client area for each inactive window by using the **UpdateColors** function. This technique does not guarantee the greatest number of exact color matches; however, it does ensure that colors in the logical palette are mapped to reasonable colors in the system palette.

Note To avoid creating an infinite loop, an application should *never* realize the palette for the window whose handle matches the handle passed in the *wParam* parameter of the **WM_PALETTECHANGED** message.

The **UpdateColors** function typically updates a client area of an inactive window faster than redrawing the area. However, because **UpdateColors** performs color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy. This means **UpdateColors** cannot be used to update colors when the window becomes active. In such cases, the application should redraw the client area.

The system can send the **WM_QUERYNEWPALETTE** message when changes to the logical palette are made. Also, the system can send the **WM_PALETTEISCHANGING** message to all top-level and overlapped windows when the system palette is about to change.

Halftone Palette and Color Adjustment

Halftone palettes are intended to be used whenever the stretching mode of a device context is set to HALFTONE. An application creates a halftone palette by using the **CreateHalftonePalette** function. The application must select and realize this palette into the device context before calling the **StretchBlt** or **StretchDIBits** function.

The system automatically adjusts the input color of source bitmaps whenever applications call the **StretchBlt** and **StretchDIBits** functions and the stretching mode of a device context is set to HALFTONE. These color adjustments affect certain attributes of the image, such as contrast and brightness. An application can set the color adjustment values by using the **SetColorAdjustment** function. The application can retrieve the color adjustment values for the specified device context by using the **GetColorAdjustment** function. The **HTUI_ColorAdjustment** function displays the default user interface for halftone color adjustment.

Color Reference

Color Functions

AnimatePalette

The **AnimatePalette** function replaces entries in the specified logical palette.

```
BOOL AnimatePalette(
  HPALETTE hpal,             // handle to logical palette
  UINT iStartIndex,          // first entry in logical palette
  UINT cEntries,             // number of entries
  CONST PALETTEENTRY *ppe    // first replacement
);
```

Parameters

hpal

[in] Handle to the logical palette.

iStartIndex

[in] Specifies the first logical palette entry to be replaced.

cEntries

[in] Specifies the number of entries to be replaced.

ppe

[in] Pointer to the first member in an array of **PALETTEENTRY** structures used to replace the current entries.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

The **AnimatePalette** function only changes entries with the PC_RESERVED flag set in the corresponding **palPalEntry** member of the **LOGPALETTE** structure.

If the given palette is associated with the active window, the colors in the palette are replaced immediately.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **CreatePalette**, **GetDeviceCaps**, **LOGPALETTE**, **PALETTEENTRY**

CreateHalftonePalette

The **CreateHalftonePalette** function creates a halftone palette for the specified device context (DC).

```
HPALETTE CreateHalftonePalette(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

If the function succeeds, the return value is a handle to a logical halftone palette.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application should create a halftone palette when the stretching mode of a device context is set to HALFTONE. The logical halftone palette returned by **CreateHalftonePalette** should then be selected and realized into the device context before the **StretchBlt** or **StretchDIBits** function is called.

When you no longer need the palette, call the **DeleteObject** function to delete it.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **DeleteObject**, **RealizePalette**, **SelectPalette**, **SetStretchBltMode**, **StretchBlt, StretchDIBits**

CreatePalette

The **CreatePalette** function creates a logical palette.

```
HPALETTE CreatePalette(
  CONST LOGPALETTE *lplgpl   // logical palette
);
```

Parameters

lplgpl

[in] Pointer to a **LOGPALETTE** structure that contains information about the colors in the logical palette.

Return Values

If the function succeeds, the return value is a handle to a logical palette.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

Once an application creates a logical palette, it can select that palette into a device context by calling the **SelectPalette** function. A palette selected into a device context can be realized by calling the **RealizePalette** function.

When you no longer need the palette, call the **DeleteObject** function to delete it.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Colors Overview, Color Functions, **DeleteObject**, **GetDeviceCaps**, **LOGPALETTE**, **RealizePalette**, **SelectPalette**

GetColorAdjustment

The **GetColorAdjustment** function retrieves the color adjustment values for the specified device context (DC).

```
BOOL GetColorAdjustment(
  HDC hdc,                  // handle to DC
  LPCOLORADJUSTMENT lpca    // color adjustment values
);
```

Parameters

hdc
[in] Handle to the device context.

lpca
[out] Pointer to a **COLORADJUSTMENT** structure that receives the color adjustment values.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **COLORADJUSTMENT**, **SetColorAdjustment**

GetNearestColor

The **GetNearestColor** function returns a color value identifying a color from the system palette that will be displayed when the specified color value is used.

```
COLORREF GetNearestColor(
  HDC hdc,           // handle to DC
  COLORREF crColor   // color to be matched
);
```

Parameters

hdc

[in] Handle to the device context.

crColor

[in] Specifies a color value that identifies a requested color. To create a **COLORREF** color value, use the **RGB** macro.

Return Values

If the function succeeds, the return value identifies a color from the system palette that corresponds to the given color value.

If the function fails, the return value is CLR_INVALID.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **COLORREF, GetDeviceCaps**, **GetNearestPaletteIndex**, **RGB**

GetNearestPaletteIndex

The **GetNearestPaletteIndex** function retrieves the index for the entry in the specified logical palette most closely matching a specified color value.

```
UINT GetNearestPaletteIndex(
  HPALETTE hpal,      // handle to logical palette
  COLORREF crColor    // color to be matched
);
```

Parameters

hpal

[in] Handle to a logical palette.

crColor

[in] Specifies a color to be matched. To create a **COLORREF** color value, use the **RGB** macro.

Return Values

If the function succeeds, the return value is the index of an entry in a logical palette.

If the function fails, the return value is CLR_INVALID.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

If the given logical palette contains entries with the PC_EXPLICIT flag set, the return value is undefined.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **COLORREF**, **GetDeviceCaps**, **GetNearestColor**, **GetPaletteEntries**, **GetSystemPaletteEntries**, RGB

GetPaletteEntries

The **GetPaletteEntries** function retrieves a specified range of palette entries from the given logical palette.

```
UINT GetPaletteEntries(
  HPALETTE hpal,          // handle to logical palette
  UINT iStartIndex,       // first entry to retrieve
  UINT nEntries,          // number of entries to retrieve
  LPPALETTEENTRY lppe     // array that receives entries
);
```

Parameters

hpal
[in] Handle to the logical palette.

iStartIndex
[in] Specifies the first entry in the logical palette to be retrieved.

nEntries
[in] Specifies the number of entries in the logical palette to be retrieved.

lppe
[out] Pointer to an array of **PALETTEENTRY** structures to receive the palette entries. The array must contain at least as many structures as specified by the *nEntries* parameter.

Return Values

If the function succeeds and the handle to the logical palette is a valid pointer (not NULL), the return value is the number of entries retrieved from the logical palette. If the function succeeds and handle to the logical palette is NULL, the return value is the number of entries in the given palette.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

If the *nEntries* parameter specifies more entries than exist in the palette, the remaining members of the **PALETTEENTRY** structure are not altered.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **GetDeviceCaps**, **GetSystemPaletteEntries**, **PALETTEENTRY**, **SetPaletteEntries**

GetSystemPaletteEntries

The **GetSystemPaletteEntries** function retrieves a range of palette entries from the system palette that is associated with the specified device context (DC).

```
UINT GetSystemPaletteEntries(
  HDC hdc,               // handle to DC
  UINT iStartIndex,      // first entry to be retrieved
  UINT nEntries,         // number of entries to be retrieved
  LPPALETTEENTRY lppe    // array that receives entries
);
```

Parameters

hdc

[in] Handle to the device context.

iStartIndex

[in] Specifies the first entry to be retrieved from the system palette.

nEntries

[in] Specifies the number of entries to be retrieved from the system palette.

lppe

[out] Pointer to an array of **PALETTEENTRY** structures to receive the palette entries. The array must contain at least as many structures as specified by the *nEntries* parameter. If this parameter is NULL, the function returns the total number of entries in the palette.

Return Values

If the function succeeds, the return value is the number of entries retrieved from the palette.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **GetDeviceCaps**, **GetPaletteEntries**, **PALETTEENTRY**

GetSystemPaletteUse

The **GetSystemPaletteUse** function retrieves the current state of the system (physical) palette for the specified device context (DC).

```
UINT GetSystemPaletteUse(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
[in] Handle to the device context.

Return Values

If the function succeeds, the return value is the current state of the system palette. This parameter can be one of the following values:

Value	Meaning
SYSPAL_ERROR	The given device context is invalid or does not support a color palette.
SYSPAL_NOSTATIC	The system palette contains no static colors, except for black and white.
SYSPAL_STATIC	The system palette contains static colors that will not change when an application realizes its logical palette.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

By default, the system palette contains 20 static colors that are not changed when an application realizes its logical palette. An application can gain access to most of these colors by calling the **SetSystemPaletteUse** function.

The device context identified by the *hdc* parameter must represent a device that supports color palettes.

An application can determine whether or not a device supports color palettes by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **GetDeviceCaps**, **SetSystemPaletteUse**

HTUI_ColorAdjustment

The **HTUI_ColorAdjustment** function displays the default user interface for halftone color adjustment.

```
LONG HTUI_ColorAdjustment(
  LPTSTR pCallerTitle,                       // title of caller
  HANDLE hDefDIB,                            // handle to DIB
  LPTSTR pDefDIBTitle,                       // DIB picture name
  PCOLORADJUSTMENT pColorAdjustment,         // color adjustment
  BOOL ShowMonochromeOnly,                   // display type
  BOOL UpdatePermission                      // update permission
);
```

Parameters

pCallerTitle

[in] Pointer to the title of the calling application or device. The value of *pCallerTitle* will be displayed in the **Modify For** dialog box. If the value of this parameter is NULL, no title is displayed.

hDefDIB

[in] Handle to the device independent bitmap (DIB). If *hDefDIB* is not NULL, the function will use this DIB as the default picture for color adjustment testing. If *hDefDIB* is NULL, one of three standard pictures is displayed for the user to adjust preferences. The picture displayed can be:

- RGB color chart
- Reference color chart
- NTSC color chart

pDefDIBTitle

[in] Pointer to a string that specifies the DIB picture name or a description of the *hDefDIB* passed.

pColorAdjustment

[in/out] Pointer to the **COLORADJUSTMENT** data structure.

ShowMonochromeOnly

[in] Limits the display to a monochrome version of the bitmap. This setting may be used if the output device is monochrome.

UpdatePermission

[in] Update permission for the **COLORADJUSTMENT** structure. The *UpdatePermission* values are as follows:

Value	Meaning
True	Color adjustment is not limited to the current user interface. Changes to the **COLORADJUSTMENT** structure settings will be saved when exiting the halftone color adjustment user interface.
False	Color adjustment is limited to the user's current session. The **COLORADJUSTMENT** structure is not changed.

Return Values

The **HTUI_ColorAdjustment** function returns one of the following values:

Value	Meaning
> 0	The user elected to update the **COLORADJUSTMENT** structure.
= 0	The user elected to cancel the update to the **COLORADJUSTMENT** structure.
< 0	An error occurred. The value given in the error message identifies a predefined error code.

Remarks

Applications can either link to **htui.dll** or use the **LoadLibrary** and **GetProcAddress** functions to obtain the location of htui.dll.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winddi.h; include windows.h.
Library: Included as a resource in htui.dll.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Colors Overview, Color Functions, **COLORADJUSTMENT**

RealizePalette

The **RealizePalette** function maps palette entries from the current logical palette to the system palette.

```
UINT RealizePalette(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context into which a logical palette has been selected.

Return Values

If the function succeeds, the return value is the number of entries in the logical palette mapped to the system palette.

If the function fails, the return value is GDI_ERROR.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

The **RealizePalette** function modifies the palette for the device associated with the specified device context. If the device context is a memory DC, the color table for the bitmap selected into the DC is modified. If the device context is a display DC, the physical palette for that device is modified.

A logical palette is a buffer between color-intensive applications and the system, allowing these applications to use as many colors as needed without interfering with colors displayed by other windows.

When an application's window has the focus and it calls the **RealizePalette** function, the system attempts to realize as many of the requested colors as possible. The same is true also for applications with inactive windows.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Colors Overview, Color Functions, **CreatePalette**, **GetDeviceCaps**, **SelectPalette**

ResizePalette

The **ResizePalette** function increases or decreases the size of a logical palette based on the specified value.

```
BOOL ResizePalette(
  HPALETTE hpal,   // handle to logical palette
  UINT nEntries    // number of entries in logical palette
);
```

Parameters

hpal

[in] Handle to the palette to be changed.

nEntries

[in] Specifies the number of entries in the palette after it has been resized. **Windows NT/2000:** The number of entries is limited to 1024.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

If an application calls **ResizePalette** to reduce the size of the palette, the entries remaining in the resized palette are unchanged. If the application calls **ResizePalette** to enlarge the palette, the additional palette entries are set to black (the red, green, and blue values are all 0) and their flags are set to zero.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Colors Overview, Color Functions, **GetDeviceCaps**

SelectPalette

The **SelectPalette** function selects the specified logical palette into a device context.

```
HPALETTE SelectPalette(
  HDC hdc,                  // handle to DC
  HPALETTE hpal,            // handle to logical palette
  BOOL bForceBackground     // foreground or background mode
);
```

Parameters

hdc

[in] Handle to the device context.

hpal

[in] Handle to the logical palette to be selected.

bForceBackground

[in] Specifies whether the logical palette is forced to be a background palette. If this value is TRUE, the **RealizePalette** function causes the logical palette to be mapped to the colors already in the physical palette in the best possible way. This is always done, even if the window for which the palette is realized belongs to a thread without active focus.

If this value is FALSE, **RealizePalette** causes the logical palette to be copied into the device palette when the application is in the foreground. (If the *hdc* parameter is a memory device context, this parameter is ignored.)

Return Values

If the function succeeds, the return value is a handle to the device context's previous logical palette.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

An application can select a logical palette into more than one device context only if device contexts are compatible. Otherwise **SelectPalette** fails. To create a device context that is compatible with another device context, call **CreateCompatibleDC** with the first device context as the parameter. If a logical palette is selected into more than brone device context, changes to the logical palette will affect all device contexts for which it is selected.

An application might call the **SelectPalette** function with the *bForceBackground* parameter set to TRUE if the child windows of a top-level window each realize their own palettes. However, only the child window that needs to realize its palette must set *bForceBackground* to TRUE; other child windows must set this value to FALSE.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **CreateCompatibleDC**, **CreatePalette**, **GetDeviceCaps**, **RealizePalette**

SetColorAdjustment

The **SetColorAdjustment** function sets the color adjustment values for a device context (DC) using the specified values.

```
BOOL SetColorAdjustment(
  HDC hdc,                        // handle to DC
  CONST COLORADJUSTMENT *lpca     // color adjustment values
);
```

Parameters

hdc

[in] Handle to the device context.

lpca

[in] Pointer to a **COLORADJUSTMENT** structure containing the color adjustment values.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The color adjustment values are used to adjust the input color of the source bitmap for calls to the **StretchBlt** and **StretchDIBits** functions when HALFTONE mode is set.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **COLORADJUSTMENT**, **GetColorAdjustment**, **SetStretchBltMode**, **StretchBlt**, **StretchDIBits**

SetPaletteEntries

The **SetPaletteEntries** function sets RGB (red, green, blue) color values and flags in a range of entries in a logical palette.

```
UINT SetPaletteEntries(
  HPALETTE hpal,             // handle to logical palette
  UINT iStart,               // index of first entry to set
  UINT cEntries,             // number of entries to set
  CONST PALETTEENTRY *lppe   // array of palette entries
);
```

Parameters

hpal

[in] Handle to the logical palette.

iStart

[in] Specifies the first logical-palette entry to be set.

cEntries

[in] Specifies the number of logical-palette entries to be set.

lppe

[in] Pointer to the first member of an array of **PALETTEENTRY** structures containing the RGB values and flags.

Return Values

If the function succeeds, the return value is the number of entries that were set in the logical palette.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether or not a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

Even if a logical palette has been selected and realized, changes to the palette do not affect the physical palette in the surface. **RealizePalette** must be called again to set the new logical palette into the surface.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Colors Overview, Color Functions, **GetDeviceCaps**, **GetPaletteEntries**, **PALETTEENTRY**, **RealizePalette**

SetSystemPaletteUse

The **SetSystemPaletteUse** function allows an application to specify whether the system palette contains 2 or 20 static colors. The default system palette contains 20 static colors. (Static colors cannot be changed when an application realizes a logical palette.)

```
UINT SetSystemPaletteUse(
  HDC hdc,      // handle to DC
  UINT uUsage   // palette usage
);
```

Parameters

hdc

[in] Handle to the device context. This device context must refer to a device that supports color palettes.

uUsage

[in] Specifies the new use of the system palette. This parameter can be one of the following values:

Value	Meaning
SYSPAL_NOSTATIC	The system palette contains two static colors (black and white).
SYSPAL_NOSTATIC256	**Windows 2000:** The system palette contains no static colors.
SYSPAL_STATIC	The system palette contains static colors that will not change when an application realizes its logical palette.

Return Values

If the function succeeds, the return value is the previous system palette. It can be either SYSPAL_NOSTATIC, SYSPAL_NOSTATIC256, or SYSPAL_STATIC.

If the function fails, the return value is SYSPAL_ERROR.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

When an application window moves to the foreground and the SYSPAL_NOSTATIC value is set, the application must call the **GetSysColor** function to save the current system colors setting. It must also call **SetSysColors** to set reasonable values using only black and white. When the application returns to the background or terminates, the previous system colors must be restored.

If the function returns SYSPAL_ERROR, the specified device context is invalid or does not support color palettes.

An application must call this function only when its window is maximized and has the input focus.

If an application calls **SetSystemPaletteUse** with *uUsage* set to SYSPAL_NOSTATIC, the system continues to set aside two entries in the system palette for pure white and pure black, respectively.

After calling this function with *uUsage* set to SYSPAL_NOSTATIC, an application must take the following steps:

1. Realize the logical palette.
2. Call the **GetSysColor** function to save the current system-color settings.
3. Call the **SetSysColors** function to set the system colors to reasonable values using black and white. For example, adjacent or overlapping items (such as window frames and borders) should be set to black and white, respectively.
4. Send the **WM_SYSCOLORCHANGE** message to other top-level windows to allow them to be redrawn with the new system colors.

When the application's window loses focus or closes, the application must take the following steps:

1. Call **SetSystemPaletteUse** with the *uUsage* parameter set to SYSPAL_STATIC.
2. Realize the logical palette.
3. Restore the system colors to their previous values.
4. Send the **WM_SYSCOLORCHANGE** message.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Colors Overview, Color Functions, **GetDeviceCaps**, **GetSysColor**, **GetSystemPaletteUse, SetSysColors**

UnrealizeObject

The **UnrealizeObject** function resets the origin of a brush or resets a logical palette. If the *hgdiobj* parameter is a handle to a brush, **UnrealizeObject** directs the system to reset the origin of the brush the next time it is selected. If the *hgdiobj* parameter is a handle to a logical palette, **UnrealizeObject** directs the system to realize the palette as though it had not previously been realized. The next time the application calls the **RealizePalette** function for the specified palette, the system completely remaps the logical palette to the system palette.

```
BOOL UnrealizeObject(
  HGDIOBJ hgdiobj   // handle to logical palette
);
```

Parameters

hgdiobj
[in] Handle to the logical palette to be reset.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **UnrealizeObject** function should not be used with stock objects. For example, the default palette, obtained by calling **GetStockObject**(DEFAULT_PALETTE), is a stock object.

A palette identified by *hgdiobj* can be the currently selected palette of a device context.

Windows 95/98: Automatic tracking of the brush origin is not supported. Applications must use the **UnrealizeObject**, **SetBrushOrgEx**, and **SelectObject** functions to align the brush before using it.

Windows 2000: If *hgdiobj* is a brush, **UnrealizeObject** does nothing, and the function returns TRUE. Use **SetBrushOrgEx** to set the origin of a brush.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **GetStockObject**, **RealizePalette**, **SetBrushOrgEx**

UpdateColors

The **UpdateColors** function updates the client area of the specified device context by remapping the current colors in the client area to the currently realized logical palette.

```
BOOL UpdateColors(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can determine whether a device supports palette operations by calling the **GetDeviceCaps** function and specifying the RASTERCAPS constant.

An inactive window with a realized logical palette may call **UpdateColors** as an alternative to redrawing its client area when the system palette changes.

The **UpdateColors** function typically updates a client area faster than redrawing the area. However, because **UpdateColors** performs the color translation based on the color of each pixel before the system palette changed, each call to this function results in the loss of some color accuracy.

This function must be called soon after a **WM_PALETTECHANGED** message is received.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Colors Overview, Color Functions, **GetDeviceCaps**, **RealizePalette**

Color Structures

COLORREF

The **COLORREF** value is used to specify an RGB color.

```
typedef DWORD COLORREF;
typedef DWORD *LPCOLORREF;
```

Remarks

When specifying an explicit RGB color, the **COLORREF** value has the following hexadecimal form:

```
0x00bbggrr
```

The low-order byte contains a value for the relative intensity of red, the second byte contains a value for green, and the third byte contains a value for blue. The high-order byte must be zero. The maximum value for a single byte is 0xFF.

To create a **COLORREF** color value, use the **RGB** macro. To extract the individual values for the red, green, and blue components of a color value, use the **GetRValue**, **GetGValue**, and **GetBValue** macros, respectively.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in windef.h; include windows.h.

See Also

Colors Overview, Color Structures, **GetBValue, GetGValue, GetRValue, RGB**

LOGPALETTE

The **LOGPALETTE** structure defines a logical palette.

```
typedef struct tagLOGPALETTE {
  WORD         palVersion;
  WORD         palNumEntries;
  PALETTEENTRY palPalEntry[1];
} LOGPALETTE;
```

Members

palVersion
Specifies the version number of the system.

palNumEntries
Specifies the number of entries in the logical palette.

palPalEntry
Specifies an array of **PALETTEENTRY** structures that define the color and usage of each entry in the logical palette.

Remarks

The colors in the palette-entry table should appear in order of importance because entries earlier in the logical palette are most likely to be placed in the system palette.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Structures, **CreatePalette**, **PALETTEENTRY**

PALETTEENTRY

The **PALETTEENTRY** structure specifies the color and usage of an entry in a logical palette. A logical palette is defined by a **LOGPALETTE** structure.

```
typedef struct tagPALETTEENTRY {
  BYTE peRed;
  BYTE peGreen;
  BYTE peBlue;
  BYTE peFlags;
} PALETTEENTRY;
```

Members

peRed

Specifies a red intensity value for the palette entry.

peGreen

Specifies a green intensity value for the palette entry.

peBlue

Specifies a blue intensity value for the palette entry.

peFlags

Specifies how the palette entry is to be used. The **peFlags** member may be set to NULL or one of the following values:

Value	Meaning
PC_EXPLICIT	Specifies that the low-order word of the logical palette entry designates a hardware palette index. This flag allows the application to show the contents of the display device palette.
PC_NOCOLLAPSE	Specifies that the color be placed in an unused entry in the system palette instead of being matched to an existing color in the system palette. If there are no unused entries in the system palette, the color is matched normally. Once this color is in the system palette, colors in other logical palettes can be matched to this color.
PC_RESERVED	Specifies that the logical palette entry be used for palette animation. This flag prevents other windows from matching colors to the palette entry since the color frequently changes. If an unused system-palette entry is available, the color is placed in that entry. Otherwise, the color is not available for animation.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Structures, **LOGPALETTE**

Color Macros

GetBValue

The **GetBValue** macro retrieves an intensity value for the blue component of a red, green, blue (RGB) value.

```
BYTE GetBValue(
  DWORD rgb      // RGB value
);
```

Parameters

rgb
Specifies an RGB color value.

Return Values

The return value is the intensity of the blue component of the specified RGB color.

Remarks

The intensity value is in the range 0 through 255.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Macros, **GetGValue**, **GetRValue**, **PALETTEINDEX**, **PALETTERGB**, **RGB**

GetGValue

The **GetGValue** macro retrieves an intensity value for the green component of a red, green, blue (RGB) value.

```
BYTE GetGValue(
  DWORD rgb      // RGB value
);
```

Parameters

rgb
Specifies an RGB color value.

Return Values

The return value is the intensity of the green component of the specified RGB color.

Remarks

The intensity value is in the range 0 through 255.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

See Also

Colors Overview, Color Macros, **GetBValue**, **GetRValue**, **PALETTEINDEX**, **PALETTERGB**, **RGB**

GetRValue

The **GetRValue** macro retrieves an intensity value for the red component of a red, green, blue (RGB) value.

```
BYTE GetRValue(
  DWORD rgb  // RGB value
);
```

Parameters

rgb
Specifies an RGB color value.

Return Values

The return value is the intensity of the red component of the specified RGB color.

Remarks

The intensity value is in the range 0 through 255.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Macros, **GetBValue**, **GetGValue**, **PALETTEINDEX**, **PALETTERGB**, **RGB**

PALETTEINDEX

The **PALETTEINDEX** macro accepts an index to a logical-color palette entry and returns a palette-entry specifier consisting of a **COLORREF** value that specifies the color associated with the given index. An application using a logical palette can pass this specifier, instead of an explicit red, green, blue (RGB) value, to GDI functions that expect a color. This allows the function to use the color in the specified palette entry.

```
COLORREF PALETTEINDEX(
  WORD wPaletteIndex  // index to palette entry
);
```

Parameters

wPaletteIndex

Specifies an index to the palette entry containing the color to be used for a graphics operation.

Return Values

The return value is a logical-palette index specifier.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Macros, **COLORREF**, **PALETTERGB**, **RGB**

PALETTERGB

The **PALETTERGB** macro accepts three values that represent the relative intensities of red, green, and blue and returns a palette-relative red, green, blue (RGB) specifier consisting of 2 in the high-order byte and an RGB value in the three low-order bytes. An application using a color palette can pass this specifier, instead of an explicit RGB value, to functions that expect a color.

```
COLORREF PALETTERGB(
  BYTE bRed,    // red component of palette-relative RGB
  BYTE bGreen,  // green component of palette-relative RGB
  BYTE bBlue    // blue component of palette-relative RGB
);
```

Parameters

bRed

Specifies the intensity of the red color field.

bGreen

Specifies the intensity of the green color field.

bBlue

Specifies the intensity of the blue color field.

Return Values

The return value is a palette-relative RGB specifier. For output devices that support logical palettes, the system matches a palette-relative RGB value to the nearest color in the logical palette of the device context as though the application had specified an index to that palette entry. If an output device does not support a system palette, the system uses the palette-relative RGB as though it were a conventional RGB value returned by the RGB macro.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Macros, **COLORREF**, **PALETTEINDEX**, **RGB**

RGB

The **RGB** macro selects a red, green, blue (RGB) color based on the arguments supplied and the color capabilities of the output device.

```
COLORREF RGB(
  BYTE byRed,    // red component of color
  BYTE byGreen,  // green component of color
  BYTE byBlue    // blue component of color
);
```

Parameters

byRed
  Specifies the intensity of the red color.

byGreen
  Specifies the intensity of the green color.

byBlue
  Specifies the intensity of the blue color.

Return Values

The return value is the resultant RGB color as a **COLORREF** value.

Remarks

The intensity for each argument is in the range 0 through 255. If all three intensities are zero, the result is black. If all three intensities are 255, the result is white.

To extract the individual values for the red, green, and blue components of a **COLORREF** color value, use the **GetRValue, GetGValue, and GetBValue** macros, respectively.

When creating or examining a logical palette, use the **RGBQUAD** structure to define color values and examine individual component values. For more information about using color values in a color palette, see the descriptions of the **PALETTEINDEX** and **PALETTERGB** macros.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Colors Overview, Color Macros, **COLORREF, GetBValue, GetGValue, GetRValue, PALETTEINDEX**, **PALETTERGB**, **RGBQUAD**

Color Messages

WM_PALETTECHANGED

The **WM_PALETTECHANGED** message is sent to all top-level and overlapped windows after the window with the keyboard focus has realized its logical palette, thereby changing the system palette. This message enables a window that uses a color palette but does not have the keyboard focus to realize its logical palette and update its client area.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_PALETTECHANGED
  WPARAM wParam,    // handle to window (HWND)
  LPARAM lParam     // not used
);
```

Parameters

wParam
Handle to the window that caused the system palette to change.

lParam
This parameter is not used.

Remarks

This message must be sent to all top-level and overlapped windows, including the one that changed the system palette. If any child windows use a color palette, this message must be passed on to them as well.

To avoid creating an infinite loop, a window that receives this message must not realize its palette, unless it determines that *wParam* does not contain its own window handle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

+ See Also

Colors Overview, Color Messages, **WM_PALETTEISCHANGING**, **WM_QUERYNEWPALETTE**

WM_PALETTEISCHANGING

The **WM_PALETTEISCHANGING** message informs applications that an application is going to realize its logical palette.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,          // handle to window
  UINT uMsg,          // WM_PALETTEISCHANGING
  WPARAM wParam,      // handle to window (HWND)
  LPARAM lParam       // not used
);
```

Parameters

wParam

Handle to the window that is going to realize its logical palette.

lParam

This parameter is not used.

Return Values

If an application processes this message, it should return zero.

Remarks

The application changing its palette does not wait for acknowledgment of this message before changing the palette and sending the **WM_PALETTECHANGED** message. As a result, the palette might already be changed by the time an application receives this message.

If the application either ignores or fails to process this message and a second application realizes its palette while the first is using palette indexes, there is a strong possibility that the user will see unexpected colors during subsequent drawing operations.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

+ See Also

Colors Overview, Color Messages, **WM_PALETTECHANGED**, **WM_QUERYNEWPALETTE**

WM_QUERYNEWPALETTE

The **WM_QUERYNEWPALETTE** message informs a window that it is about to receive the keyboard focus, giving the window the opportunity to realize its logical palette when it receives the focus.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_QUERYNEWPALETTE
  WPARAM wParam,    // not used
  LPARAM lParam     // not used
);
```

Parameters

This message has no parameters.

Return Values

If the window realizes its logical palette, it must return TRUE; otherwise, it must return FALSE.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

+ See Also

Colors Overview, Color Messages, **WM_PALETTECHANGED**, **WM_PALETTEISCHANGING**

WM_SYSCOLORCHANGE

The **WM_SYSCOLORCHANGE** message is sent to all top-level windows when a change is made to a system color setting.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_SYSCOLORCHANGE
  WPARAM wParam,    // not used
  LPARAM lParam     // not used
);
```

Parameters

This message has no parameters.

Remarks

The system sends a **WM_PAINT** message to any window that is affected by a system color change.

Applications that have brushes using the existing system colors should delete those brushes and recreate them using the new system colors.

Top level windows that use common controls must forward the **WM_SYSCOLORCHANGE** message to the controls; otherwise, the controls will not be notified of the color change. This ensures that the colors used by your common controls are consistent with those used by other user interface objects. For example, a toolbar control uses the "3D Objects" color to draw its buttons. If the user changes the 3D Objects color, but the **WM_SYSCOLORCHANGE** message is not forwarded to the toolbar, the toolbar buttons will remain in their original color while the color of other buttons in the system changes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in winuser.h; include windows.h.

See Also

Colors Overview, Color Messages, **WM_PAINT**

CHAPTER 10

Coordinate Spaces and Transformations

Win32-based applications use coordinate spaces and transformations to scale, rotate, translate, shear, and reflect graphics output. A *coordinate space* is a planar space that locates two-dimensional objects by using two reference axes that are perpendicular to each other. There are four coordinate spaces: world, page, device, and physical device (client area, desktop, or page of printer paper).

A *transformation* is an algorithm that alters ("transforms") the size, orientation, and shape of objects. Transformations also transfer a graphics object from one coordinate space to another. Ultimately, the object appears on the physical device, which is usually a screen or printer.

About Coordinate Spaces and Transformations

Coordinate spaces and transformations are used by the following types of applications:

- Desktop publishing applications (to "zoom" parts of a page or to display adjacent pages in a window).
- Computer-aided design (CAD) applications (to rotate objects, scale drawings, or create perspective views).
- Spreadsheet applications (to move and size graphs).

The following illustrations show successive views of an object created in a drawing application. Figure 10-1 shows the object as it appears in the original drawing; Figures 10-2 through 10-6 show the effects of applying various transformations.

Transformation of Coordinate Spaces

A *coordinate space* is a planar space based on the Cartesian coordinate system. This system provides a means of specifying the location of each point on a plane. It requires two axes that are perpendicular and equal in length. Figure 10-7 shows a coordinate space.

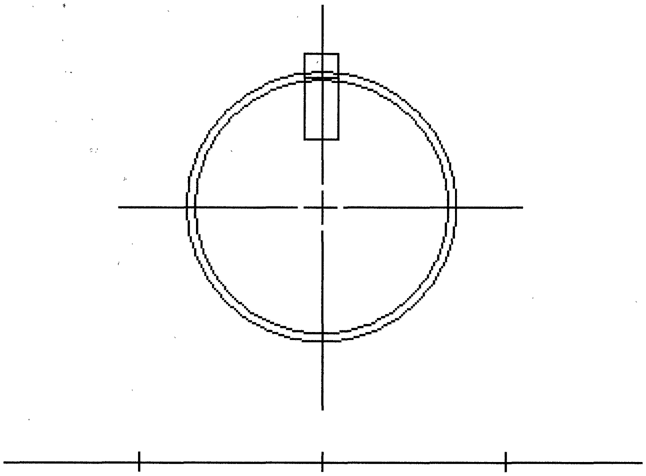


Figure 10-1: The object as it appears in the original drawing.

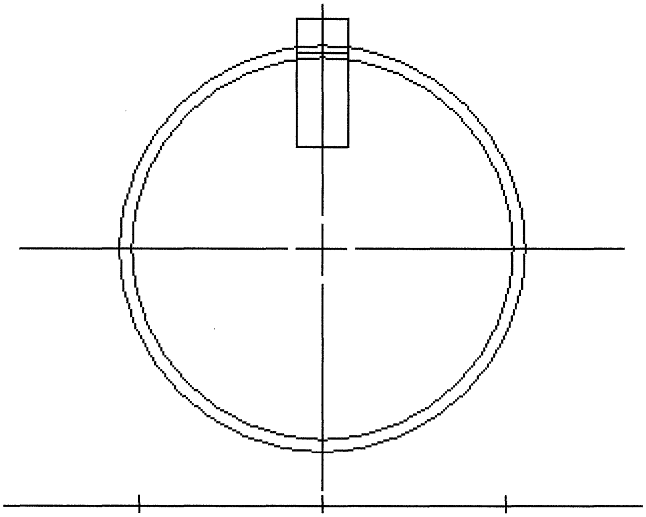


Figure 10-2.

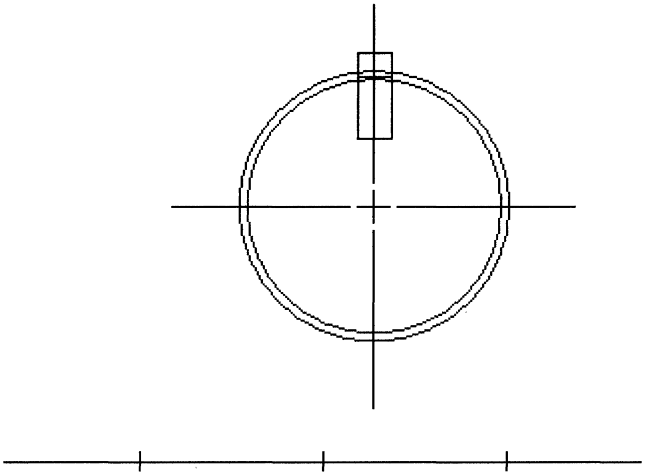


Figure 10-3.

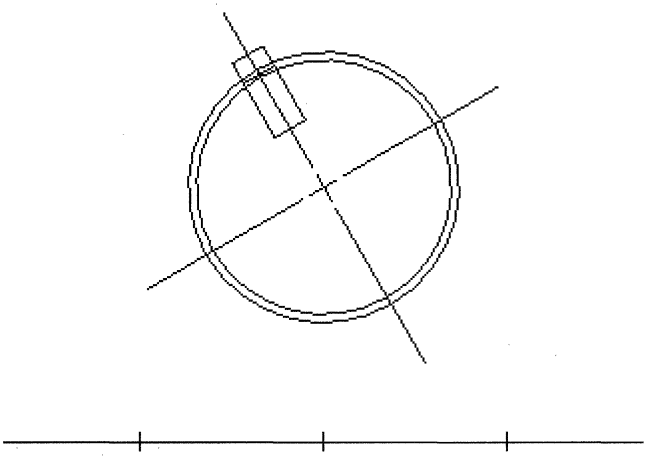


Figure 10-4.

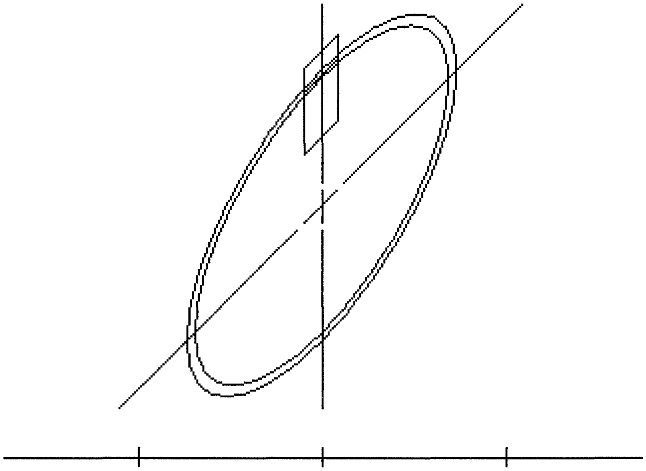


Figure 10-5.

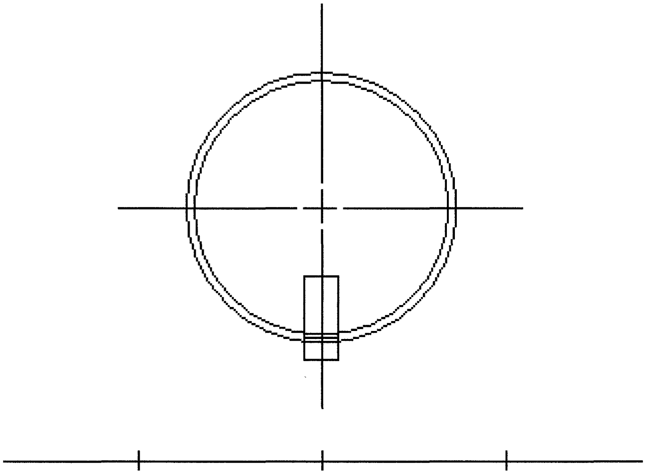


Figure 10-6.

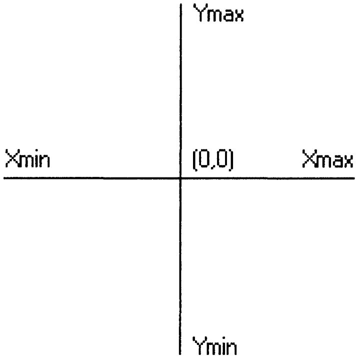


Figure 10-7: A coordinate space.

The system supports four coordinate spaces, as described in the following table:

Coordinate space	Description
world	Used optionally as the starting coordinate space for graphics transformations. It allows scaling, translation, rotation, shearing, and reflection. World space measures 2^32 units high by 2^32 units wide.
page	Used either as the next space after world space or as the starting space for graphics transformations. It sets the mapping mode. Page space (referred to as logical space in 16-bit versions of Windows) also measures 2^32 units high by 2^32 units wide.
device	Used as the next space after page space. It only allows translation, which ensures the origin of the device space maps to the proper location in physical device space. Device space measures 2^27 units high by 2^27 units wide.
physical device	The final (output) space for graphics transformations. It usually refers to the client area of the application window; however, it can also include the entire desktop, a complete window (including the frame, title bar, and menu bar), or a page of printer or plotter paper, depending on the function that obtained the handle for the device context. Physical device dimensions vary according to the dimensions set by the display, printer, or plotter technology.

Page space works with device space to provide applications with device-independent units, such as millimeters and inches. This overview refers to both world space and page space as logical space.

To depict output on a physical device, the system copies (or maps) a rectangular region from one coordinate space into the next using a transformation until the output appears in its entirety on the physical device. Mapping begins in the application's world space if the application has called the **SetWorldTransform** function; otherwise, mapping occurs in page space. As the system copies each point within the rectangular region from one space into another, it applies an algorithm called a transformation. A *transformation* alters (or transforms) the size, orientation, and shape of objects that are copied from one coordinate space into another. Although a transformation affects an object as a whole, it is applied to each point, or to each line, in the object.

Figure 10-8 shows a typical transformation performed by using the **SetWorldTransform** function.

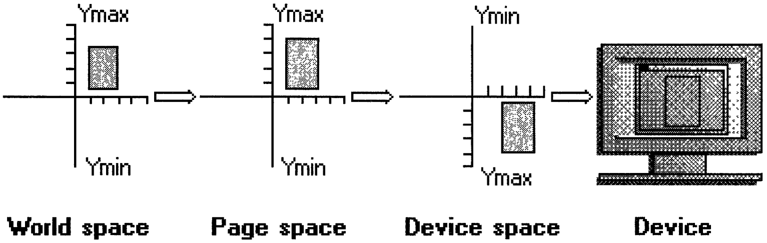


Figure 10-8: A typical transformation occurring in the application's world space.

World-Space to Page-Space Transformations

World-space to page-space transformations support translation and scaling. In addition, they support rotation, shear, and reflection capabilities. The following sections describe these transformations, illustrate their effects, and provide the algorithms used to achieve them:

- Translation
- Scaling
- Rotation
- Shear
- Reflection
- Combined World-to-Page Space Transformations

Translation

Some applications translate (or shift) objects drawn in the client area. by calling the **SetWorldTransform** function to set the appropriate world-space to page-space transformation. The **SetWorldTransform** function receives a pointer to an **XFORM** structure containing the appropriate values. The **eDx** and **eDy** members of **XFORM** specify the horizontal and vertical translation components, respectively.

When *translation* occurs, each point in an object is shifted vertically, horizontally, or both, by a specified amount. Figure 10-9 shows a 20-unit-by-20-unit rectangle that was translated to the right by 10 units when copied from world-coordinate space to page-coordinate space.

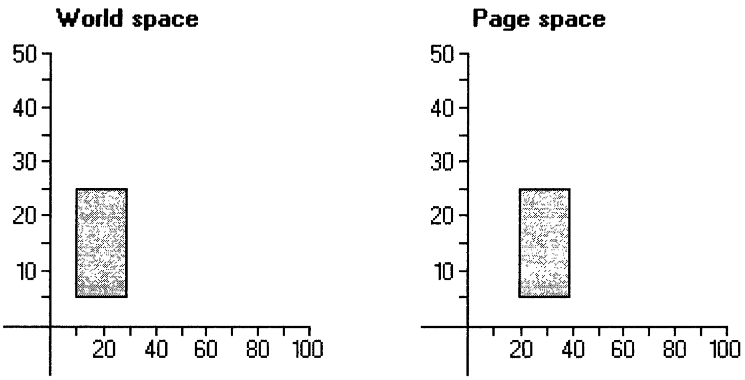


Figure 10-9: An object translated to the right of its origin.

In the preceding illustration, the x-coordinate of each point in the rectangle is 10 units greater than the original x-coordinate.

Horizontal translation can be represented by the following algorithm:

```
x' = x + Dx
```

Where *x'* is the new x-coordinate, *x* is the original x-coordinate, and *Dx* is the horizontal distance moved.

Vertical translation can be represented by the following algorithm:

```
y' = y + Dy
```

Where *y'* is the new y-coordinate, *y* is the original y-coordinate, and *Dy* is the vertical distance moved.

The horizontal and vertical translation transformations can be combined into a single operation by using a 3-by-3 matrix.

```
                   |1   0   0|
|x' y' 1| = |x y 1| * |0   1   0|
                   |Dx  Dy  1|
```

(The rules of matrix multiplication state that the number of rows in one matrix must equal the number of columns in the other. The integer 1 in the matrix |x *y* 1| is a placeholder that was added to meet this requirement.)

The 3-by-3 matrix that produced the illustrated translation transformation contains the following values:

```
|1  0  0|
|0  1  0|
|10 0  1|
```

Scaling

Most CAD and drawing applications provide features that scale output created by the user. Applications that include scaling (or zoom) capabilities call the **SetWorldTransform** function to set the appropriate world-space to page-space transformation. This function receives a pointer to an **XFORM** structure containing the appropriate values. The **eM11** and **eM22** members of **XFORM** specify the horizontal and vertical scaling components, respectively.

When *scaling* occurs, the vertical and horizontal lines (or vectors), that constitute an object, are stretched or compressed with respect to the x-axis or y-axis. Figure 10-10 shows a 20-by-20-unit rectangle scaled vertically to twice its original height when copied from world-coordinate space to page-coordinate space.

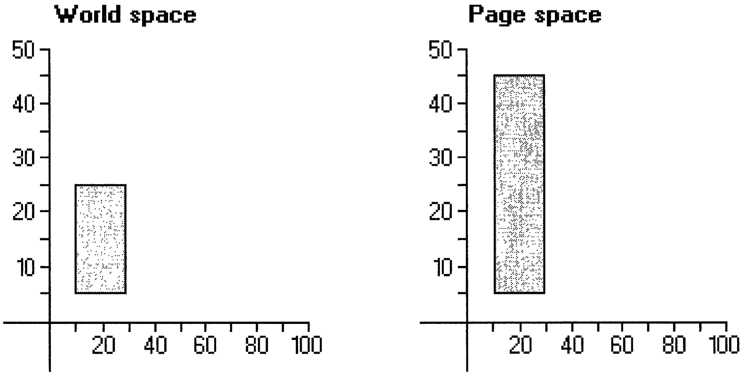


Figure 10-10: An object scaled vertically to twice its original height.

In the preceding illustration, the vertical lines that define the original rectangle's side measure 20 units, while the vertical lines that define the scaled rectangle's sides measure 40 units.

Vertical scaling can be represented by the following algorithm:

```
y' = y * Dy
```

Where *y'* is the new length, *y* is the original length, and *Dy* is the vertical scaling factor.

Horizontal scaling can be represented by the following algorithm:

```
x' = x * Dx
```

Where *x'* is the new length, *x* is the original length, and *Dx* is the horizontal scaling factor.

The vertical and horizontal scaling transformations can be combined into a single operation by using a 2-by-2 matrix.

```
|x' y'| = |Dx  0| * |x y|
          |0  Dy|
```

The 2-by-2 matrix that produced the scaling transformation contains the following values:

```
|1   0|
|0   2|
```

Rotation

Many CAD applications provide features that rotate objects drawn in the client area. Applications that include rotation capabilities use the **SetWorldTransform** function to set the appropriate world-space to page-space transformation. This function receives a pointer to an **XFORM** structure containing the appropriate values. The **eM11**, **eM12**, **eM21**, and **eM22** members of **XFORM** specify respectively, the cosine, sine, negative sine, and cosine of the angle of rotation.

When *rotation* occurs, the points that constitute an object are rotated with respect to the coordinate-space origin. Figure 10-11 shows a 20-unit-by-20-unit rectangle rotated 30 degrees when copied from world-coordinate space to page-coordinate space.

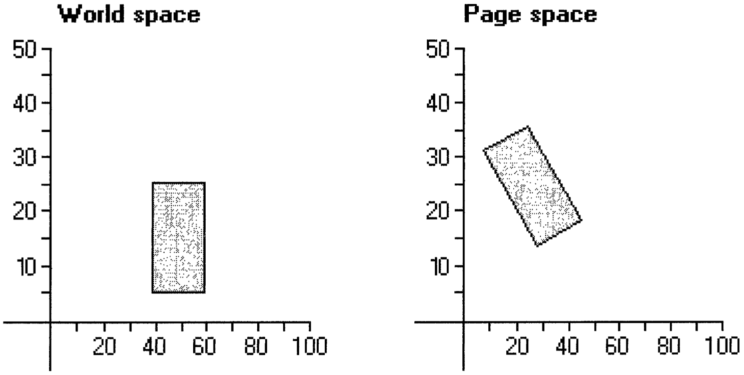


Figure 10-11: An object rotated 30 degrees from its origin.

In the preceding illustration, each point in the rectangle was rotated 30 degrees with respect to the coordinate-space origin.

The following algorithm computes the new x-coordinate (*x'*) for a point (*x*,*y*) that is rotated by angle *A* with respect to the coordinate-space origin:

```
x' = (x * cos A) - (y * sin A)
```

The following algorithm computes the y-coordinate (*y'*) for a point (*x*,*y*) that is rotated by the angle *A* with respect to the origin:

```
y' = (x * sin A) + (y * cos A)
```

The two rotation transformations can be combined in a 2-by-2 matrix as follows:

```
|x' y'| = |x y| * | cos A   sin A|
                  |-sin A   cos A|
```

The 2-by-2 matrix that produced the rotation contains the following values:

```
| .8660    .5000|
|-.5000    .8660|
```

Rotation Algorithm Derivation

Rotation algorithms are based on trigonometry's addition theorem stating that the trigonometric function of a sum of two angles (*A1* and *A2*) can be expressed in terms of the trigonometric functions of the two angles.

```
sin(A1 + A2) = (sin A1 * cos A2) + (cos A1 * sin A2)
cos(A1 + A2) = (cos A1 * cos A2) - (sin A1 * sin A2)
```

Figure 10-12 shows a point *p* rotated counterclockwise to a new position *p'*. In addition, it shows two triangles formed by a line drawn from the coordinate-space origin to each point and a line drawn from each point through the x-axis.

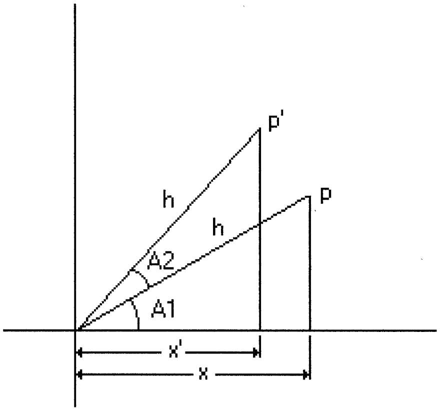


Figure 10-12: An object rotated by algorithmic derivation.

Using trigonometry, the x-coordinate of point *p* can be obtained by multiplying the length of the hypotenuse *h* by the cosine of *A1*.

```
x = h * cos A1
```

The y-coordinate of point *p* can be obtained by multiplying the length of the hypotenuse *h* by the sine of *A1*.

```
y = h * sin A1
```

Likewise, the x-coordinate of point *p'* can be obtained by multiplying the length of the hypotenuse *h* by the cosine of (*A1* + *A2*).

```
x' = h * cos (A1 + A2)
```

Finally, the y-coordinate of point *p'* can be obtained by multiplying the length of the hypotenuse *h* by the sine of (*A1* + *A2*).

```
y' = h * sin (A1 + A2)
```

Using the addition theorem, the preceding algorithms become the following:

```
x' = (h * cos A1 * cos A2) - (h * sin A1 * sin A2)
y' = (h * cos A1 * sin A2) + (h * sin A1 * cos A2)
```

The rotation algorithms for a given point rotated by angle *A2* can be obtained by substituting *x* for each occurrence of (*h* * cos *A1*) and by substituting *y* for each occurrence of (*h* * sin *A1*).

```
x' = (x * cos A2) - (y * sin A2)
y' = (x * sin A2) + (y * cos A2)
```

Shear

Some applications provide features that shear objects drawn in the client area. Applications that use shear capabilities use the **SetWorldTransform** function to set appropriate values in the world-space to page-space transformation. This function receives a pointer to an **XFORM** structure containing the appropriate values. The **eM12** and **eM21** members of **XFORM** specify the horizontal and vertical proportionality constants, respectively.

There are two components of the *shear* transformation. The first alters the vertical lines in an object; the second alters the horizontal lines. Figure 10-13 shows a 20-unit-by-20-unit rectangle sheared horizontally when copied from world space to page space.

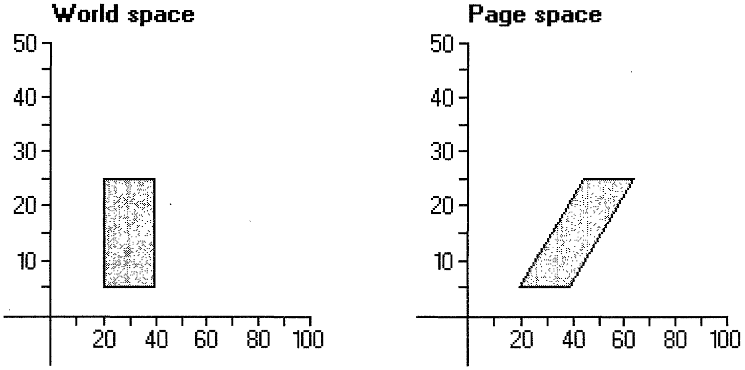


Figure 10-13: An object sheared horizontally.

A horizontal shear can be represented by the following algorithm:

```
x' = x + (Sx * y)
```

where *x* is the original x-coordinate, *Sx* is the proportionality constant, and *x'* is the result of the shear transformation.

A vertical shear can be represented by the following algorithm:

```
y' = y + (Sy * x)
```

where *y* is the original y-coordinate, *Sy* is the proportionality constant, and *y'* is the result of the shear transformation.

The horizontal-shear and vertical-shear transformations can be combined into a single operation using a 2-by-2 matrix.

```
|x' y'| == |x y| * | 1   Sx|
                   | Sy   1|
```

The 2-by-2 matrix that produced the shear contains the following values:

```
|1   1|
|0   1|
```

Reflection

Some applications provide features that reflect (or mirror) objects drawn in the client area. Applications that contain reflection capabilities use the **SetWorldTransform** function to set the appropriate values in the world-space to page-space transformation. This function receives a pointer to an **XFORM** structure containing the appropriate values. The **eM11** and **eM22** members of **XFORM** specify the horizontal and vertical reflection components, respectively.

The *reflection* transformation creates a mirror image of an object with respect to either the x-axis or y-axis. In short, reflection is just negative scaling. To produce a horizontal reflection, x-coordinates are multiplied by –1. To produce a vertical reflection, y-coordinates are multiplied by –1.

Horizontal reflection can be represented by the following algorithm:

```
x' = -x
```

where *x* is the x-coordinate and *x'* is the result of the reflection.

The 2-by-2 matrix that produced horizontal reflection contains the following values:

```
|-1   0|
|0    1|
```

Vertical reflection can be represented by the following algorithm:

```
y' = -y
```

where *y* is the y-coordinate and *y'* is the result of the reflection.

The 2-by-2 matrix that produced vertical reflection contains the following values:

```
|1    0|
|0   -1|
```

The horizontal-reflection and vertical-reflection operations can be combined into a single operation by using the following 2-by-2 matrix:

```
|-1   0|
|0   -1|
```

Combined World-to-Page Space Transformations

The five world-to-page transformations can be combined into a single 3-by-3 matrix. The **CombineTransform** function can be used to combine two world-space to page-space transformations. The combined transformations can be used to alter output associated with a particular device context (DC) by calling the **SetWorldTransform** function and supplying the elements for this matrix. When an application calls **SetWorldTransform**, it stores the elements of the 3-by-3 matrix in an **XFORM** structure. The members of this structure correspond to the first two columns of a 3-by-3 matrix; the last column of the matrix is not required because its values are constant.

The elements of the current world transformation matrix can be revived by calling the **GetWorldTransform** function and supplying a pointer to an **XFORM** structure.

Page-Space to Device-Space Transformations

The page-space to device-space transformation determines the mapping mode for all graphics output associated with a particular DC. A *mapping mode* is a scaling transformation that specifies the size of the units used for drawing operations. The mapping mode may also perform translation. In some cases, the mapping mode alters the orientation of the x-axis and y-axis in device space.

Mapping Modes and Translations

The mapping modes are described in the following table:

Mapping mode	**Description**
MM_ANISOTROPIC	Each unit in page space is mapped to an application-specified unit in device space. The axis may or may not be equally scaled (for example, a circle drawn in world space may appear to be an ellipse when depicted on a given device). The orientation of the axis is also specified by the application.
MM_HIENGLISH	Each unit in page space is mapped to 0.001 inch in device space. The value of *x* increases from left to right. The value of *y* increases from bottom to top.
MM_HIMETRIC	Each unit in page space is mapped to 0.01 millimeter in device space. The value of *x* increases from left to right. The value of *y* increases from bottom to top.

(continued)

(continued)

Mapping mode	Description
MM_ISOTROPIC	Each unit in page space is mapped to an application-defined unit in device space. The axes are always equally scaled. The orientation of the axes may be specified by the application.
MM_LOENGLISH	Each unit in page space is mapped to 0.01 inch in device space. The value of *x* increases from left to right. The value of *y* increases from bottom to top.
MM_LOMETRIC	Each unit in page space is mapped to 0.1 millimeter in device space. The value of *x* increases from left to right. The value of *y* increases from bottom to top.
MM_TEXT	Each unit in page space is mapped to one pixel; that is, no scaling is performed at all. When no translation is in effect (this is the default), page space in the MM_TEXT mapping mode is equivalent to physical device space. The value of *x* increases from left to right. The value of *y* increases from top to bottom.
MM_TWIPS	Each unit in page space is mapped to one twentieth of a printer's point (1/1440 inch). The value of *x* increases from left to right. The value of *y* increases from bottom to top.

To set a mapping mode, call the **SetMapMode** function. Retrieve the current mapping mode for a DC by calling the **GetMapMode** function.

The page-space to device-space transformations consist of values calculated from the points given by the window and viewport. In this context, the window refers to the logical coordinate system of the page space, while the viewport refers to the device coordinate system of the device space. The window and viewport each consist of an origin, a horizontal (x) extent, and a vertical (y) extent. The window parameters are in logical coordinates; the viewport in device coordinates (pixels). The system combines the origins and extents from both the window and viewport to create the transformation. This means that the window and viewport each specify half of the factors needed to define the transformation used to map points in page space to device space. Thus, the system maps the window origin to the viewport origin and the window extents to the viewport extents, as shown in Figure 10-14.

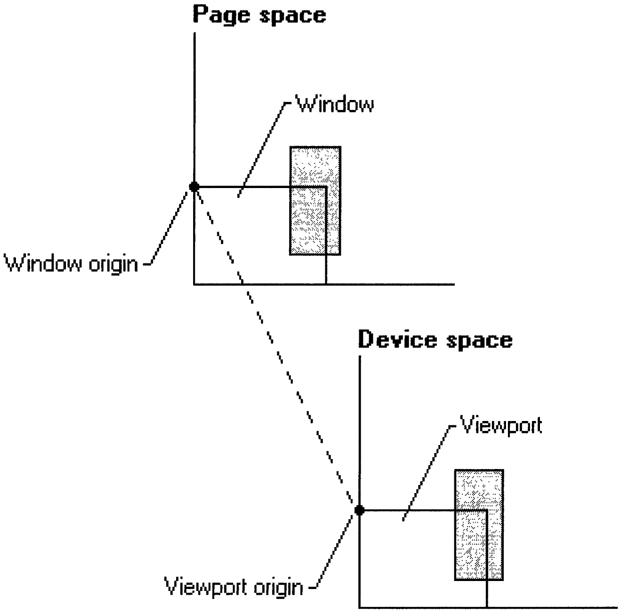


Figure 10-14: Origin and viewpoint mapping.

The window and viewport extents establish a ratio or scaling factor used in the page-space to device-space transformations. For the six predefined mapping modes (MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, MM_TEXT, and MM_TWIPS), the extents are set by the system when **SetMapMode** is called. They cannot be changed. The other two mapping modes (MM_ISOTROPIC and MM_ANISOTROPIC) require that the extents are specified. This is done by calling **SetMapMode** to set the appropriate mode and then calling the **SetWindowExtEx** and **SetViewportExtEx** functions to specify the extents. In the MM_ISOTROPIC mapping mode, it is important to call **SetWindowExtEx** before calling **SetViewportExtEx**.

The window and viewport origins establish the translation used in the page-space to device-space transformations. Set the window and viewport origins by using the **SetWindowOrgEx** and **SetViewportOrgEx** functions. The origins are independent of the extents, and an application can set them regardless of the current mapping mode. Changing a mapping mode does not affect the currently set origins (although it can affect the extents). Origins are specified in absolute units that the current mapping mode does not affect. To alter the origins, use the **OffsetWindowOrgEx** and **OffsetViewportOrgEx** functions.

The following formula shows the math involved in converting a point from page space to device space:

```
Dx = ((Lx - WOx) * VEx / WEx) + VOx
```

The following variables are involved:

```
Dx     x value in device units
Lx     x value in logical units (also known as page space
units)
WOx     window x origin
VOx     viewport x origin
WEx     window x-extent
VEx     viewport x-extent
```

The same equation with *y* replacing *x* transforms the *y* component of a point.

The formula first offsets the point from its coordinate origin. This value, no longer biased by the origin, is then scaled into the destination coordinate system by the ratio of the extents. Finally, the scaled value is offset by the destination origin to its final mapping.

The **LPtoDP** and **DPtoLP** functions may be used to convert from logical points to device points and from device points to logical points, respectively.

Predefined Mapping Modes

Of the six predefined mapping modes, one is device dependent (MM_TEXT)—the remaining five (MM_HIENGLISH, MM_LOENGLISH, MM_HIMETRIC, MM_LOMETRIC, and MM_TWIPS) are device independent.

The default mapping mode is MM_TEXT. One logical unit equals one pixel. Positive *x* is to the right, and positive *y* is down. This mode maps directly to the device's coordinate system. The logical-to-physical mapping involves only an offset in *x* and *y* that is defined by the application-controlled window and viewport origins. The viewport and window extents are all set to 1, creating a one-to-one mapping.

Applications that display geometric shapes (circles, squares, polygons, and so on, make use of one of the device-independent mapping modes. For example, if you are writing an application to provide charting capabilities for a spreadsheet program and want to guarantee that the diameter of each pie chart is 2 inches, use the MM_LOENGLISH mapping mode and call the appropriate functions to draw and fill the chart. Specifying MM_LOENGLISH, guarantees that the diameter of the chart is consistent on any display or printer. If MM_TEXT is used instead of MM_LOENGLISH, a chart that appears circular on a VGA display would appear elliptical on an EGA display and would appear very small on a 300-dpi laser printer.

Application-Defined Mapping Modes

The two application-defined mapping modes (MM_ISOTROPIC and MM_ANISOTROPIC) are provided for application-specific mapping modes. The MM_ISOTROPIC mode guarantees that logical units in the x-direction and in the y-direction are equal, while the MM_ANISOTROPIC mode allows the units to differ. A CAD or drawing application can benefit from the MM_ISOTROPIC mapping mode but may need to specify logical units that correspond to the increments on an engineer's scale (1/64 inch). These units would be difficult to obtain with the predefined mapping modes

(MM_HIENGLISH or MM_HIMETRIC); however, they can easily be obtained by selecting the MM_ISOTROPIC (or MM_ANISOTROPIC) mode. The following example shows how to set logical units to 1/64 inch:

```
SetMapMode(hDC, MM_ISOTROPIC);
SetWindowExtEx(hDC, 64, 64, NULL);
SetViewportExtEx(hDC, GetDeviceCaps(hDC, LOGPIXELSX),
                      GetDeviceCaps(hDC, LOGPIXELSY),
NULL);
```

Device-Space to Physical-Device Transformation

The device-space to physical-device transformation is unique in several respects. For example, it is limited to translation and is controlled by the system. The sole purpose of this transformation is to ensure that the origin of device space is mapped to the proper point on the physical device. There are no functions to set this transformation, nor are there any functions to retrieve related data.

Default Transformations

Whenever an application creates a DC and immediately begins calling GDI drawing or output functions, it takes advantage of the default page-space to device-space, and device-space to client-area transformations. A world-to-page space transformation cannot happen until the application first calls the **SetGraphicsMode** function to set the mode to GM_ADVANCED and then calls the **SetWorldTransform** function.

Use of MM_TEXT (the default page-space to device-space transformation) results in a one-to-one mapping; that is, a given point in page space maps to the same point in device space. As previously mentioned, this transformation is not specified by a matrix. Instead, it is obtained by dividing the width of the viewport by the width of the window and the height of the viewport by the height of the window. In the default case, the viewport dimensions are 1 pixel by 1 pixel, and the window dimensions are 1 page unit by 1 page unit.

The device-space to physical-device (client area, desktop, or printer paper) transformation *always* results in a one-to-one mapping; that is, one unit in device space is always equivalent to one unit in the client area, on the desktop, or on a page. The sole purpose of this transformation is translation; it ensures that output appears correctly in an application's window no matter where that window is moved on the desktop.

The one unique aspect of MM_TEXT is the orientation of the y-axis in page space. In MM_TEXT, the positive y-axis extends downward and the negative y-axis extends upward.

Coordinate Space and Transformation Reference

Coordinate Space and Transformation Functions

ClientToScreen

The **ClientToScreen** function converts the client-area coordinates of a specified point to screen coordinates.

```
BOOL ClientToScreen(
  HWND hWnd,         // handle to window
  LPPOINT lpPoint    // screen coordinates
);
```

Parameters

hWnd

[in] Handle to the window whose client area is used for the conversion.

lpPoint

[in/out] Pointer to a **POINT** structure that contains the client coordinates to be converted. The new screen coordinates are copied into this structure if the function succeeds.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **ClientToScreen** function replaces the client-area coordinates in the **POINT** structure with the screen coordinates. The screen coordinates are relative to the upper-left corner of the screen. Note, a screen-coordinate point that is above the window's client area has a negative *y*-coordinate. Similarly, a screen coordinate to the left of a client area has a negative *x*-coordinate.

All coordinates are device coordinates.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **MapWindowPoints**, **POINT**, **ScreenToClient**

CombineTransform

The **CombineTransform** function concatenates two world-space to page-space transformations.

```
BOOL CombineTransform(
  LPXFORM lpxformResult,  // combined transformation
  CONST XFORM *lpxform1,  // first transformation
  CONST XFORM *lpxform2   // second transformation
);
```

Parameters

lpxformResult
  [out] Pointer to an **XFORM** structure that receives the combined transformation.

lpxform1
  [in] Pointer to an **XFORM** structure that specifies the first transformation.

lpxform2
  [in] Pointer to an **XFORM** structure that specifies the second transformation.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Applying the combined transformation has the same effect as applying the first transformation and then applying the second transformation.

The three transformations need not be distinct. For example, *lpxform1* can point to the same **XFORM** structure as *lpxformResult*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWorldTransform**, **ModifyWorldTransform**, **SetWorldTransform**, **XFORM**

DPtoLP

The **DPtoLP** function converts device coordinates into logical coordinates. The conversion depends on the mapping mode of the device context, the settings of the origins and extents for the window and viewport, and the world transformation.

```
BOOL DPtoLP(
  HDC hdc,              // handle to device context
  LPPOINT lpPoints,     // array of points
  int nCount            // count of points in array
);
```

Parameters

hdc

[in] Handle to the device context.

lpPoints

[in/out] Pointer to an array of **POINT** structures. The x-coordinates and y-coordinates contained in each **POINT** structure will be transformed.

nCount

[in] Specifies the number of points in the array.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **DPtoLP** function fails if the device coordinates exceed 27 bits, or if the converted logical coordinates exceed 32 bits. In the case of such an overflow, the results for all the points are undefined.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.

Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **LPtoDP**, **POINT**

GetCurrentPositionEx

The **GetCurrentPositionEx** function retrieves the current position in logical coordinates.

```
BOOL GetCurrentPositionEx(
  HDC hdc,          // handle to device context
  LPPOINT lpPoint // current position
);
```

Parameters

hdc

[in] Handle to the device context.

lpPoint

[out] Pointer to a **POINT** structure that receives the coordinates of the current position.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **MoveToEx**, **POINT**

GetGraphicsMode

The **GetGraphicsMode** function retrieves the current graphics mode for the specified device context.

```
int GetGraphicsMode(
  HDC hdc   // handle to device context
);
```

Parameters

hdc
[in] Handle to the device context.

Return Values

If the function succeeds, the return value is the current graphics mode. It can be one of the following values:

Value	Meaning
GM_ADVANCED	**Windows NT/2000:** The current graphics mode is the advanced graphics mode, a mode that allows world transformations. In this graphics mode, an application can set or modify the world transformation for the specified device context. **Windows 95/98:** The GM_ADVANCED value is not supported.
GM_COMPATIBLE	The current graphics mode is the compatible graphics mode, a mode that is compatible with 16-bit Windows. In this graphics mode, an application cannot set or modify the world transformation for the specified device context. The compatible graphics mode is the default graphics mode.

Otherwise, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can set the graphics mode for a device context by calling the **SetGraphicsMode** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.

Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **SetGraphicsMode**

GetMapMode

The **GetMapMode** function retrieves the current mapping mode.

```
int GetMapMode(
  HDC hdc   // handle to device context
);
```

Parameters

hdc
[in] Handle to the device context.

Return Values

If the function succeeds, the return value specifies the mapping mode.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The following are the various mapping modes:

Mode	Description
MM_ANISOTROPIC	Logical units are mapped to arbitrary units with arbitrarily scaled axes. Use the **SetWindowExtEx** and **SetViewportExtEx** functions to specify the units, orientation, and scaling required.
MM_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive *x* is to the right; positive *y* is up.
MM_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive *x* is to the right; positive *y* is up.

(continued)

(continued)

Mode	Description
MM_ISOTROPIC	Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the x-axis is equal to one unit along the y-axis. Use the **SetWindowExtEx** and **SetViewportExtEx** functions to specify the units and the orientation of the axes. Graphics device interface makes adjustments as necessary to ensure the *x* and *y* units remain the same size. (When the windows extent is set, the viewport will be adjusted to keep the units isotropic).
MM_LOENGLISH	Each logical unit is mapped to 0.01 inch. Positive *x* is to the right; positive *y* is up.
MM_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive *x* is to the right; positive *y* is up.
MM_TEXT	Each logical unit is mapped to one device pixel. Positive *x* is to the right; positive *y* is down.
MM_TWIPS	Each logical unit is mapped to one-twentieth of a printer's point (1/1440 inch, also called a "twip"). Positive *x* is to the right; positive *y* is up.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **SetMapMode**, **SetWindowExtEx**, **SetViewportExtEx**

GetViewportExtEx

The **GetViewportExtEx** function retrieves the x-extent and y-extent of the current viewport for the specified device context.

```
BOOL GetViewportExtEx(
  HDC hdc,      // handle to device context
  LPSIZE lpSize // viewport dimensions
);
```

Parameters

hdc

[in] Handle to the device context.

lpSize

[out] Pointer to a **SIZE** structure that receives the x-extent and y-extent, in device units.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWindowExtEx**, **SetViewportExtEx**, **SetWindowExtEx**

GetViewportOrgEx

The **GetViewportOrgEx** function retrieves the x-coordinates and y-coordinates of the viewport origin for the specified device context.

```
BOOL GetViewportOrgEx(
  HDC hdc,        // handle to device context
  LPPOINT lpPoint // viewport origin
);
```

Parameters

hdc

[in] Handle to the device context.

lpPoint

[out] Pointer to a **POINT** structure that receives the coordinates of the origin, in device units.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWindowOrgEx**, **POINT**, **SetViewportOrgEx**, **SetWindowOrgEx**

GetWindowExtEx

This function retrieves the x-extent and y-extent of the window for the specified device context.

```
BOOL GetWindowExtEx(
  HDC hdc,          // handle to device context
  LPSIZE lpSize     // window extents
);
```

Parameters

hdc

[in] Handle to the device context.

lpSize

[out] Pointer to a **SIZE** structure that receives the x-extent and y-extent in page-space units; that is, in logical units.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportExtEx**, **SetViewportExtEx**, **SetWindowExtEx**

GetWindowOrgEx

The **GetWindowOrgEx** function retrieves the x-coordinates and y-coordinates of the window origin for the specified device context.

```
BOOL GetWindowOrgEx(
  HDC hdc,          // handle to device context
  LPPOINT lpPoint   // window origin
);
```

Parameters

hdc
[in] Handle to the device context.

lpPoint
[out] Pointer to a **POINT** structure that receives the coordinates, in logical units, of the window origin.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportOrgEx**, **SetViewportOrgEx**, **SetWindowOrgEx**

GetWorldTransform

The **GetWorldTransform** function retrieves the current world-space to page-space transformation.

```
BOOL GetWorldTransform(
  HDC hdc,          // handle to device context
  LPXFORM lpXform   // transformation
);
```

Parameters

hdc

[in] Handle to the device context.

lpXform

[out] Pointer to an **XFORM** structure that receives the current world-space to page-space transformation.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The precision of the transformation may be altered if an application calls the **ModifyWorldTransform** function prior to calling **GetWorldTransform**. (This is because the internal format for storing transformation values uses a higher precision than a **FLOAT** value.)

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **ModifyWorldTransform**, **SetWorldTransform**

LPtoDP

The **LPtoDP** function converts logical coordinates into device coordinates. The conversion depends on the mapping mode of the device context, the settings of the origins and extents for the window and viewport, and the world transformation.

```
BOOL LPtoDP(
  HDC hdc,           // handle to device context
  LPPOINT lpPoints,  // array of points
  int nCount         // count of points in array
);
```

Parameters

hdc

[in] Handle to the device context.

lpPoints

[in/out] Pointer to an array of **POINT** structures. The x-coordinates and y-coordinates contained in each of the **POINT** structures will be transformed.

nCount

[in] Specifies the number of points in the array.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This function fails if the logical coordinates exceed 32 bits, or if the converted device coordinates exceed 27 bits. In the case of such an overflow, the results for all the points are undefined.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **DPtoLP**, **POINT**

MapWindowPoints

The **MapWindowPoints** function converts (maps) a set of points from a coordinate space relative to one window to a coordinate space relative to another window.

```
int MapWindowPoints(
  HWND hWndFrom,      // handle to source window
  HWND hWndTo,        // handle to destination window
  LPPOINT lpPoints,   // array of points to map
  UINT cPoints        // number of points in array
);
```

Parameters

hWndFrom

[in] Handle to the window from which points are converted. If this parameter is NULL or HWND_DESKTOP, the points are presumed to be in screen coordinates.

hWndTo

[in] Handle to the window to which points are converted. If this parameter is NULL or HWND_DESKTOP, the points are converted to screen coordinates.

lpPoints

[in/out] Pointer to an array of **POINT** structures that contain the set of points to be converted. The points are in device units. This parameter can also point to a **RECT** structure, in which case the *cPoints* parameter should be set to 2.

cPoints

[in] Specifies the number of **POINT** structures in the array pointed to by the *lpPoints* parameter.

Return Values

If the function succeeds, the low-order word of the return value is the number of pixels added to the horizontal coordinate of each source point in order to compute the horizontal coordinate of each destination point; the high-order word is the number of pixels added to the vertical coordinate of each source point in order to compute the vertical coordinate of each destination point.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If *hWndFrom* or *hWndTo* (or both) are mirrored windows (that is, have WS_EX_LAYOUTRTL extended style), **MapWindowPoints** will automatically adjust mirrored coordinates if you pass two or less points in *lpPoints*. If you pass more than two points, the function will not fail but it will return erroneous positions. Thus, to guarantee the correct transformation of rectangle coordinates, you must call **MapWindowPoints** with two or less points at a time, as shown in the following example:

```
    RECT        rc[10];

   for(int i =0; i < (sizeof(rc)/sizeof(rc[0])); i++)
   {
      MapWindowPoints(hWnd1, hWnd2, (LPPOINT)(&rc[i]),
(sizeof(RECT)/sizeof(POINT)) );
   }
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **ClientToScreen**, **POINT**, **RECT**, **ScreenToClient**

ModifyWorldTransform

The **ModifyWorldTransform** function changes the world transformation for a device context using the specified mode.

```
BOOL ModifyWorldTransform(
  HDC hdc,                // handle to device context
  CONST XFORM *lpXform,   // transformation data
  DWORD iMode             // modification mode
);
```

Parameters

hdc
[in] Handle to the device context.

lpXform
[in] Pointer to an **XFORM** structure used to modify the world transformation for the given device context.

iMode

[in] Specifies how the transformation data modifies the current world transformation. This parameter must be one of the following values:

Value	Description
MWT_IDENTITY	Resets the current world transformation by using the identity matrix. If this mode is specified, the **XFORM** structure pointed to by *lpXform* is ignored.
MWT_LEFTMULTIPLY	Multiplies the current transformation by the data in the **XFORM** structure. (The data in the **XFORM** structure becomes the left multiplicand, and the data for the current transformation becomes the right multiplicand.)
MWT_RIGHTMULTIPLY	Multiplies the current transformation by the data in the **XFORM** structure. (The data in the **XFORM** structure becomes the right multiplicand, and the data for the current transformation becomes the left multiplicand.)

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **ModifyWorldTransform** function will fail unless graphics mode for the specified device context has been set to GM_ADVANCED by previously calling the **SetGraphicsMode** function. Likewise, it will not be possible to reset the graphics mode for the device context to the default GM_COMPATIBLE mode, unless world transform has first been reset to the default identity transform by calling **SetWorldTransform** or **ModifyWorldTransform**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWorldTransform**, **SetGraphicsMode**, **SetWorldTransform**, **XFORM**

OffsetViewportOrgEx

The **OffsetViewportOrgEx** function modifies the viewport origin for a device context using the specified horizontal and vertical offsets.

```
BOOL OffsetViewportOrgEx(
  HDC hdc,         // handle to device context
  int nXOffset,    // horizontal offset
  int nYOffset,    // vertical offset
  LPPOINT lpPoint  // original origin
);
```

Parameters

hdc
[in] Handle to the device context.

nXOffset
[in] Specifies the horizontal offset, in device units.

nYOffset
[in] Specifies the vertical offset, in device units.

lpPoint
[out] Pointer to a **POINT** structure. The previous viewport origin, in device units, is placed in this structure. If *lpPoint* is NULL, the previous viewport origin is not returned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

The new origin is the sum of the current origin and the horizontal and vertical offsets.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportOrgEx**, **OffsetWindowOrgEx**, **SetViewportOrgEx**

OffsetWindowOrgEx

The **OffsetWindowOrgEx** function modifies the window origin for a device context using the specified horizontal and vertical offsets.

```
BOOL OffsetWindowOrgEx(
  HDC hdc,            // handle to device context
  int nXOffset,       // horizontal offset
  int nYOffset,       // vertical offset
  LPPOINT lpPoint     // original origin
);
```

Parameters

hdc

[in] Handle to the device context.

nXOffset

[in] Specifies the horizontal offset, in logical units.

nYOffset

[in] Specifies the vertical offset, in logical units.

lpPoint

[out] Pointer to a **POINT** structure. The logical coordinates of the previous window origin are placed in this structure. If *lpPoint* is NULL, the previous origin is not returned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.**Windows CE:** Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportOrgEx**, **OffsetViewportOrgEx**, **POINT**

ScaleViewportExtEx

The **ScaleViewportExtEx** function modifies the viewport for a device context (DC) by using the ratios formed by the specified multiplicands and divisors.

```
BOOL ScaleViewportExtEx(
  HDC hdc,         // handle to device context
  int Xnum,        // horizontal multiplicand
  int Xdenom,      // horizontal divisor
  int Ynum,        // vertical multiplicand
  int Ydenom,      // vertical divisor
  LPSIZE lpSize    // previous viewport extents
);
```

Parameters

hdc
  [in] Handle to the device context.

Xnum
  [in] Specifies the amount by which to multiply the current horizontal extent.

Xdenom
  [in] Specifies the amount by which to divide the current horizontal extent.

Ynum
  [in] Specifies the amount by which to multiply the current vertical extent.

Ydenom
  [in] Specifies the amount by which to divide the current vertical extent.

lpSize
  [out] Pointer to a **SIZE** structure that receives the previous viewport extents, in device units. If *lpSize* is NULL, this parameter is not used.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The viewport extents are modified as follows:

```
  xNewVE = (xOldVE * Xnum) / Xdenom
  yNewVE = (yOldVE * Ynum) / Ydenom
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportExtEx**, **SIZE**

ScaleWindowExtEx

The **ScaleWindowExtEx** function modifies the window for a device context using the ratios formed by the specified multiplicands and divisors.

```
BOOL ScaleWindowExtEx(
  HDC hdc,        // handle to device context
  int Xnum,       // horizontal multiplicand
  int Xdenom,     // horizontal divisor
  int Ynum,       // vertical multiplicand
  int Ydenom,     // vertical divisor
  LPSIZE lpSize   // previous window extents
);
```

Parameters

hdc
[in] Handle to the device context.

Xnum
[in] Specifies the amount by which to multiply the current horizontal extent.

Xdenom
[in] Specifies the amount by which to divide the current horizontal extent.

Ynum
[in] Specifies the amount by which to multiply the current vertical extent.

Ydenom
[in] Specifies the amount by which to divide the current vertical extent.

lpSize
[out] Pointer to a **SIZE** structure that receives the previous window extents, in logical units. If *lpSize* is NULL, this parameter is not used.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The window extents are modified as follows:

```
   xNewWE = (xOldWE * Xnum) / Xdenom
   yNewWE = (yOldWE * Ynum) / Ydenom
```

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWindowExtEx**, **SIZE**

ScreenToClient

The **ScreenToClient** function converts the screen coordinates of a specified point on the screen to client coordinates.

```
BOOL ScreenToClient(
  HWND hWnd,        // handle to window
  LPPOINT lpPoint   // screen coordinates
);
```

Parameters

hWnd
[in] Handle to the window whose client area will be used for the conversion.

lpPoint
[in] Pointer to a **POINT** structure that specifies the screen coordinates to be converted.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The function uses the window identified by the *hWnd* parameter and the screen coordinates given in the **POINT** structure to compute client coordinates. It then replaces the screen coordinates with the client coordinates. The new coordinates are relative to the upper-left corner of the specified window's client area.

The **ScreenToClient** function assumes the specified point is in screen coordinates.

All coordinates are in device units.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **ClientToScreen**, **MapWindowPoints**, **POINT**

SetGraphicsMode

The **SetGraphicsMode** function sets the graphics mode for the specified device context.

```
int SetGraphicsMode(
  HDC hdc,    // handle to device context
  int iMode   // graphics mode
);
```

Parameters

hdc

[in] Handle to the device context.

iMode

[in] Specifies the graphics mode. This parameter can be one of the following values:

Value	Meaning
GM_ADVANCED	**Windows NT/2000:** Sets the advanced graphics mode that allows world transformations. This value must be specified if the application will set or modify the world transformation for the specified device context. In this mode all graphics, including text output, fully conform to the world-to-device transformation specified in the device context. **Windows 95/98:** The GM_ADVANCED value is not supported. When playing enhanced metafiles, Windows 95/98 attempts to make enhanced metafiles on Windows 95/98 look the same as they do on Windows NT/Windows 2000. To accomplish this, Windows 95/98 may simulate GM_ADVANCED mode when playing specific enhanced metafile records.
GM_COMPATIBLE	Sets the graphics mode that is compatible with 16-bit Windows. This is the default mode. If this value is specified, the application can only modify the world-to-device transform by calling functions that set window and viewport extents and origins, but not by using **SetWorldTransform** or **ModifyWorldTransform**; calls to those functions will fail. Examples of functions that set window and viewport extents and origins are **SetViewportExtEx** and **SetWindowExtEx**.

Return Values

If the function succeeds, the return value is the old graphics mode.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

There are three areas in which graphics output differs according to the graphics mode:

- Text Output: In the GM_COMPATIBLE mode, TrueType (or vector font) text output behaves much the same way as raster font text output with respect to the world-to-device transformations in the DC. The TrueType text is always written from left to right and right side up, even if the rest of the graphics will be flipped on the *x*-axis or *y*-axis. Only the height of the TrueType (or vector font) text is scaled. The only way to write text that is not horizontal in the GM_COMPATIBLE mode is to specify nonzero escapement and orientation for the logical font selected in this device context.

  In the GM_ADVANCED mode, TrueType (or vector font) text output fully conforms to the world-to-device transformation in the device context. The raster fonts only have very limited transformation capabilities (stretching by some integer factors). Graphics device interface (GDI) tries to produce the best output it can with raster fonts for nontrivial transformations.

- Rectangle Exclusion: If the default GM_COMPATIBLE graphics mode is set, the system excludes bottom and rightmost edges when it draws rectangles.

  The GM_ADVANCED graphics mode is required if applications want to draw rectangles that are bottom-right inclusive.

- Arc Drawing: If the default GM_COMPATIBLE graphics mode is set, GDI draws arcs using the current arc direction in the device space. With this convention, arcs do not respect page-to-device transforms that require a flip along the *x*-axis or *y*-axis.

  If the GM_ADVANCED graphics mode is set, GDI always draws arcs in the counterclockwise direction in logical space. This is equivalent to the statement that, in the GM_ADVANCED graphics mode, both arc control points and arcs themselves fully respect the device context's world-to-device transformation.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **CreateDC**, **GetArcDirection**, **GetDC**, **GetGraphicsMode**, **ModifyWorldTransform**, **SetArcDirection**, **SetViewportExtent**, **SetViewportExtEx**, **SetWindowExtent**, **SetWindowExtEx**, **SetWorldTransform**

SetMapMode

The **SetMapMode** function sets the mapping mode of the specified device context. The mapping mode defines the unit of measure used to transform page-space units into device-space units, and also defines the orientation of the device's *x*-axisand *y*-axis.

```
int SetMapMode(
  HDC hdc,          // handle to device context
  int fnMapMode     // new mapping mode
);
```

Parameters

hdc

[in] Handle to the device context.

fnMapMode

[in] Specifies the new mapping mode. This parameter can be one of the following values:

Value	Description
MM_ANISOTROPIC	Logical units are mapped to arbitrary units with arbitrarily scaled axes. Use the **SetWindowExtEx** and **SetViewportExtEx** functions to specify the units, orientation, and scaling.
MM_HIENGLISH	Each logical unit is mapped to 0.001 inch. Positive *x* is to the right; positive *y* is up.
MM_HIMETRIC	Each logical unit is mapped to 0.01 millimeter. Positive *x* is to the right; positive *y* is up.
MM_ISOTROPIC	Logical units are mapped to arbitrary units with equally scaled axes; that is, one unit along the x-axis is equal to one unit along the y-axis. Use the **SetWindowExtEx** and **SetViewportExtEx** functions to specify the units and the orientation of the axes. Graphics device interface (GDI) makes adjustments as necessary to ensure the *x* and *y* units remain the same size (when the window extent is set, the viewport will be adjusted to keep the units isotropic).
MM_LOENGLISH	Each logical unit is mapped to 0.01 inch. Positive *x* is to the right; positive *y* is up.
MM_LOMETRIC	Each logical unit is mapped to 0.1 millimeter. Positive *x* is to the right; positive *y* is up.
MM_TEXT	Each logical unit is mapped to one device pixel. Positive *x* is to the right; positive *y* is down.
MM_TWIPS	Each logical unit is mapped to one twentieth of a printer's point (1/1440 inch, also called a twip). Positive *x* is to the right; positive *y* is up.

Return Values

If the function succeeds, the return value identifies the previous mapping mode.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The MM_TEXT mode allows applications to work in device pixels, whose size varies from device to device.

The MM_HIENGLISH, MM_HIMETRIC, MM_LOENGLISH, MM_LOMETRIC, and MM_TWIPS modes are useful for applications drawing in physically meaningful units (such as inches or millimeters).

The MM_ISOTROPIC mode ensures a 1:1 aspect ratio.

The MM_ANISOTROPIC mode allows the x-coordinates and y-coordinates to be adjusted independently.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetMapMode**, **SetViewportExtEx**, **SetViewportOrgEx**, **SetWindowExtEx**, **SetWindowOrgEx**

SetViewportExtEx

The **SetViewportExtEx** function sets the horizontal and vertical extents of the viewport for a device context by using the specified values.

```
BOOL SetViewportExtEx(
  HDC hdc,          // handle to device context
  int nXExtent,     // new horizontal viewport extent
  int nYExtent,     // new vertical viewport extent
  LPSIZE lpSize     // original viewport extent
);
```

Parameters

hdc
: [in] Handle to the device context.

nXExtent
: [in] Specifies the horizontal extent, in device units, of the viewport.

nYExtent
: [in] Specifies the vertical extent, in device units, of the viewport.

lpSize
: [out] Pointer to a **SIZE** structure that receives the previous viewport extents, in device units. If *lpSize* is NULL, this parameter is not used.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The *extent* is the maximum value of an axis. This function sets the maximum values for the horizontal and vertical axes of the viewport (in device coordinates or pixels). In combination with **SetWindowExtEx**, **SetViewportExtEx** determines the scaling factor between the window and the viewport.

When the following mapping modes are set, calls to the **SetWindowExtEx** and **SetViewportExtEx** functions are ignored:

- MM_HIENGLISH
- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExtEx** function before it calls **SetViewportExtEx**. Note that for the MM_ISOTROPIC mode certain portions of a nonsquare screen may not be available for display because the logical units on both axes represent equal physical distances.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportExtEx**, **SetWindowExtEx**, **SIZE**

SetViewportOrgEx

The **SetViewportOrgEx** function specifies which device point maps to the window origin (0,0).

```
BOOL SetViewportOrgEx(
  HDC hdc,          // handle to device context
  int X,            // new x-coordinate of viewport origin
  int Y,            // new y-coordinate of viewport origin
  LPPOINT lpPoint   // original viewport origin
);
```

Parameters

hdc

[in] Handle to the device context.

X

[in] Specifies the x-coordinate, in device units, of the new viewport origin.

Y

[in] Specifies the y-coordinate, in device units, of the new viewport origin.

lpPoint

[out] Pointer to a **POINT** structure that receives the previous viewport origin, in device coordinates. If *lpPoint* is NULL, this parameter is not used.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This helps define the mapping from the logical coordinate space (also known as a *window*) to the device coordinate space (the *viewport*). **SetViewportOrgEx** specifies which device point maps to the logical point (0,0). It has the effect of shifting the axes so that the logical point (0,0) no longer refers to the upper-left corner.

```
//map the logical point (0,0) to the device point
(xViewOrg, yViewOrg)
SetViewportOrgEx ( hdc, xViewOrg, yViewOrg, NULL)
```

This is related to the **SetViewportOrgEx** function. Generally, you will use one function or the other, but not both. Regardless of your use of **SetWindowOrgEx** and **SetViewportOrgEx**, the device point (0,0) is always the upper-left corner.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportOrgEx**, **POINT**, **SetWindowOrgEx**

SetWindowExtEx

The **SetWindowExtEx** function sets the horizontal and vertical extents of the window for a device context by using the specified values.

```
BOOL SetWindowExtEx(
  HDC hdc,          // handle to device context
  int nXExtent,     // new horizontal window extent
  int nYExtent,     // new vertical window extent
  LPSIZE lpSize     // original window extent
);
```

Parameters

hdc
[in] Handle to the device context.

nXExtent
[in] Specifies the window's horizontal extent in logical units.

nYExtent
[in] Specifies the window's vertical extent in logical units.

lpSize
[out] Pointer to a **SIZE** structure that receives the previous window extents, in logical units. If *lpSize* is NULL, this parameter is not used.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The *extent* is the maximum value of an axis. This function sets the maximum values for the horizontal and vertical axes of the window (in logical coordinates). In combination with **SetViewportExtEx**, **SetWindowExtEx** determines the scaling factor between the window and the viewport.

When the following mapping modes are set, calls to the **SetWindowExtEx** and **SetViewportExtEx** functions are ignored:

- MM_HIENGLISH
- MM_HIMETRIC
- MM_LOENGLISH
- MM_LOMETRIC
- MM_TEXT
- MM_TWIPS

When MM_ISOTROPIC mode is set, an application must call the **SetWindowExtEx** function before calling **SetViewportExtEx**. Note that for the MM_ISOTROPIC mode, certain portions of a nonsquare screen may not be available for display because the logical units on both axes represent equal physical distances.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWindowExtEx**, **SetViewportExtEx**, **SIZE**

SetWindowOrgEx

The **SetWindowOrgEx** function specifies which window point maps to the viewport origin (0,0).

```
BOOL SetWindowOrgEx(
  HDC hdc,          // handle to device context
  int X,            // new x-coordinate of window origin
  int Y,            // new y-coordinate of window origin
  LPPOINT lpPoint   // original window origin
);
```

Parameters

hdc

[in] Handle to the device context.

X

[in] Specifies the logical x-coordinate of the new window origin.

Y

[in] Specifies the logical y-coordinate of the new window origin.

lpPoint

[out] Pointer to a **POINT** structure that receives the previous origin of the window. If *lpPoint* is NULL, this parameter is not used.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This helps define the mapping from the logical coordinate space (also known as a *window*) to the device coordinate space (the *viewport*). **SetWindowOrgEx** specifies which logical point maps to the device point (0,0). It has the effect of shifting the axes so that the logical point (0,0) no longer refers to the upper-left corner.

```
//map the logical point (xWinOrg, yWinOrg) to the device
point (0,0)
SetWindowOrgEx (hdc, xWinOrg, yWinOrg, NULL)
```

This is related to the **SetViewportOrgEx** function. Generally, you will use one function or the other, but not both. Regardless of your use of **SetWindowOrgEx** and **SetViewportOrgEx**, the device point (0,0) is always the upper-left corner.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetViewportOrgEx**, **GetWindowOrgEx**, **POINT**, **SetViewportOrgEx**

SetWorldTransform

The **SetWorldTransform** function sets a two-dimensional linear transformation between world space and page space for the specified device context. This transformation can be used to scale, rotate, shear, or translate graphics output.

```
BOOL SetWorldTransform(
  HDC hdc,               // handle to device context
  CONST XFORM *lpXform   // transformation data
);
```

Parameters

hdc

[in] Handle to the device context.

lpXform

[in] Pointer to an **XFORM** structure that contains the transformation data.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

For any coordinates (x, y) in world space, the transformed coordinates in page space (x', y') can be determined by the following algorithm:

```
x' = x * eM11 + y * eM21 + eDx,
y' = x * eM12 + y * eM22 + eDy,
```

where the transformation matrix is represented by the following:

```
| eM11 eM12 0 |
| eM21 eM22 0 |
| eDx  eDy  1 |
```

This function uses logical units.

The world transformation is usually used to scale or rotate logical images in a device-independent way.

The default world transformation is the identity matrix with zero offset.

The **SetWorldTransform** function will fail unless the graphics mode for the given device context has been set to GM_ADVANCED by previously calling the **SetGraphicsMode** function. Likewise, it will not be possible to reset the graphics mode for the device context to the default GM_COMPATIBLE mode, unless the world transformation has first been reset to the default identity transformation by calling **SetWorldTransform** or **ModifyWorldTransform**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Functions, **GetWorldTransform**, **ModifyWorldTransform**, **SetGraphicsMode**, **SetMapMode**, **SetViewportExtEx**, **SetViewportOrgEx**, **SetWindowExtEx**, **SetWindowOrgEx**, **XFORM**

Coordinate Space and Transformation Structures

XFORM

The **XFORM** structure specifies a world-space to page-space transformation.

```
typedef struct _XFORM {
  FLOAT eM11;
  FLOAT eM12;
  FLOAT eM21;
  FLOAT eM22;
  FLOAT eDx;
  FLOAT eDy;
} XFORM, *PXFORM;
```

Members

eM11

Specifies the following:

Operation	Meaning
Scaling	Horizontal scaling component
Rotation	Cosine of rotation angle
Reflection	Horizontal component

eM12

Specifies the following:

Operation	Meaning
Shear	Horizontal proportionality constant
Rotation	Sine of the rotation angle

eM21

Specifies the following:

Operation	Meaning
Shear	Vertical proportionality constant
Rotation	Negative sine of the rotation angle

eM22

Specifies the following:

Operation	Meaning
Scaling	Vertical scaling component
Rotation	Cosine of rotation angle
Reflection	Vertical reflection component

eDx

Specifies the horizontal translation component, in logical units.

eDy

Specifies the vertical translation component, in logical units.

Remarks

The following list describes how the members are used for each operation:

Operation	eM11	eM12	eM21	eM22
Rotation	Cosine	Sine	Negative sine	Cosine
Scaling	Horizontal scaling component	Not used	Not used	Vertical scaling component
Shear	Not used	Horizontal proportionality constant	Vertical proportionality constant	Not used
Reflection	Horizontal reflection component	Not used	Not used	Vertical reflection component

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Coordinate Spaces and Transformations Overview, Coordinate Space and Transformation Structures, **ExtCreateRegion**, **GetWorldTransform**, **ModifyWorldTransform**, **PlayEnhMetaFile**, **SetWorldTransform**

CHAPTER 11

Device Contexts

A *device context* is a structure that defines a set of graphic objects and their associated attributes, and the graphic modes that affect output. The *graphic objects* include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations.

About Device Contexts

Device independence is one of the chief features of Windows. Win32-based applications can draw and print output on a variety of devices. The software that supports this device independence is contained in two dynamic-link libraries. The first, Gdi.dll, is referred to as the graphical device interface (GDI); the second is referred to as a device driver. The name of the second depends on the device where the application draws output. For example, if the application draws output in the client area of its window on a VGA display, this library is Vga.dll; if the application prints output on an Epson FX-80 printer, this library is Epson9.dll.

An application must instruct GDI to load a particular device driver and, once the driver is loaded, to prepare the device for drawing operations (such as selecting a line color and width, a brush pattern and color, a font typeface, a clipping region, and so on). These tasks are accomplished by creating and maintaining a device context (DC). A DC is a structure that defines a set of graphic objects and their associated attributes, and the graphic modes that affect output. The graphic objects include a pen for line drawing, a brush for painting and filling, a bitmap for copying or scrolling parts of the screen, a palette for defining the set of available colors, a region for clipping and other operations, and a path for painting and drawing operations. Unlike most of the structures, an application never has direct access to the DC; instead, it operates on the structure indirectly by calling various functions.

Graphic Objects

The pen, brush, bitmap, palette, region, and path associated with a DC are referred to as its graphic objects. The following attributes are associated with each of these objects:

Graphic object	Associated attributes
Bitmap	Size, in bytes; dimensions, in pixels; color-format; compression scheme; and so on.
Brush	Style, color, pattern, and origin.
Palette	Colors and size (or number of colors).
Font	Typeface name, width, height, weight, character set, and so on.
Path	Shape.
Pen	Style, width, and color.
Region	Location and dimensions.

When an application creates a DC, the system automatically stores a set of default objects in it. (There is no default bitmap or path.) An application can examine the attributes of the default objects by calling the **GetCurrentObject** and **GetObject** functions. The application can change these defaults by creating a new object and selecting it into the DC. An object is selected into a DC by calling the **SelectObject** function.

An application can set the current brush color to a specified color value with **SetDCBrushColor**.

The **GetDCBrushColor** function returns the DC brush color. The **SetDCPenColor** function sets the pen color to a specified color value. The **GetDCPenColor** function returns the DC pen color.

Graphic Modes

The Win32 API supports five graphic modes that allow an application to specify how colors are mixed, where output appears, how the output is scaled, and so on. These modes, which are stored in a DC, are described in the following table:

Graphics mode	Description
Background	Defines how background colors are mixed with existing window or screen colors for bitmap and text operations.
Drawing	Defines how foreground colors are mixed with existing window or screen colors for pen, brush, bitmap, and text operations.
Mapping	Defines how graphics output is mapped from logical (or world) space onto the window, screen, or printer paper.
Polygon-fill	Defines how the brush pattern is used to fill the interior of complex regions.
Stretching	Defines how bitmap colors are mixed with existing window or screen colors when the bitmap is compressed (or scaled down).

As it does with graphic objects, the system initializes a DC with default graphic modes. An application can retrieve and examine these default modes by calling the following functions:

Graphics mode	Function
Background	**GetBkMode**
Drawing	**GetROP2**
Mapping	**GetMapMode**
Polygon-fill	**GetPolyFillMode**
Stretching	**GetStretchBltMode**

An application can change the default modes by calling one of the following functions:

Graphics mode	Function
Background	**SetBkMode**
Drawing	**SetROP2**
Mapping	**SetMapMode**
Polygon-fill	**SetPolyFillMode**
Stretching	**SetStretchBltMode**

Device Context Types

There are four types of DCs: display, printer, memory (or compatible), and information. Each type serves a specific purpose, as described in the following table:

Device context	Description
Display	Supports drawing operations on a video display.
Printer	Supports drawing operations on a printer or plotter.
Memory	Supports drawing operations on a bitmap.
Information	Supports the retrieval of device data.

Display Device Contexts

An application obtains a display DC by calling the **BeginPaint**, **GetDC**, or **GetDCEx** function and identifying the window in which the corresponding output will appear. Typically, an application obtains a display DC only when it must draw in the client area. When the application is finished drawing, it must release the DC by calling the **EndPaint** or **ReleaseDC** function.

There are three types of DCs for video displays:

- Class
- Common
- Private

Class Device Contexts

Class device contexts are supported strictly for compatibility with 16-bit versions of Windows. When writing a Win32-based application, avoid using the class device context; use a private device context instead.

Common Device Contexts

Common device contexts are display DCs maintained in a special cache by the system. Common device contexts are used in applications that perform infrequent drawing operations. Before the system returns the DC handle, it initializes the common device context with default objects, attributes, and modes. Any drawing operations performed by the application use these defaults unless one of the GDI functions is called to select a new object, change the attributes of an existing object, or select a new mode.

Because only a limited number of common device contexts exist, an application should release them after it has finished drawing. When the application releases a common device context, any changes to the default data are lost.

Private Device Contexts

Private device contexts are display DCs that, unlike common device contexts, retain any changes to the default data—even after an application releases them. Private device contexts are used in applications that perform numerous drawing operations such as computer-aided design (CAD) applications, desktop-publishing applications, drawing and painting applications, and so on. Private device contexts are not part of the system cache and therefore need not be released after use. The system automatically removes a private device context after the last window of that class has been destroyed.

An application creates a private device context by first specifying the CS_OWNDC window-class style when it initializes the **style** member of the **WNDCLASS** structure and calls the **RegisterClass** function. For more information about window classes, see *Window Classes*.

After creating a window with the CS_OWNDC style, an application can call the **GetDC**, **GetDCEx**, or **BeginPaint** function once to obtain a handle identifying a private device context. The application can continue using this handle (and the associated DC) until it deletes the window created with this class. Any changes to graphic objects and their attributes, or graphic modes are retained by the system until the window is deleted.

Printer Device Contexts

The printer DC can be used when printing on a dot-matrix printer, ink-jet printer, laser printer, or plotter. An application creates a printer DC by calling the **CreateDC** function and supplying the appropriate arguments (the name of the printer driver, the name of the printer, the file or device name for the physical output medium, and other initialization data). When an application has finished printing, it deletes the printer DC by calling the **DeleteDC** function. An application must delete (rather than release) a printer DC; the **ReleaseDC** function fails when an application attempts to use it to release a printer DC.

For more information, see *Printer Output*.

Memory Device Contexts

To enable applications to place output in memory rather than sending it to an actual device, use a special device context for bitmap operations called a *memory device context*. A memory DC enables the system to treat a portion of memory as a virtual device. It is an array of bits in memory that an application can use temporarily to store the color data for bitmaps created on a normal drawing surface. Because the bitmap is compatible with the device, a memory DC is also sometimes referred to as a *compatible device context*.

The memory DC stores bitmap images for a particular device. An application can create a memory DC by calling the **CreateCompatibleDC** function.

The original bitmap in a memory DC is simply a placeholder. Its dimensions are one pixel by one pixel. Before an application can begin drawing, it must select a bitmap with the appropriate width and height into the DC by calling the **SelectObject** function. To create a bitmap of the appropriate dimensions, use the **CreateBitmap**, **CreateBitmapIndirect**, or **CreateCompatibleBitmap** function. After the bitmap is selected into the memory DC, the system replaces the single-bit array with an array large enough to store color information for the specified rectangle of pixels.

When an application passes the handle returned by **CreateCompatibleDC** to one of the drawing functions, the requested output does not appear on a device's drawing surface. Instead, the system stores the color information for the resultant line, curve, text or region in the array of bits. The application can copy the image stored in memory back onto a drawing surface by calling the **BitBlt** function, identifying the memory DC as the source device context and a window or screen DC as the target device context.

When displaying a DIB or a DDB created from a DIB on a palette device, you can improve the speed at which the image is drawn by arranging the logical palette to match the layout of the system palette. To do this, call **GetDeviceCaps** with the NUMRESERVED value to get the number of reserved colors in the system. Then call **GetSystemPaletteEntries** and fill in the first and last NUMRESERVED/2 entries of the logical palette with the corresponding system colors. For example, if NUMRESERVED is 20, you would fill in the first and last 10 entries of the logical palette with the system colors. Then fill in the remaining 256-NUMRESERVED colors of the logical palette (in our example, the remaining 236 colors) with colors from the DIB and set the PC_NOCOLLAPSE flag on each of these colors.

For more information about color and palettes, see *Colors*. For more information about bitmaps and bitmap operations, see *Bitmaps*.

Information Device Contexts

The information DC is used to retrieve default device data. For example, an application can call the **CreateIC** function to create an information DC for a particular model of printer and then call the **GetCurrentObject** and **GetObject** functions to retrieve the default pen or brush attributes. Because the system can retrieve device information without creating the structures normally associated with the other types of device contexts, an information DC involves far less overhead and is created significantly faster than any of the other types. After an application finishes retrieving data by using an information DC, it must call the **DeleteDC** function.

Device Context Operations

An application can perform the following operations on a device context:

- Enumerate existing graphic objects.
- Select new graphic objects.
- Delete existing graphic objects.
- Save the current graphic objects, their attributes, and the graphic modes.
- Restore previously saved graphic objects, their attributes, and the graphic modes.

In addition, an application can use a device context to:

- Determine how graphics output is translated.
- Cancel lengthy drawing operations (begun by a thread in a multithreaded application).
- Reset a printer to a particular state.

Operations on Graphic Objects

After an application creates a display or printer DC, it can begin drawing on the associated device or, in the case of the memory DC, it can begin drawing on the bitmap stored in memory. However, before drawing begins—and sometimes while drawing is in progress—it is often necessary to replace the default objects with new objects.

An application can examine a default object's attributes by calling the **GetCurrentObject** and **GetObject** functions. The **GetCurrentObject** function returns a handle identifying the current pen, brush, palette, bitmap, or font, and the **GetObject** function initializes a structure containing that object's attributes.

Some printers provide resident pens, brushes, and fonts that can be used to improve drawing speed in an application. Two functions can be used to enumerate these objects: **EnumObjects** and **EnumFontFamilies**. If the application must enumerate resident pens or brushes, it can call the **EnumObjects** function to examine the corresponding attributes. If the application must enumerate resident fonts, it can call the **EnumFontFamilies** function (which can also enumerate GDI fonts).

Once an application determines that a default object needs replacing, it creates a new object by calling one of the following creation functions:

Graphic object	Function
Bitmap	**CreateBitmap**, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, **CreateDiscardableBitmap**, **CreateDIBitmap**
Brush	**CreateBrushIndirect**, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**
Color Palette	**CreatePalette**
Font	**CreateFont**, **CreateFontIndirect**
Pen	**CreatePen**, **CreatePenIndirect**, **ExtCreatePen**
Region	**CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreatePolygonRgn**, **CreatePolyPolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**, **CreateRoundRectRgn**

Each of these functions returns a handle identifying a new object. After an application retrieves a handle, it must call the **SelectObject** function to replace the default object. However, the application should save the handle identifying the default object and use this handle to replace the new object when it is no longer needed. When the application finishes drawing with the new object, it must restore the default object by calling the **SelectObject** function and then delete the new object by calling the **DeleteObject** function. Failing to delete objects causes serious performance problems.

Cancellation of Drawing Operations

When complex drawing applications perform lengthy graphics operations, they consume valuable system resources. By taking advantage of the system's multitasking features, an application can use threads and the **CancelDC** function to manage these operations. For example, if the graphics operation performed by thread A is consuming needed resources, thread B can call the **CancelDC** function to halt that operation.

Retrieving Device Data

The Win32 API provides two functions that applications can use to retrieve device data using a device context: **GetDeviceCaps** and **DeviceCapabilities**.

GetDeviceCaps retrieves general device data for the following devices:

- Raster displays
- Dot-matrix printers
- Ink-jet printers
- Laser printers
- Vector plotters
- Raster cameras

The data includes the supported capabilities of the device, including device resolution (for video displays), color format (for video displays and color printers), number of graphic objects, raster capabilities, curve drawing, line drawing, polygon drawing, and text drawing. An application retrieves this data by supplying a handle identifying the appropriate device context, as well as an index specifying the type of data the function is to retrieve.

The **DeviceCapabilities** function retrieves data specific to printers, including the number of available paper bins, the duplex capabilities of the printer, the resolutions supported by the printer, the maximum and minimum supported paper size, and so on. An application retrieves this data by supplying strings specifying a printer device and port, as well as an index specifying the type of data that the function is to retrieve.

Saving, Restoring, and Resetting a Device Context

The Win32 API provides three functions that an application can use to save, restore, and reset a device context: **SaveDC**, **RestoreDC**, and **ResetDC**. The **SaveDC** function records on a special GDI stack the current DC's graphic objects and their attributes, and graphic modes. A drawing application can call this function before a user begins drawing and save the application's original state—providing a clean slate for the user. To return to this original state, the application calls the **RestoreDC** function.

ResetDC is provided to reset printer DC data. An application calls this function to reset the paper orientation, paper size, output scaling factor, number of copies to be printed, paper source (or bin), duplex mode, and so on. Typically, an application calls this function after a user has changed one of the printer options and the system has issued a **WM_DEVMODECHANGE** message.

ICM-Enabled Device Context Functions

Microsoft Windows 98 and Windows 2000 work with Microsoft Image Color Management (ICM). ICM technology ensures that a color image, graphic, or text object is rendered as closely as possible to its original intent on any device, despite differences in imaging technologies and color capabilities between devices. For more information, see *About Image Color Management Version 2.0.*

There are various functions in the graphics device interface (GDI) that use or operate on color data. The following device context functions are enabled for use with ICM:

- **CreateCompatibleDC**
- **CreateDC**
- **GetDCBrushColor**
- **GetDCPenColor**
- **ResetDC**
- **SelectObject**
- **SetDCBrushColor**
- **SetDCPenColor**

Device Context Reference

Device Context Functions

CancelDC

The **CancelDC** function cancels any pending operation on the specified device context (DC).

```
BOOL CancelDC(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
  [in] Handle to the DC.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **CancelDC** function is used by multithreaded applications to cancel lengthy drawing operations. If thread A initiates a lengthy drawing operation, thread B may cancel that operation by calling this function.

If an operation is canceled, the affected thread returns an error and the result of its drawing operation is undefined. The results are also undefined if no drawing operation was in progress when the function was called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **CreateThread**, **GetCurrentThread**

ChangeDisplaySettings

The **ChangeDisplaySettings** function changes the settings of the default display device to the graphics mode specified in *lpDevMode*.

To change the settings of a specified display device, use the **ChangeDisplaySettingsEx** function.

```
LONG ChangeDisplaySettings(
  LPDEVMODE lpDevMode,  // graphics mode
  DWORD dwflags         // graphics mode options
);
```

Parameters

lpDevMode

[in] Pointer to a **DEVMODE** structure that describes the new graphics mode. If *lpDevMode* is NULL, all the values currently in the registry will be used for the display setting. Passing NULL for the *lpDevMode* parameter and 0 for the *dwFlags* parameter is the easiest way to return to the default mode after a dynamic mode change.

The **dmSize** member of **DEVMODE** must be initialized to the size, in bytes, of the **DEVMODE** structure. The **dmDriverExtra** member of **DEVMODE** must be initialized to indicate the number of bytes of private driver data following the **DEVMODE** structure. In addition, you can use any or all of the following members of the **DEVMODE** structure:

Member	Meaning
dmBitsPerPel	Bits per pixel
dmPelsWidth	Pixel width
dmPelsHeight	Pixel height
dmDisplayFlags	Mode flags
dmDisplayFrequency	Mode frequency
dmPosition	**Windows 98, Windows 2000:** Position of the device in a multimonitor configuration

In addition to using one or more of the preceding **DEVMODE** members, you must also set one or more of the following values in the **dmFields** member to change the display setting:

Value	Meaning
DM_BITSPERPEL	Use the **dmBitsPerPel** value.
DM_PELSWIDTH	Use the **dmPelsWidth** value.
DM_PELSHEIGHT	Use the **dmPelsHeight** value.
DM_DISPLAYFLAGS	Use the **dmDisplayFlags** value.
DM_DISPLAYFREQUENCY	Use the **dmDisplayFrequency** value.
DM_POSITION	**Windows 98, Windows 2000:** Use the **dmPosition** value.

dwflags

[in] Indicates how the graphics mode should be changed. This parameter can be one of the following values:

Value	Meaning
0	The graphics mode for the current screen will be changed dynamically.
CDS_UPDATEREGISTRY	The graphics mode for the current screen will be changed dynamically and the graphics mode will be updated in the registry. The mode information is stored in the USER profile.
CDS_TEST	The system tests if the requested graphics mode could be set.
CDS_FULLSCREEN	The mode is temporary in nature. **Windows NT/2000:** If you change to and from another desktop, this mode will not be reset.

(continued)

(continued)

Value	Meaning
CDS_GLOBAL	The settings will be saved in the global settings area so that they will affect all users on the machine. Otherwise, only the settings for the user are modified. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.
CDS_SET_PRIMARY	This device will become the primary device.
CDS_RESET	The settings should be changed, even if the requested settings are the same as the current settings.
CDS_NORESET	The settings will be saved in the registry, but will not take affect. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.

Specifying CDS_TEST allows an application to determine which graphics modes are actually valid, without causing the system to change to that graphics mode.

If CDS_UPDATEREGISTRY is specified and it is possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_SUCCESSFUL is returned. If it is not possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_RESTART is returned.

Windows NT/2000: If CDS_UPDATEREGISTRY is specified and the information could not be stored in the registry, the graphics mode is not changed and DISP_CHANGE_NOTUPDATED is returned.

Return Values

The **ChangeDisplaySettings** function returns one of the following values:

Value	Meaning
DISP_CHANGE_SUCCESSFUL	The settings change was successful.
DISP_CHANGE_RESTART	The computer must be restarted in order for the graphics mode to work.
DISP_CHANGE_BADFLAGS	An invalid set of flags was passed in.
DISP_CHANGE_BADPARAM	An invalid parameter was passed in. This can include an invalid flag or combination of flags.
DISP_CHANGE_FAILED	The display driver failed the specified graphics mode.
DISP_CHANGE_BADMODE	The graphics mode is not supported.
DISP_CHANGE_NOTUPDATED	**Windows NT/2000:** Unable to write settings to the registry.

Remarks

To ensure that the **DEVMODE** structure passed to **ChangeDisplaySettings** is valid and contains only values supported by the display driver, use the **DEVMODE** returned by the **EnumDisplaySettings** function.

When the display mode is changed dynamically, the **WM_DISPLAYCHANGE** message is sent to all running applications with the following message parameters:

Parameters	Meaning
wParam	New bits per pixel
LOWORD(lParam)	New pixel width
HIWORD(lParam)	New pixel height

Windows 95: If the calling thread has any top-level windows, **ChangeDisplaySettings** sends these windows the **WM_DISPLAYCHANGE** message right away (for all other windows the message is posted). This may cause the shell to get its message too soon and could squash icons. To avoid this problem, have **ChangeDisplaySettings** do resolution switching by calling on a thread with no windows, for example, a new thread.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettingsEx**, **CreateDC**, **DEVMODE**, **EnumDisplayDevices**, **EnumDisplaySettings**, **WM_DISPLAYCHANGE**

ChangeDisplaySettingsEx

The **ChangeDisplaySettingsEx** function changes the settings of the display device specified in the *lpszDeviceName* parameter to the graphics mode specified in the *lpDevMode* parameter.

```
LONG ChangeDisplaySettingsEx(
  LPCTSTR lpszDeviceName,  // name of display device
  LPDEVMODE lpDevMode,     // graphics mode
  HWND hwnd,               // not used; must be NULL
  DWORD dwflags            // graphics mode options
  LPVOID lParam            // video parameters (or NULL)
);
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device whose graphics mode the function will obtain information about. See ***EnumDisplayDevices*** for further information on the names associated with these display devices.

The *lpszDeviceName* parameter can be NULL. A NULL value specifies the default display device.

lpDevMode

[in] Pointer to a **DEVMODE** structure that describes the new graphics mode. If *lpDevMode* is NULL, all the values currently in the registry will be used for the display setting. Passing NULL for the *lpDevMode* parameter and 0 for the *dwFlags* parameter is the easiest way to return to the default mode after a dynamic mode change.

The **dmSize** member must be initialized to the size, in bytes, of the **DEVMODE** structure. The **dmDriverExtra** member must be initialized to indicate the number of bytes of private driver data following the **DEVMODE** structure. In addition, you can use any of the following members of the **DEVMODE** structure:

Member	Meaning
dmBitsPerPel	Bits per pixel
dmPelsWidth	Pixel width
dmPelsHeight	Pixel height
dmDisplayFlags	Mode flags
dmDisplayFrequency	Mode frequency
dmPosition	**Windows 98, Windows 2000:** Position of the device in a multi-monitor configuration.

In addition to using one or more of the preceding **DEVMODE** members, you must also set one or more of the following values in the **dmFields** member to change the display settings:

Value	Meaning
DM_BITSPERPEL	Use the **dmBitsPerPel** value.
DM_PELSWIDTH	Use the **dmPelsWidth** value.
DM_PELSHEIGHT	Use the **dmPelsHeight** value.
DM_DISPLAYFLAGS	Use the **dmDisplayFlags** value.
DM_DISPLAYFREQUENCY	Use the **dmDisplayFrequency** value.
DM_POSITION	**Windows 98, Windows 2000:** Use the **dmPosition** value.

hwnd

Reserved; must be NULL.

dwflags

[in] Indicates how the graphics mode should be changed. This parameter can be one of the following values:

Value	Meaning
0	The graphics mode for the current screen will be changed dynamically.
CDS_FULLSCREEN	The mode is temporary in nature. **Windows NT/2000:** If you change to and from another desktop, this mode will not be reset.
CDS_GLOBAL	The settings will be saved in the global settings area so that they will affect all users on the machine. Otherwise, only the settings for the user are modified. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.
CDS_NORESET	The settings will be saved in the registry, but will not take effect. This flag is only valid when specified with the CDS_UPDATEREGISTRY flag.
CDS_RESET	The settings should be changed, even if the requested settings are the same as the current settings.
CDS_SET_PRIMARY	This device will become the primary device.
CDS_TEST	The system tests if the requested graphics mode could be set.
CDS_UPDATEREGISTRY	The graphics mode for the current screen will be changed dynamically and the graphics mode will be updated in the registry. The mode information is stored in the USER profile.
CDS_VIDEOPARAMETERS	**Windows NT/2000:** When set, the *lParam* parameter is a pointer to a **VIDEOPARAMETERS** structure.

Specifying CDS_TEST allows an application to determine which graphics modes are actually valid, without causing the system to change to that graphics mode.

If CDS_UPDATEREGISTRY is specified and it is possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_SUCCESSFUL is returned. If it is not possible to change the graphics mode dynamically, the information is stored in the registry and DISP_CHANGE_RESTART is returned.

Windows NT/2000: If CDS_UPDATEREGISTRY is specified and the information could not be stored in the registry, the graphics mode is not changed and DISP_CHANGE_NOTUPDATED is returned.

lParam

Windows NT/2000: [in] If *dwFlags* is CDS_VIDEOPARAMETERS, *lParam* is a pointer to a **VIDEOPARAMETERS** structure. Otherwise *lParam* must be NULL.

Return Values

The **ChangeDisplaySettingsEx** function returns one of the following values:

Value	Meaning
DISP_CHANGE_BADESC	**Windows NT/2000:** Driver does not support this functionality, or the driver returned an error.
DISP_CHANGE_BADFLAGS	An invalid set of flags was passed in.
DISP_CHANGE_BADMODE	The graphics mode is not supported.
DISP_CHANGE_BADPARAM	An invalid parameter was passed in. This can include an invalid flag or combination of flags.
DISP_CHANGE_FAILED	The display driver failed the specified graphics mode.
DISP_CHANGE_NOTUPDATED	**Windows NT/2000:** Unable to write settings to the registry.
DISP_CHANGE_RESTART	The computer must be restarted for the graphics mode to work.
DISP_CHANGE_SUCCESSFUL	The settings change was successful.

Remarks

To ensure that the **DEVMODE** structure passed to **ChangeDisplaySettingsEx** is valid and contains only values supported by the display driver, use the **DEVMODE** returned by the **EnumDisplaySettings** function.

When the display mode is changed dynamically, the **WM_DISPLAYCHANGE** message is sent to all running applications with the following message parameters:

Parameters	Meaning
wParam	New bits per pixel
LOWORD(lParam)	New pixel width
HIWORD(lParam)	New pixel height

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Device Contexts Overview, Device Context Functions, **CreateDC**, **DEVMODE**, **EnumDisplayDevices**, **EnumDisplaySettings**, **VIDEOPARAMETERS**, **WM_DISPLAYCHANGE**

CreateCompatibleDC

The **CreateCompatibleDC** function creates a memory device context (DC) compatible with the specified device.

```
HDC CreateCompatibleDC(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to an existing DC. If this handle is NULL, the function creates a memory DC compatible with the application's current screen.

Return Values

If the function succeeds, the return value is the handle to a memory DC.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A memory DC exists only in memory. When the memory DC is created, its display surface is exactly one monochrome pixel wide and one monochrome pixel high. Before an application can use a memory DC for drawing operations, it must select a bitmap of the correct width and height into the DC. To select a bitmap into a DC, use the **CreateCompatibleBitmap** function, specifying the height, width, and color organization required.

When a memory DC is created, all attributes are set to normal default values. The memory DC can be used as a normal DC. You can set the attributes; obtain the current settings of its attributes; and select pens, brushes, and regions.

The **CreateCompatibleDC** function can only be used with devices that support raster operations. An application can determine whether a device supports these operations by calling the **GetDeviceCaps** function.

When you no longer need the memory DC, call the **DeleteDC** function.

ICM: If the DC that is passed to this function is enabled for Independent Color Management (ICM), the DC created by the function is ICM-enabled. The source and destination color spaces are specified in the DC.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **CreateCompatibleBitmap**, **DeleteDC**, **GetDeviceCaps**

CreateDC

The **CreateDC** function creates a device context (DC) for a device by using the specified name.

```
HDC CreateDC(
  LPCTSTR lpszDriver,        // driver name
  LPCTSTR lpszDevice,        // device name
  LPCTSTR lpszOutput,        // not used; should be NULL
  CONST DEVMODE *lpInitData  // optional printer data
);
```

Parameters

lpszDriver

Windows NT/2000: [in] Pointer to a null-terminated character string that specifies either DISPLAY for a display driver, or the name of a printer driver, which is usually WINSPOOL.

Windows 95/98: In Win32-based applications, this parameter is ignored and should be NULL, with one exception: You may obtain a display DC by specifying the null-terminated string DISPLAY. If this parameter is DISPLAY, all other parameters must be NULL.

lpszDevice

[in] Pointer to a null-terminated character string that specifies the name of the specific output device being used, as shown by the Print Manager (for example, Epson FX-80). It is not the printer model name. The *lpszDevice* parameter must be used.

Windows NT/2000: If *lpszDriver* is DISPLAY, *lpszDevice* must be NULL or the device name of a specific display device (of the form \\.\DisplayX, where X is a positive integer). If *lpszDevice* is NULL or \\.\DISPLAY1, then a DC is created for the primary display device.

Windows NT 3.51 and 4.0: Only the primary display is possible.

Windows2000: It is possible to have more than one monitor on the system. See ***EnumDisplayDevices*** and *Multiple Display Monitors*.

lpszOutput

This parameter is ignored for Win32-based applications, and should be set to NULL. It is provided only for compatibility with 16-bit Windows. For more information, see the Remarks section.

lpInitData

[in] Pointer to a **DEVMODE** structure containing device-specific initialization data for the device driver. The **DocumentProperties** function retrieves this structure filled in for a specified device. The *lpInitData* parameter must be NULL if the device driver is to use the default initialization (if any) specified by the user.

Windows NT/2000: If *lpszDriver* is DISPLAY, *lpInitData* must be NULL or a pointer to a valid **DEVMODE** structure for the display device. This structure can be initialized using the **EnumDisplaySettings** function. If *lpInitData* is NULL, then the display device's current **DEVMODE** is used.

Return Values

If the function succeeds, the return value is the handle to a DC for the specified device.

If the function fails, the return value is NULL. The function will return NULL for a **DEVMODE** structure other that the current **DEVMODE**.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Applications written for 16-bit versions of Windows used the *lpszOutput* parameter to specify a port name or to print to a file. Win32-based applications do not need to specify a port name. Win32-based applications can print to a file by calling the **StartDoc** function with a **DOCINFO** structure whose **lpszOutput** member specifies the path of the output file name.

When you no longer need the DC, call the **DeleteDC** function.

ICM: To enable ICM, set the **dmICMMethod** member of the **DEVMODE** structure (pointed to by the *pInitData* parameter) to the appropriate value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **Multiple Display Monitors**, **DeleteDC**, **DEVMODE**, **EnumDisplayDevices**, **DOCINFO**, **DocumentProperties**, **StartDoc**

CreateIC

The **CreateIC** function creates an information context for the specified device. The information context provides a fast way to get information about the device without creating a device context (DC). However, GDI drawing functions cannot accept a handle to an information context.

```
HDC CreateIC(
  LPCTSTR lpszDriver,        // driver name
  LPCTSTR lpszDevice,        // device name
  LPCTSTR lpszOutput,        // port or file name
  CONST DEVMODE *lpdvmInit   // optional initialization data
);
```

Parameters

lpszDriver
[in] Pointer to a null-terminated character string that specifies the name of the device driver (for example, Epson).

lpszDevice
[in] Pointer to a null-terminated character string that specifies the name of the specific output device being used, as shown by the Print Manager (for example, Epson FX-80). It is not the printer model name. The *lpszDevice* parameter must be used.

lpszOutput

This parameter is ignored for Win32-based applications, and should be set to NULL. It is provided only for compatibility with 16-bit Windows. For more information, see the Remarks section.

lpdvmInit

[in] Pointer to a **DEVMODE** structure containing device-specific initialization data for the device driver. The **DocumentProperties** function retrieves this structure filled in for a specified device. The *lpdvmInit* parameter must be NULL if the device driver is to use the default initialization (if any) specified by the user.

Return Values

If the function succeeds, the return value is the handle to an information context.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Applications written for 16-bit versions of Windows used the *lpszOutput* parameter to specify a port name or to print to a file. Win32-based applications do not need to specify a port name.

When you no longer need the information DC, call the **DeleteDC** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **DeleteDC**, **DocumentProperties**, **DEVMODE**, **GetDeviceCaps**

DeleteDC

The **DeleteDC** function deletes the specified device context (DC).

```
BOOL DeleteDC(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application must not delete a DC whose handle was obtained by calling the **GetDC** function. Instead, it must call the **ReleaseDC** function to free the DC.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **CreateDC**, **GetDC**, **ReleaseDC**

DeleteObject

The **DeleteObject** function deletes a logical pen, brush, font, bitmap, region, or palette, freeing all system resources associated with the object. After the object is deleted, the specified handle is no longer valid.

```
BOOL DeleteObject(
  HGDIOBJ hObject   // handle to graphic object
);
```

Parameters

hObject

[in] Handle to a logical pen, brush, font, bitmap, region, or palette.

Return Values

If the function succeeds, the return value is nonzero.

If the specified handle is not valid or is currently selected into a DC, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Do not delete a drawing object (pen or brush) while it is still selected into a DC.

When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted independently.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **SelectObject**

DrawEscape

The **DrawEscape** function accesses the drawing capabilities of a video display that are not directly available through the graphical device interface (GDI).

```
int DrawEscape(
  HDC hdc,            // handle to DC
  int nEscape,        // escape function
  int cbInput,        // size of structure for input
  LPCSTR lpszInData   // structure for input
);
```

Parameters

hdc
[in] Handle to the DC for the specified video display.

nEscape
[in] Specifies the escape function to be performed.

cbInput
[in] Specifies the number of bytes of data pointed to by the *lpszInData* parameter.

lpszInData
[in] Pointer to the input structure required for the specified escape.

Return Values

The return value specifies the outcome of the function. It is greater than zero if the function is successful, except for the QUERYESCSUPPORT draw escape, which checks for implementation only. The return value is zero if the escape is not implemented. The return value is less than zero if an error occurred.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When an application calls the **DrawEscape** function, the data identified by *cbInput* and *lpszInData* is passed directly to the specified display driver.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions

EnumDisplayDevices

The **EnumDisplayDevices** function lets you obtain information about the display devices in a system.

```
BOOL EnumDisplayDevices(
  PVOID Unused,                      // reserved
  DWORD iDevNum,                     // display device
  PDISPLAY_DEVICE lpDisplayDevice,   // device information
  DWORD dwFlags                      // reserved
);
```

Parameters

Unused

This parameter is not used and should be set to NULL.

iDevNum

[in] Index value that specifies the display device of interest.

The operating system identifies each display device with an index value. The index values are consecutive integers, starting at 0. If a system has three display devices, for example, they are specified by the index values 0, 1, and 2.

lpDisplayDevice

[out] Pointer to a **DISPLAY_DEVICE** structure that receives information about the display device specified by *iDevNum.*

Before calling **EnumDisplayDevices**, you must initialize the **cb** member of **DISPLAY_DEVICE** to the size, in bytes, of **DISPLAY_DEVICE**.

dwFlags

This parameter is currently not used and should be set to zero.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function fails if *iDevNum* is greater than the largest device index.

Remarks

In order to query all display devices in the system, call this function in a loop, starting with *iDevNum* set to 0, and incrementing *iDevNum* until the function fails. And in order to query all display devices in the desktop, the caller should filter out the display devices which do not have the DISPLAY_DEVICE_ATTACHED_TO_DESKTOP flag in the **DISPLAY_DEVICE** structure.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettings**, **ChangeDisplaySettingsEx**, **CreateDC**, **DEVMODE**, **DISPLAY_DEVICE,** **EnumDisplaySettings**

EnumDisplaySettings

The **EnumDisplaySettings** function obtains information about one of a display device's graphics modes. You can obtain information for all of a display device's graphics modes by making a series of calls to this function.

```
BOOL EnumDisplaySettings(
  LPCTSTR lpszDeviceName,   // display device
  DWORD iModeNum,           // graphics mode
  LPDEVMODE lpDevMode       // graphics mode settings
);
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device whose graphics mode the function will obtain information about.

This parameter can be NULL. A NULL value specifies the current display device on the computer that the calling thread is running on.

If *lpszDeviceName* is not NULL, the string must be of the form **\\.\Display*X***, where ***X*** can have the values 1, 2, or 3.

Windows 95/98: *lpszDeviceName* must be NULL.

iModeNum

[in] Specifies the type of information to retrieve. This value can be a graphics mode index or one of the following values:

Value	Meaning
ENUM_CURRENT_SETTINGS	Retrieve the current settings for the display device.
ENUM_REGISTRY_SETTINGS	Retrieve the settings for the display device that are currently stored in the registry.

Graphics mode indexes start at zero. To obtain information for all of a display device's graphics modes, make a series of calls to **EnumDisplaySettings**, as follows: Set *iModeNum* to zero for the first call, and increment *iModeNum* by one for each subsequent call. Continue calling the function until the return value is zero.

When you call **EnumDisplaySettings** with *iModeNum* set to zero, the operating system initializes and caches information about the display device. When you call **EnumDisplaySettings** with *iModeNum* set to a non-zero value, the function returns the information that was cached the last time the function was called with *iModeNum* set to zero.

lpDevMode

[out] Pointer to a **DEVMODE** structure into which the function stores information about the specified graphics mode. Before calling **EnumDisplaySettings**, set the **dmSize** member to **sizeof(DEVMODE)**, and set the **dmDriverExtra** member to indicate the size, in bytes, of the additional space available to receive private driver-data.

The **EnumDisplaySettings** function sets values for the following five **DEVMODE** members:

- **dmBitsPerPel**
- **dmPelsWidth**
- **dmPelsHeight**
- **dmDisplayFlags**
- **dmDisplayFrequency**

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The function fails if *iModeNum* is greater than the index of the display device's last graphics mode. As noted in the description of the *iModeNum* parameter, you can use this behavior to enumerate all of a display device's graphics modes.

! Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettings**, **ChangeDisplaySettingsEx**, **CreateDC**, **CreateDesktop**, **DEVMODE**, **EnumDisplayDevices**

EnumDisplaySettingsEx

The **EnumDisplaySettingsEx** function obtains information about one of the graphics modes for a display device. You can obtain information for all of the graphics modes for a display device by making a series of calls to this function.

This function differs from **EnumDisplaySettings** in that there is a *dwFlags* parameter.

```
BOOL EnumDisplaySettingsEx(
  LPCTSTR lpszDeviceName,   // display device
  DWORD iModeNum,           // graphics mode
  LPDEVMODE lpDevMode       // graphics mode settings
  DWORD dwFlags             // options
);
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device about which graphics mode the function will obtain information.

This parameter can be NULL. A NULL value specifies the current display device on the computer that the calling thread is running on.

If *lpszDeviceName* is not NULL, the string must be of the form **\\.\Display*X***, where ***X*** can have the values 1, 2, or 3.

iModeNum

[in] Indicates the type of information to retrieve. This value can be a graphics mode index or one of the following values:

Value	Meaning
ENUM_CURRENT_SETTINGS	Retrieve the current settings for the display device.
ENUM_REGISTRY_SETTINGS	Retrieve the settings for the display device that are currently stored in the registry.

Graphics mode indexes start at zero. To obtain information for all of a display device's graphics modes, make a series of calls to **EnumDisplaySettingsEx**, as follows: Set *iModeNum* to zero for the first call, and increment *iModeNum* by one for each subsequent call. Continue calling the function until the return value is zero.

When you call **EnumDisplaySettingsEx** with *iModeNum* set to zero, the operating system initializes and caches information about the display device. When you call **EnumDisplaySettingsEx** with *iModeNum* set to a nonzero value, the function returns the information that was cached the last time the function was called with *iModeNum* set to zero.

lpDevMode

[out] Pointer to a **DEVMODE** structure into which the function stores information about the specified graphics mode. Before calling **EnumDisplaySettingsEx**, set the **dmSize** member to **sizeof(DEVMODE)**, and set the **dmDriverExtra** member to indicate the size, in bytes, of the additional space available to receive private driver-data.

The **EnumDisplaySettingsEx** function sets values for the following five **DEVMODE** members:

- **dmBitsPerPel**
- **dmPelsWidth**
- **dmPelsHeight**
- **dmDisplayFlags**
- **dmDisplayFrequency**

dwFlags

[in] This parameter can be the following value:

Value	Meaning
EDS_RAWMODE	If set, the function will return all graphics modes reported by the adapter driver, regardless of monitor capabilities. Otherwise, it will only return modes that are compatible with current monitors.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

To get extended error information, call **GetLastError**.

Remarks

The function fails if *iModeNum* is greater than the index of the display device's last graphics mode. As noted in the description of the *iModeNum* parameter, you can use this behavior to enumerate all of a display device's graphics modes.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettings**, **ChangeDisplaySettingsEx**, **CreateDC**, **CreateDesktop**, **DEVMODE**, **EnumDisplaySettings**, **EnumDisplayDevices**

EnumObjects

The **EnumObjects** function enumerates the pens or brushes available for the specified device context (DC). This function calls the application-defined callback function once for each available object, supplying data describing that object. **EnumObjects** continues calling the callback function until the callback function returns zero or until all of the objects have been enumerated.

```
int EnumObjects(
  HDC hdc,                    // handle to DC
  int nObjectType,            // object-type identifier
  GOBJENUMPROC lpObjectFunc,  // callback function
  LPARAM lParam               // application-supplied data
);
```

Parameters

hdc

[in] Handle to the DC.

nObjectType

[in] Specifies the object type. This parameter can be OBJ_BRUSH or OBJ_PEN.

lpObjectFunc

[in] Pointer to the application-defined callback function. For more information about the callback function, see ***EnumObjectsProc***.

lParam

[in] Pointer to the application-defined data. The data is passed to the callback function along with the object information.

Return Values

If the function succeeds, the function returns the last value returned by the callback function. Its meaning is user-defined.

If there are too many objects to enumerate, the function returns –1. In this case, the callback function is not called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **EnumObjectsProc**, **GetObject**

EnumObjectsProc

The **EnumObjectsProc** function is an application-defined callback function used with the **EnumObjects** function. It is used to process the object data. The **GOBJENUMPROC** type defines a pointer to this callback function. **EnumObjectsProc** is a placeholder for the application-defined function name.

```
int CALLBACK EnumObjectsProc(
  LPVOID lpLogObject,  // object attributes
  LPARAM lpData        // application-defined data
);
```

Parameters

lpLogObject

[in] Pointer to a **LOGPEN** or **LOGBRUSH** structure describing the attributes of the object.

lpData

[in] Pointer to the application-defined data passed by the **EnumObjects** function.

Return Values

To continue enumeration, the callback function must return a nonzero value. This value is user-defined.

To stop enumeration, the callback function must return zero.

Remarks

An application must register this function by passing its address to the **EnumObjects** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Device Contexts Overview, Device Context Functions, **EnumObjects**, **GlobalAlloc**, **GlobalLock**, **LOGPEN**, **LOGBRUSH**

GetCurrentObject

The **GetCurrentObject** function obtains a handle to an object of the specified type that has been selected into the specified device context (DC).

```
HGDIOBJ GetCurrentObject(
  HDC hdc,            // handle to DC
  UINT uObjectType    // object type
);
```

Parameters

hdc

[in] Handle to the DC.

uObjectType

[in] Specifies the object type to be queried. This parameter can be one of the following values:

Value	Meaning
OBJ_BITMAP	Returns the current selected bitmap.
OBJ_BRUSH	Returns the current selected brush.
OBJ_COLORSPACE	Returns the current color space.
OBJ_FONT	Returns the current selected font.
OBJ_PAL	Returns the current selected palette.
OBJ_PEN	Returns the current selected pen.

Return Values

If the function succeeds, the return value is a handle to the specified object.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can use the **GetCurrentObject** and **GetObject** functions to retrieve descriptions of the graphic objects currently selected into the specified DC.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **DeleteObject**, **GetObject**, **SelectObject**, **CreateColorSpace**

GetDC

The **GetDC** function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

The **GetDCEx** function is an extension to **GetDC**, which gives an application more control over how and whether clipping occurs in the client area.

```
HDC GetDC(
  HWND hWnd   // handle to window
);
```

Parameters

hWnd

[in] Handle to the window whose DC is to be retrieved. If this value is NULL, **GetDC** retrieves the DC for the entire screen.

Windows 98, Windows 2000: If this parameter is NULL, **GetDC** retrieves the DC for the primary display monitor. To get the DC for other display monitors, use the **EnumDisplayMonitors** and **CreateDC** functions.

Return Values

If the function succeeds, the return value is a handle to the DC for the specified window's client area.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **GetDC** function retrieves a common, class, or private DC depending on the class style specified for the specified window. For common DCs, **GetDC** assigns default attributes to the DC each time it is retrieved. For class and private DCs, **GetDC** leaves the previously assigned attributes unchanged.

After painting with a common DC, the **ReleaseDC** function must be called to release the DC. Class and private DCs do not have to be released. The number of DCs is limited only by available memory.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Device Contexts Overview, Device Context Functions, **GetDCEx**, **ReleaseDC**, **GetWindowDC**

GetDCBrushColor

The **GetDCBrushColor** function retrieves a handle to the device context (DC) whose brush color is to be returned.

```
COLORREF GetDCBrushColor(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
[in] Handle to the DC whose brush color is to be returned.

Return Values

If the function succeeds, the return value is a **COLORREF** which is a color reference for the current DC brush color.

If the function fails, the return value is CLR_INVALID.

Remarks

The **GetDCBrushColor** function returns the previous DC_BRUSH color even if the stock object DC_BRUSH is not selected in the DC. For information on setting the brush color, see ***SetDCBrushColor***.

ICM: Color management is performed if ICM is enabled.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

See Also

Device Contexts Overview, Device Context Functions, **SetDCBrushColor**, **COLORREF**, About Device Contexts

GetDCEx

The **GetDCEx** function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

This function is an extension to the **GetDC** function, which gives an application more control over how and whether clipping occurs in the client area.

```
HDC GetDCEx(
  HWND hWnd,       // handle to window
  HRGN hrgnClip,   // handle to clipping region
  DWORD flags      // creation options
);
```

Parameters

hWnd

[in] Handle to the window whose DC is to be retrieved. If this value is NULL, **GetDCEx** retrieves the DC for the entire screen.

Windows 98, Windows 2000: If this parameter is NULL, **GetDCEx** retrieves the DC for the primary display monitor. To get the DC for other display monitors, use the **EnumDisplayMonitors** and **CreateDC** functions.

hrgnClip

[in] Specifies a clipping region that may be combined with the visible region of the DC. If the value of *flags* is DCX_INTERSECTRGN or DCX_EXCLUDERGN, then the operating system assumes ownership of the region and will automatically delete it when it is no longer needed. In this case, applications should not use the region—not even delete it—after a successful call to **GetDCEx**.

flags

[in] Specifies how the DC is created. This parameter can be one or more of the following values:

Value	Meaning
DCX_WINDOW	Returns a DC that corresponds to the window rectangle rather than the client rectangle.
DCX_CACHE	Returns a DC from the cache, rather than the OWNDC or CLASSDC window. Essentially overrides CS_OWNDC and CS_CLASSDC.

(continued)

(continued)

Value	Meaning
DCX_PARENTCLIP	Uses the visible region of the parent window. The parent's WS_CLIPCHILDREN and CS_PARENTDC style bits are ignored. The origin is set to the upper-left corner of the window identified by *hWnd*.
DCX_CLIPSIBLINGS	Excludes the visible regions of all sibling windows above the window identified by *hWnd*.
DCX_CLIPCHILDREN	Excludes the visible regions of all child windows below the window identified by *hWnd*.
DCX_NORESETATTRS	Does not reset the attributes of this DC to the default attributes when this DC is released.
DCX_LOCKWINDOWUPDATE	Allows drawing even if there is a **LockWindowUpdate** call in effect that would otherwise exclude this window. Used for drawing during tracking.
DCX_EXCLUDERGN	The clipping region identified by *hrgnClip* is excluded from the visible region of the returned DC.
DCX_INTERSECTRGN	The clipping region identified by *hrgnClip* is intersected with the visible region of the returned DC.
DCX_VALIDATE	When specified with DCX_INTERSECTUPDATE, causes the DC to be completely validated. Using this function with both DCX_INTERSECTUPDATE and DCX_VALIDATE is identical to using the **BeginPaint** function.

Return Values

If the function succeeds, the return value is the handle to the DC for the specified window.

If the function fails, the return value is NULL. An invalid value for the *hWnd* parameter will cause the function to fail.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Unless the display DC belongs to a window class, the **ReleaseDC** function must be called to release the DC after painting. Because only five common DCs are available at any time, failure to release a DC can prevent other applications from accessing one.

The function returns a handle to a DC that belongs to the window's class if CS_CLASSDC, CS_OWNDC or CS_PARENTDC was specified as a style in the **WNDCLASS** structure when the class was registered.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.10 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **BeginPaint**, **GetWindowDC**, **ReleaseDC**, **WNDCLASS**

GetDCOrgEx

The **GetDCOrgEx** function obtains the final translation origin for a specified device context (DC). The final translation origin specifies an offset that the system uses to translate device coordinates into client coordinates (for coordinates in an application's window).

```
BOOL GetDCOrgEx(
  HDC hdc,           // handle to a DC
  LPPOINT lpPoint    // translation origin
);
```

Parameters

hdc

[in] Handle to the DC whose final translation origin is to be retrieved.

lpPoint

[out] Pointer to a **POINT** structure that receives the final translation origin, in device coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The final translation origin is relative to the physical origin of the screen.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **CreateIC**, **POINT**

GetDCPenColor

The **GetDCPenColor** function sets the current device context (DC) pen color to the specified color value. **GetDCPenColor** will return the nearest physical color if the device cannot represent the specified color value.

```
COLORREF GetDCPenColor(
  HDC hdc  // handle to DC
);
```

Parameters

hdc
[in] Handle to the DC whose brush color is to be returned.

Return Values

If the function succeeds, the return value is a color reference (**COLORREF**) for the previous DC pen color.

If the function fails, the return value is CLR_INVALID.

Remarks

The **GetDCPenColor** function will return the previous DC_PEN color even if the stock object DC_PEN is not selected in the DC. *See Setting the Pen or Brush Color* and ***SetDCPenColor*** for more information.

ICM: Color management is performed if ICM is enabled.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

+ See Also

Device Contexts Overview, Device Context Functions, **COLORREF**

If the specified handle is not valid or is currently selected into a DC, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Do not delete a drawing object (pen or brush) while it is still selected into a DC.

When a pattern brush is deleted, the bitmap associated with the brush is not deleted. The bitmap must be deleted independently.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **SelectObject**

DrawEscape

The **DrawEscape** function accesses the drawing capabilities of a video display that are not directly available through the graphical device interface (GDI).

```
int DrawEscape(
  HDC hdc,           // handle to DC
  int nEscape,       // escape function
  int cbInput,       // size of structure for input
  LPCSTR lpszInData  // structure for input
);
```

Parameters

hdc
[in] Handle to the DC for the specified video display.

nEscape
[in] Specifies the escape function to be performed.

cbInput
[in] Specifies the number of bytes of data pointed to by the *lpszInData* parameter.

lpszInData
[in] Pointer to the input structure required for the specified escape.

Return Values

The return value specifies the outcome of the function. It is greater than zero if the function is successful, except for the QUERYESCSUPPORT draw escape, which checks for implementation only. The return value is zero if the escape is not implemented. The return value is less than zero if an error occurred.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When an application calls the **DrawEscape** function, the data identified by *cbInput* and *lpszInData* is passed directly to the specified display driver.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Device Contexts Overview, Device Context Functions

EnumDisplayDevices

The **EnumDisplayDevices** function lets you obtain information about the display devices in a system.

```
BOOL EnumDisplayDevices(
  PVOID Unused,                        // reserved
  DWORD iDevNum,                       // display device
  PDISPLAY_DEVICE lpDisplayDevice,     // device information
  DWORD dwFlags                        // reserved
);
```

Parameters

Unused

This parameter is not used and should be set to NULL.

iDevNum

[in] Index value that specifies the display device of interest.

The operating system identifies each display device with an index value. The index values are consecutive integers, starting at 0. If a system has three display devices, for example, they are specified by the index values 0, 1, and 2.

lpDisplayDevice

[out] Pointer to a **DISPLAY_DEVICE** structure that receives information about the display device specified by *iDevNum.*

Before calling **EnumDisplayDevices**, you must initialize the **cb** member of **DISPLAY_DEVICE** to the size, in bytes, of **DISPLAY_DEVICE**.

dwFlags

This parameter is currently not used and should be set to zero.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function fails if *iDevNum* is greater than the largest device index.

Remarks

In order to query all display devices in the system, call this function in a loop, starting with *iDevNum* set to 0, and incrementing *iDevNum* until the function fails. And in order to query all display devices in the desktop, the caller should filter out the display devices which do not have the DISPLAY_DEVICE_ATTACHED_TO_DESKTOP flag in the **DISPLAY_DEVICE** structure.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettings**, **ChangeDisplaySettingsEx**, **CreateDC**, **DEVMODE**, **DISPLAY_DEVICE**, **EnumDisplaySettings**

EnumDisplaySettings

The **EnumDisplaySettings** function obtains information about one of a display device's graphics modes. You can obtain information for all of a display device's graphics modes by making a series of calls to this function.

```
BOOL EnumDisplaySettings(
  LPCTSTR lpszDeviceName,   // display device
  DWORD iModeNum,           // graphics mode
  LPDEVMODE lpDevMode       // graphics mode settings
);
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device whose graphics mode the function will obtain information about.

This parameter can be NULL. A NULL value specifies the current display device on the computer that the calling thread is running on.

If *lpszDeviceName* is not NULL, the string must be of the form **\\.\Display*X***, where ***X*** can have the values 1, 2, or 3.

Windows 95/98: *lpszDeviceName* must be NULL.

iModeNum

[in] Specifies the type of information to retrieve. This value can be a graphics mode index or one of the following values:

Value	Meaning
ENUM_CURRENT_SETTINGS	Retrieve the current settings for the display device.
ENUM_REGISTRY_SETTINGS	Retrieve the settings for the display device that are currently stored in the registry.

Graphics mode indexes start at zero. To obtain information for all of a display device's graphics modes, make a series of calls to **EnumDisplaySettings**, as follows: Set *iModeNum* to zero for the first call, and increment *iModeNum* by one for each subsequent call. Continue calling the function until the return value is zero.

When you call **EnumDisplaySettings** with *iModeNum* set to zero, the operating system initializes and caches information about the display device. When you call **EnumDisplaySettings** with *iModeNum* set to a non-zero value, the function returns the information that was cached the last time the function was called with *iModeNum* set to zero.

lpDevMode

[out] Pointer to a **DEVMODE** structure into which the function stores information about the specified graphics mode. Before calling **EnumDisplaySettings**, set the **dmSize** member to **sizeof(DEVMODE)**, and set the **dmDriverExtra** member to indicate the size, in bytes, of the additional space available to receive private driver-data.

The **EnumDisplaySettings** function sets values for the following five **DEVMODE** members:

- **dmBitsPerPel**
- **dmPelsWidth**
- **dmPelsHeight**
- **dmDisplayFlags**
- **dmDisplayFrequency**

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The function fails if *iModeNum* is greater than the index of the display device's last graphics mode. As noted in the description of the *iModeNum* parameter, you can use this behavior to enumerate all of a display device's graphics modes.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettings**, **ChangeDisplaySettingsEx**, **CreateDC**, **CreateDesktop**, **DEVMODE**, **EnumDisplayDevices**

EnumDisplaySettingsEx

The **EnumDisplaySettingsEx** function obtains information about one of the graphics modes for a display device. You can obtain information for all of the graphics modes for a display device by making a series of calls to this function.

This function differs from **EnumDisplaySettings** in that there is a *dwFlags* parameter.

```
BOOL EnumDisplaySettingsEx(
  LPCTSTR lpszDeviceName,   // display device
  DWORD iModeNum,           // graphics mode
  LPDEVMODE lpDevMode       // graphics mode settings
  DWORD dwFlags             // options
);
```

Parameters

lpszDeviceName

[in] Pointer to a null-terminated string that specifies the display device about which graphics mode the function will obtain information.

This parameter can be NULL. A NULL value specifies the current display device on the computer that the calling thread is running on.

If *lpszDeviceName* is not NULL, the string must be of the form **\\.\Display*X***, where ***X*** can have the values 1, 2, or 3.

iModeNum

[in] Indicates the type of information to retrieve. This value can be a graphics mode index or one of the following values:

Value	Meaning
ENUM_CURRENT_SETTINGS	Retrieve the current settings for the display device.
ENUM_REGISTRY_SETTINGS	Retrieve the settings for the display device that are currently stored in the registry.

Graphics mode indexes start at zero. To obtain information for all of a display device's graphics modes, make a series of calls to **EnumDisplaySettingsEx**, as follows: Set *iModeNum* to zero for the first call, and increment *iModeNum* by one for each subsequent call. Continue calling the function until the return value is zero.

When you call **EnumDisplaySettingsEx** with *iModeNum* set to zero, the operating system initializes and caches information about the display device. When you call **EnumDisplaySettingsEx** with *iModeNum* set to a nonzero value, the function returns the information that was cached the last time the function was called with *iModeNum* set to zero.

lpDevMode

[out] Pointer to a **DEVMODE** structure into which the function stores information about the specified graphics mode. Before calling **EnumDisplaySettingsEx**, set the **dmSize** member to **sizeof**(**DEVMODE**), and set the **dmDriverExtra** member to indicate the size, in bytes, of the additional space available to receive private driver-data.

The **EnumDisplaySettingsEx** function sets values for the following five **DEVMODE** members:

- **dmBitsPerPel**
- **dmPelsWidth**
- **dmPelsHeight**
- **dmDisplayFlags**
- **dmDisplayFrequency**

dwFlags

[in] This parameter can be the following value:

Value	Meaning
EDS_RAWMODE	If set, the function will return all graphics modes reported by the adapter driver, regardless of monitor capabilities. Otherwise, it will only return modes that are compatible with current monitors.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

To get extended error information, call **GetLastError**.

Remarks

The function fails if *iModeNum* is greater than the index of the display device's last graphics mode. As noted in the description of the *iModeNum* parameter, you can use this behavior to enumerate all of a display device's graphics modes.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Device Contexts Overview, Device Context Functions, **ChangeDisplaySettings**, **ChangeDisplaySettingsEx**, **CreateDC**, **CreateDesktop**, **DEVMODE**, **EnumDisplaySettings**, **EnumDisplayDevices**

EnumObjects

The **EnumObjects** function enumerates the pens or brushes available for the specified device context (DC). This function calls the application-defined callback function once for each available object, supplying data describing that object. **EnumObjects** continues calling the callback function until the callback function returns zero or until all of the objects have been enumerated.

```
int EnumObjects(
  HDC hdc,                      // handle to DC
  int nObjectType,              // object-type identifier
  GOBJENUMPROC lpObjectFunc,    // callback function
  LPARAM lParam                 // application-supplied data
);
```

Parameters

hdc

[in] Handle to the DC.

nObjectType

[in] Specifies the object type. This parameter can be OBJ_BRUSH or OBJ_PEN.

lpObjectFunc

[in] Pointer to the application-defined callback function. For more information about the callback function, see ***EnumObjectsProc***.

lParam

[in] Pointer to the application-defined data. The data is passed to the callback function along with the object information.

Return Values

If the function succeeds, the function returns the last value returned by the callback function. Its meaning is user-defined.

If there are too many objects to enumerate, the function returns –1. In this case, the callback function is not called.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **EnumObjectsProc**, **GetObject**

EnumObjectsProc

The **EnumObjectsProc** function is an application-defined callback function used with the **EnumObjects** function. It is used to process the object data. The **GOBJENUMPROC** type defines a pointer to this callback function. **EnumObjectsProc** is a placeholder for the application-defined function name.

```
int CALLBACK EnumObjectsProc(
  LPVOID lpLogObject,  // object attributes
  LPARAM lpData        // application-defined data
);
```

Parameters

lpLogObject
[in] Pointer to a **LOGPEN** or **LOGBRUSH** structure describing the attributes of the object.

lpData
[in] Pointer to the application-defined data passed by the **EnumObjects** function.

Return Values

To continue enumeration, the callback function must return a nonzero value. This value is user-defined.

To stop enumeration, the callback function must return zero.

Remarks

An application must register this function by passing its address to the **EnumObjects** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Device Contexts Overview, Device Context Functions, **EnumObjects**, **GlobalAlloc**, **GlobalLock**, **LOGPEN**, **LOGBRUSH**

GetCurrentObject

The **GetCurrentObject** function obtains a handle to an object of the specified type that has been selected into the specified device context (DC).

```
HGDIOBJ GetCurrentObject(
  HDC hdc,           // handle to DC
  UINT uObjectType   // object type
);
```

Parameters

hdc

[in] Handle to the DC.

uObjectType

[in] Specifies the object type to be queried. This parameter can be one of the following values:

Value	Meaning
OBJ_BITMAP	Returns the current selected bitmap.
OBJ_BRUSH	Returns the current selected brush.
OBJ_COLORSPACE	Returns the current color space.
OBJ_FONT	Returns the current selected font.
OBJ_PAL	Returns the current selected palette.
OBJ_PEN	Returns the current selected pen.

Return Values

If the function succeeds, the return value is a handle to the specified object.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can use the **GetCurrentObject** and **GetObject** functions to retrieve descriptions of the graphic objects currently selected into the specified DC.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **DeleteObject**, **GetObject**, **SelectObject**, **CreateColorSpace**

GetDC

The **GetDC** function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

The **GetDCEx** function is an extension to **GetDC**, which gives an application more control over how and whether clipping occurs in the client area.

```
HDC GetDC(
  HWND hWnd   // handle to window
);
```

Parameters

hWnd

[in] Handle to the window whose DC is to be retrieved. If this value is NULL, **GetDC** retrieves the DC for the entire screen.

Windows 98, Windows 2000: If this parameter is NULL, **GetDC** retrieves the DC for the primary display monitor. To get the DC for other display monitors, use the **EnumDisplayMonitors** and **CreateDC** functions.

Return Values

If the function succeeds, the return value is a handle to the DC for the specified window's client area.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **GetDC** function retrieves a common, class, or private DC depending on the class style specified for the specified window. For common DCs, **GetDC** assigns default attributes to the DC each time it is retrieved. For class and private DCs, **GetDC** leaves the previously assigned attributes unchanged.

After painting with a common DC, the **ReleaseDC** function must be called to release the DC. Class and private DCs do not have to be released. The number of DCs is limited only by available memory.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Device Contexts Overview, Device Context Functions, **GetDCEx**, **ReleaseDC**, **GetWindowDC**

GetDCBrushColor

The **GetDCBrushColor** function retrieves a handle to the device context (DC) whose brush color is to be returned.

```
COLORREF GetDCBrushColor(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
[in] Handle to the DC whose brush color is to be returned.

Return Values

If the function succeeds, the return value is a **COLORREF** which is a color reference for the current DC brush color.

If the function fails, the return value is CLR_INVALID.

Remarks

The **GetDCBrushColor** function returns the previous DC_BRUSH color even if the stock object DC_BRUSH is not selected in the DC. For information on setting the brush color, see ***SetDCBrushColor***.

ICM: Color management is performed if ICM is enabled.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

See Also

Device Contexts Overview, Device Context Functions, **SetDCBrushColor**, **COLORREF**, About Device Contexts

GetDCEx

The **GetDCEx** function retrieves a handle to a display device context (DC) for the client area of a specified window or for the entire screen. You can use the returned handle in subsequent GDI functions to draw in the DC.

This function is an extension to the **GetDC** function, which gives an application more control over how and whether clipping occurs in the client area.

```
HDC GetDCEx(
  HWND hWnd,       // handle to window
  HRGN hrgnClip,   // handle to clipping region
  DWORD flags      // creation options
);
```

Parameters

hWnd

[in] Handle to the window whose DC is to be retrieved. If this value is NULL, **GetDCEx** retrieves the DC for the entire screen.

Windows 98, Windows 2000: If this parameter is NULL, **GetDCEx** retrieves the DC for the primary display monitor. To get the DC for other display monitors, use the **EnumDisplayMonitors** and **CreateDC** functions.

hrgnClip

[in] Specifies a clipping region that may be combined with the visible region of the DC. If the value of *flags* is DCX_INTERSECTRGN or DCX_EXCLUDERGN, then the operating system assumes ownership of the region and will automatically delete it when it is no longer needed. In this case, applications should not use the region—not even delete it—after a successful call to **GetDCEx**.

flags

[in] Specifies how the DC is created. This parameter can be one or more of the following values:

Value	Meaning
DCX_WINDOW	Returns a DC that corresponds to the window rectangle rather than the client rectangle.
DCX_CACHE	Returns a DC from the cache, rather than the OWNDC or CLASSDC window. Essentially overrides CS_OWNDC and CS_CLASSDC.

(continued)

(continued)

Value	Meaning
DCX_PARENTCLIP	Uses the visible region of the parent window. The parent's WS_CLIPCHILDREN and CS_PARENTDC style bits are ignored. The origin is set to the upper-left corner of the window identified by *hWnd*.
DCX_CLIPSIBLINGS	Excludes the visible regions of all sibling windows above the window identified by *hWnd*.
DCX_CLIPCHILDREN	Excludes the visible regions of all child windows below the window identified by *hWnd*.
DCX_NORESETATTRS	Does not reset the attributes of this DC to the default attributes when this DC is released.
DCX_LOCKWINDOWUPDATE	Allows drawing even if there is a **LockWindowUpdate** call in effect that would otherwise exclude this window. Used for drawing during tracking.
DCX_EXCLUDERGN	The clipping region identified by *hrgnClip* is excluded from the visible region of the returned DC.
DCX_INTERSECTRGN	The clipping region identified by *hrgnClip* is intersected with the visible region of the returned DC.
DCX_VALIDATE	When specified with DCX_INTERSECTUPDATE, causes the DC to be completely validated. Using this function with both DCX_INTERSECTUPDATE and DCX_VALIDATE is identical to using the **BeginPaint** function.

Return Values

If the function succeeds, the return value is the handle to the DC for the specified window.

If the function fails, the return value is NULL. An invalid value for the *hWnd* parameter will cause the function to fail.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Unless the display DC belongs to a window class, the **ReleaseDC** function must be called to release the DC after painting. Because only five common DCs are available at any time, failure to release a DC can prevent other applications from accessing one.

The function returns a handle to a DC that belongs to the window's class if CS_CLASSDC, CS_OWNDC or CS_PARENTDC was specified as a style in the **WNDCLASS** structure when the class was registered.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.10 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **BeginPaint**, **GetWindowDC**, **ReleaseDC**, **WNDCLASS**

GetDCOrgEx

The **GetDCOrgEx** function obtains the final translation origin for a specified device context (DC). The final translation origin specifies an offset that the system uses to translate device coordinates into client coordinates (for coordinates in an application's window).

```
BOOL GetDCOrgEx(
  HDC hdc,            // handle to a DC
  LPPOINT lpPoint     // translation origin
);
```

Parameters

hdc
[in] Handle to the DC whose final translation origin is to be retrieved.

lpPoint
[out] Pointer to a **POINT** structure that receives the final translation origin, in device coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The final translation origin is relative to the physical origin of the screen.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **CreateIC**, **POINT**

GetDCPenColor

The **GetDCPenColor** function sets the current device context (DC) pen color to the specified color value. **GetDCPenColor** will return the nearest physical color if the device cannot represent the specified color value.

```
COLORREF GetDCPenColor(
  HDC hdc  // handle to DC
);
```

Parameters

hdc

[in] Handle to the DC whose brush color is to be returned.

Return Values

If the function succeeds, the return value is a color reference (**COLORREF**) for the previous DC pen color.

If the function fails, the return value is CLR_INVALID.

Remarks

The **GetDCPenColor** function will return the previous DC_PEN color even if the stock object DC_PEN is not selected in the DC. *See Setting the Pen or Brush Color* and ***SetDCPenColor*** for more information.

ICM: Color management is performed if ICM is enabled.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Included as a resource in msimg32.dll.

See Also

Device Contexts Overview, Device Context Functions, **COLORREF**

GetDeviceCaps

The **GetDeviceCaps** function retrieves device-specific information about the specified device.

```
int GetDeviceCaps(
  HDC hdc,        // handle to DC
  int nIndex      // index of capability
);
```

Parameters

hdc

[in] Handle to the DC.

nIndex

[in] Specifies the item to return. This parameter can be one of the following values:

Index	Meaning	
DRIVERVERSION	The device driver version.	
TECHNOLOGY	Device technology. It can be any one of the following values:	
	DT_PLOTTER	Vector plotter
	DT_RASDISPLAY	Raster display
	DT_RASPRINTER	Raster printer
	DT_RASCAMERA	Raster camera
	DT_CHARSTREAM	Character stream
	DT_METAFILE	Metafile
	DT_DISPFILE	Display file
	If the *hdc* parameter is a handle to the DC of an enhanced metafile, the device technology is that of the referenced device as specified to the **CreateEnhMetaFile** function. To determine whether it is an enhanced metafile DC, use the **GetObjectType** function.	
HORZSIZE	Width, in millimeters, of the physical screen.	
VERTSIZE	Height, in millimeters, of the physical screen.	
HORZRES	Width, in pixels, of the screen.	
VERTRES	Height, in raster lines, of the screen.	
LOGPIXELSX	Number of pixels per logical inch along the screen width. In a system with multiple display monitors, this value is the same for all monitors.	
LOGPIXELSY	Number of pixels per logical inch along the screen height. In a system with multiple display monitors, this value is the same for all monitors.	
BITSPIXEL	Number of adjacent color bits for each pixel.	
PLANES	Number of color planes.	

(continued)

(continued)

Index	Meaning
NUMBRUSHES	Number of device-specific brushes.
NUMPENS	Number of device-specific pens.
NUMFONTS	Number of device-specific fonts.
NUMCOLORS	Number of entries in the device's color table, if the device has a color depth of no more than 8 bits per pixel. For devices with greater color depths, –1 is returned.
ASPECTX	Relative width of a device pixel used for line drawing.
ASPECTY	Relative height of a device pixel used for line drawing.
ASPECTXY	Diagonal width of the device pixel used for line drawing.
PDEVICESIZE	Reserved.
CLIPCAPS	Flag that indicates the clipping capabilities of the device. If the device can clip to a rectangle, it is 1. Otherwise, it is 0.
SIZEPALETTE	Number of entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver is compatible with 16-bit Windows.
NUMRESERVED	Number of reserved entries in the system palette. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver is compatible with 16-bit Windows.
COLORRES	Actual color resolution of the device, in bits per pixel. This index is valid only if the device driver sets the RC_PALETTE bit in the RASTERCAPS index and is available only if the driver is compatible with 16-bit Windows.
PHYSICALWIDTH	For printing devices: the width of the physical page, in device units. For example, a printer set to print at 600 dpi on 8.5-x11-inch paper has a physical width value of 5100 device units. Note that the physical page is almost always greater than the printable area of the page, and never smaller.
PHYSICALHEIGHT	For printing devices: the height of the physical page, in device units. For example, a printer set to print at 600 dpi on 8.5-x11-inch paper has a physical height value of 6600 device units. Note that the physical page is almost always greater than the printable area of the page, and never smaller.
PHYSICALOFFSETX	For printing devices: the distance from the left edge of the physical page to the left edge of the printable area, in device units. For example, a printer set to print at 600 dpi on 8.5-x11-inch paper, that cannot print on the leftmost 0.25-inch of paper, has a horizontal physical offset of 150 device units.

Index	Meaning	
PHYSICALOFFSETY	For printing devices: the distance from the top edge of the physical page to the top edge of the printable area, in device units. For example, a printer set to print at 600 dpi on 8.5-x11-inch paper, that cannot print on the topmost 0.5-inch of paper, has a vertical physical offset of 300 device units.	
VREFRESH	**Windows NT/2000:** For display devices: the current vertical refresh rate of the device, in cycles per second (Hz). A vertical refresh rate value of 0 or 1 represents the display hardware's default refresh rate. This default rate is typically set by switches on a display card or computer motherboard, or by a configuration program that does not use Win32 display functions such as **ChangeDisplaySettings**.	
SCALINGFACTORX	Scaling factor for the x-axis of the printer.	
SCALINGFACTORY	Scaling factor for the y-axis of the printer.	
BLTALIGNMENT	**Windows NT/2000:** Preferred horizontal drawing alignment, expressed as a multiple of pixels. For best drawing performance, windows should be horizontally aligned to a multiple of this value. A value of zero indicates that the device is accelerated, and any alignment may be used.	
SHADEBLENDCAPS	**Windows 98/2000:** Value that indicates the shading and blending capabilities of the device.	
	SB_CONST_ALPHA	Handles the **SourceConstantAlpha** member of the **BLENDFUNCTION** structure, which is referenced by the *blendFunction* parameter of the **AlphaBlend** function.
	SB_GRAD_RECT	Capable of doing **GradientFill** rectangles.
	SB_GRAD_TRI	Capable of doing **GradientFill** triangles.
	SB_NONE	Device does not support any of these capabilities.
	SB_PIXEL_ALPHA	Capable of handling per-pixel alpha in **AlphaBlend**.
	SB_PREMULT_ALPHA	Capable of handling premultiplied alpha in **AlphaBlend**.
RASTERCAPS	Value that indicates the raster capabilities of the device, as shown in the following table:	
	RC_BANDING	Requires banding support.
	RC_BITBLT	Capable of transferring bitmaps.
	RC_BITMAP64	Capable of supporting bitmaps larger than 64K.

(continued)

(continued)

Index	Meaning	
	RC_DI_BITMAP	Capable of supporting the **SetDIBits** and **GetDIBits** functions.
	RC_DIBTODEV	Capable of supporting the **SetDIBitsToDevice** function.
	RC_FLOODFILL	Capable of performing flood fills.
	RC_GDI20_OUTPUT	Capable of supporting features of 16-bit Windows 2.0.
	RC_PALETTE	Specifies a palette-based device.
	RC_SCALING	Capable of scaling.
	RC_STRETCHBLT	Capable of performing the **StretchBlt** function.
	RC_STRETCHDIB	Capable of performing the **StretchDIBits** function.
CURVECAPS	Value that indicates the curve capabilities of the device, as shown in the following table:	
	CC_NONE	Device does not support curves.
	CC_CHORD	Device can draw chord arcs.
	CC_CIRCLES	Device can draw circles.
	CC_ELLIPSES	Device can draw ellipses.
	CC_INTERIORS	Device can draw interiors.
	CC_PIE	Device can draw pie wedges.
	CC_ROUNDRECT	Device can draw rounded rectangles.
	CC_STYLED	Device can draw styled borders.
	CC_WIDE	Device can draw wide borders.
	CC_WIDESTYLED	Device can draw borders that are wide and styled.
LINECAPS	Value that indicates the line capabilities of the device, as shown in the following table:	
	LC_NONE	Device does not support lines.
	LC_INTERIORS	Device can draw interiors.
	LC_MARKER	Device can draw a marker.
	LC_POLYLINE	Device can draw a polyline.
	LC_POLYMARKER	Device can draw multiple markers.
	LC_STYLED	Device can draw styled lines.
	LC_WIDE	Device can draw wide lines.

Index	Meaning	
	LC_WIDESTYLED	Device can draw lines that are wide and styled.
POLYGONALCAPS	Value that indicates the polygon capabilities of the device, as shown in the following table:	
	PC_NONE	Device does not support polygons.
	PC_INTERIORS	Device can draw interiors.
	PC_POLYGON	Device can draw alternate-fill polygons.
	PC_RECTANGLE	Device can draw rectangles.
	PC_SCANLINE	Device can draw a single scanline.
	PC_STYLED	Device can draw styled borders.
	PC_WIDE	Device can draw wide borders.
	PC_WIDESTYLED	Device can draw borders that are wide and styled.
	PC_WINDPOLYGON	Device can draw winding-fill polygons.
TEXTCAPS	Value that indicates the text capabilities of the device, as shown in the following table:	
	TC_OP_CHARACTER	Device is capable of character output precision.
	TC_OP_STROKE	Device is capable of stroke output precision.
	TC_CP_STROKE	Device is capable of stroke clip precision.
	TC_CR_90	Device is capable of 90-degree character rotation.
	TC_CR_ANY	Device is capable of any character rotation.
	TC_SF_X_YINDEP	Device can scale independently in the x- and y-directions.
	TC_SA_DOUBLE	Device is capable of doubled character for scaling.
	TC_SA_INTEGER	Device uses integer multiples only for character scaling.
	TC_SA_CONTIN	Device uses any multiples for exact character scaling.
	TC_EA_DOUBLE	Device can draw double-weight characters.
	TC_IA_ABLE	Device can italicize.
	TC_UA_ABLE	Device can underline.
	TC_SO_ABLE	Device can draw strikeouts.
	TC_RA_ABLE	Device can draw raster fonts.
	TC_VA_ABLE	Device can draw vector fonts.

(continued)

(continued)

Index	Meaning	
	TC_RESERVED	Reserved; must be zero.
	TC_SCROLLBLT	Device cannot scroll using a bit-block transfer. Note that this meaning may be the opposite of what you expect.
COLORMGMTCAPS	**Windows 2000:** Value that indicates the color management capabilities of the device.	
	CM_CMYK_COLOR	Device can accept CMYK color space ICC color profile.
	CM_DEVICE_ICM	Device can perform ICM (Image Color Management) on either the device driver or the device itself.
	CM_GAMMA_RAMP	Device supports **GetDeviceGammaRamp** and **SetDeviceGammaRamp**
	CM_NONE	Device does not support ICM.

Return Values

The return value specifies the value of the desired item.

When *nIndex* is BITSPIXEL and the device has 15bpp or 16bpp, the return value is 16.

Remarks

GetDeviceCaps provides the following six indices in place of printer escapes:

Index	Printer escape replaced
PHYSICALWIDTH	GETPHYSPAGESIZE
PHYSICALHEIGHT	GETPHYSPAGESIZE
PHYSICALOFFSETX	GETPRINTINGOFFSET
PHYSICALOFFSETY	GETPHYSICALOFFSET
SCALINGFACTORX	GETSCALINGFACTOR
SCALINGFACTORY	GETSCALINGFACTOR

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **CreateEnhMetaFile**, **CreateIC**, **DeviceCapabilities**, **GetDIBits**, **GetObjectType**, **SetDIBits**, **SetDIBitsToDevice**, **StretchBlt**, **StretchDIBits**

GetObject

The **GetObject** function retrieves information about a specified graphics object. Depending on the graphics object, the function places a filled-in **BITMAP**, **DIBSECTION**, **EXTLOGPEN**, **LOGBRUSH**, **LOGFONT**, or **LOGPEN** structure, or a count of table entries (for a logical palette), into a specified buffer.

```
int GetObject(
  HGDIOBJ hgdiobj,  // handle to graphics object
  int cbBuffer,     // size of buffer for object information
  LPVOID lpvObject  // buffer for object information
);
```

Parameters

hgdiobj

[in] Handle to the graphics object of interest. This can be a handle to one of the following: a logical bitmap, a brush, a font, a palette, a pen, or a device independent bitmap created by calling the **CreateDIBSection** function.

cbBuffer

[in] Specifies the number of bytes of information to be written to the buffer.

lpvObject

[out] Pointer to a buffer that receives the information about the specified graphics object.

The following table shows the type of information the buffer receives for each type of graphics object you can specify with *hgdiobj*:

Object type	Data written to buffer
HBITMAP	**BITMAP**
HBITMAP returned from a call to **CreateDIBSection**	**DIBSECTION**, if *cbBuffer* is set to **sizeof**(**DIBSECTION**), or **BITMAP**, if *cbBuffer* is set to **sizeof**(**BITMAP**)
HPALETTE	A **WORD** count of the number of entries in the logical palette
HPEN returned from a call to **ExtCreatePen**	**EXTLOGPEN**
HPEN	**LOGPEN**
HBRUSH	**LOGBRUSH**
HFONT	**LOGFONT**

If the *lpvObject* parameter is NULL, the function return value is the number of bytes required to store the information it writes to the buffer for the specified graphics object.

Return Values

If the function succeeds, and *lpvObject* is a valid pointer, the return value is the number of bytes stored into the buffer.

If the function succeeds, and *lpvObject* is NULL, the return value is the number of bytes required to hold the information the function would store into the buffer.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The buffer pointed to by the *lpvObject* parameter must be sufficiently large to receive the information about the graphics object.

If *hgdiobj* is a handle to a bitmap created by calling **CreateDIBSection**, and the specified buffer is large enough, the **GetObject** function returns a **DIBSECTION** structure. In addition, the **bmBits** member of the **BITMAP** structure contained within the **DIBSECTION** will contain a pointer to the bitmap's bit values.

If *hgdiobj* is a handle to a bitmap created by any other means, **GetObject** returns only the width, height, and color format information of the bitmap. You can obtain the bitmap's bit values by calling the **GetDIBits** or **GetBitmapBits** function.

If *hgdiobj* is a handle to a logical palette, **GetObject** retrieves a 2-byte integer that specifies the number of entries in the palette. The function does not retrieve the **LOGPALETTE** structure defining the palette. To retrieve information about palette entries, an application can call the **GetPaletteEntries** function.

If *hgdiobj* is a handle to a font, the **LOGFONT** that is returned is the **LOGFONT** used to create the font. If Windows had to make some interpolation of the font because the precise **LOGFONT** could not be represented, the interpolation will not be reflected in the **LOGFONT**. For example, if you ask for a vertical version of a font that doesn't support vertical painting, the **LOGFONT** indicates the font is vertical, but Windows will paint it horizontally.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **CreateDIBSection**, **GetBitmapBits**, **GetDIBits**, **GetPaletteEntries**, **GetRegionData**, **BITMAP**, **DIBSECTION**, **EXTLOGPEN**, **LOGBRUSH**, **LOGFONT**, **LOGPALETTE**, **LOGPEN**

GetObjectType

The **GetObjectType** retrieves the type of the specified object.

```
DWORD GetObjectType(
  HGDIOBJ h   // handle to graphics object
);
```

Parameters

h

[in] Handle to the graphics object.

Return Values

If the function succeeds, the return value identifies the object. This value can be one of the following:

Value	Meaning
OBJ_BITMAP	Bitmap
OBJ_BRUSH	Brush
OBJ_COLORSPACE	Color space
OBJ_DC	Device context
OBJ_ENHMETADC	Enhanced metafile DC
OBJ_ENHMETAFILE	Enhanced metafile
OBJ_EXTPEN	Extended pen
OBJ_FONT	Font
OBJ_MEMDC	Memory DC
OBJ_METAFILE	Metafile
OBJ_METADC	Metafile DC
OBJ_PAL	Palette
OBJ_PEN	Pen
OBJ_REGION	Region

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **GetObject**, **SelectObject**

GetStockObject

The **GetStockObject** function retrieves a handle to one of the stock pens, brushes, fonts, or palettes.

```
HGDIOBJ GetStockObject(
  int fnObject   // stock object type
);
```

Parameters

fnObject

[in] Specifies the type of stock object. This parameter can be one of the following values:

Value	Meaning
BLACK_BRUSH	Black brush.
DKGRAY_BRUSH	Dark gray brush.
DC_BRUSH	**Windows 98/2000:** Solid color brush. The default color is white. The color can be changed by using the **SetDCBrushColor** function. For more information, see the Remarks section.
GRAY_BRUSH	Gray brush.
HOLLOW_BRUSH	Hollow brush (equivalent to NULL_BRUSH).
LTGRAY_BRUSH	Light gray brush.
NULL_BRUSH	Null brush (equivalent to HOLLOW_BRUSH).
WHITE_BRUSH	White brush.
BLACK_PEN	Black pen.

Value	Meaning
DC_PEN	**Windows 98/2000:** Solid pen color. The default color is white. The color can be changed by using the **SetDCPenColor** function. For more information, see the Remarks section.
WHITE_PEN	White pen.
ANSI_FIXED_FONT	Windows fixed-pitch (monospace) system font.
ANSI_VAR_FONT	Windows variable-pitch (proportional space) system font.
DEVICE_DEFAULT_FONT	**Windows NT/2000:** Device-dependent font.
DEFAULT_GUI_FONT	Default font for user interface objects such as menus and dialog boxes.
OEM_FIXED_FONT	Original equipment manufacturer (OEM) dependent fixed-pitch (monospace) font.
SYSTEM_FONT	System font. By default, the system uses the system font to draw menus, dialog box controls, and text.
SYSTEM_FIXED_FONT	Fixed-pitch (monospace) system font. This stock object is provided only for compatibility with 16-bit Windows versions earlier than 3.0.
DEFAULT_PALETTE	Default palette. This palette consists of the static colors in the system palette.

Return Values

If the function succeeds, the return value is a handle to the requested logical object.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Use the DKGRAY_BRUSH, GRAY_BRUSH, and LTGRAY_BRUSH stock objects only in windows with the CS_HREDRAW and CS_VREDRAW styles. Using a gray stock brush in any other style of window can lead to misalignment of brush patterns after a window is moved or sized. The origins of stock brushes cannot be adjusted.

The HOLLOW_BRUSH and NULL_BRUSH stock objects are equivalent.

The font used by the DEFAULT_GUI_FONT stock object could change. Use this stock object when you want to use the font that menus, dialog boxes, and other user interface objects use.

It is not necessary (but it is not harmful) to delete stock objects by calling **DeleteObject**.

Windows 98, Windows 2000: Both DC_BRUSH and DC_PEN can be used interchangeably with other stock objects like BLACK_BRUSH and BLACK_PEN. For information on retrieving the current pen or brush color, see ***GetDCBrushColor*** and

GetDCPenColor. See *Setting the Pen or Brush Color* for an example of setting colors. The **GetStockObject** function with an argument of DC_BRUSH OR DC_PEN can be used interchangeably with the **SetDCPenColor** and **SetDCBrushColor** functions.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **DeleteObject**, **SelectObject**

ReleaseDC

The **ReleaseDC** function releases a device context (DC), freeing it for use by other applications. The effect of the **ReleaseDC** function depends on the type of DC. It frees only common and window DCs. It has no effect on class or private DCs.

```
int ReleaseDC(
  HWND hWnd,  // handle to window
  HDC hDC     // handle to DC
);
```

Parameters

hWnd

[in] Handle to the window whose DC is to be released.

hDC

[in] Handle to the DC to be released.

Return Values

The return value indicates whether the DC was released. If the DC was released, the return value is 1.

If the DC was not released, the return value is zero.

Remarks

The application must call the **ReleaseDC** function for each call to the **GetWindowDC** function and for each call to the **GetDC** function that retrieves a common DC.

An application cannot use the **ReleaseDC** function to release a DC that was created by calling the **CreateDC** function; instead, it must use the **DeleteDC** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Device Contexts Overview, Device Context Functions, **CreateDC**, **DeleteDC**, **GetDC**, **GetWindowDC**

ResetDC

The **ResetDC** function updates the specified printer or plotter device context (DC), based on the information in the specified structure.

```
HDC ResetDC(
  HDC hdc,                   // handle to DC
  CONST DEVMODE *lpInitData  // DC information
);
```

Parameters

hdc
[in] Handle to the DC to update.

lpInitData
[in] Pointer to a **DEVMODE** structure containing information about the new DC.

Return Values

If the function succeeds, the return value is a handle to the original DC.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application will typically use the **ResetDC** function when a window receives a **WM_DEVMODECHANGE** message. **ResetDC** can also be used to change the paper orientation or paper bins while printing a document.

The **ResetDC** function cannot be used to change the driver name, device name, or the output port. When the user changes the port connection or device name, the application must delete the original DC and create a new DC with the new information.

An application can pass an information DC to the **ResetDC** function. In that situation, **ResetDC** will always return a printer DC.

ICM: The color profile of the DC specified by the *hdc* parameter will be reset based on the information contained in the **lpInitData** member of the **DEVMODE** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Device Contexts Overview, Device Context Functions, **DeviceCapabilities**, **DEVMODE**, **Escape**

RestoreDC

The **RestoreDC** function restores a device context (DC) to the specified state. The DC is restored by popping state information off a stack created by earlier calls to the **SaveDC** function.

```
BOOL RestoreDC(
  HDC hdc,         // handle to DC
  int nSavedDC     // restore state
);
```

Parameters

hdc

[in] Handle to the DC.

nSavedDC

[in] Specifies the saved state to be restored. If this parameter is positive, *nSavedDC* represents a specific instance of the state to be restored. If this parameter is negative, *nSavedDC* represents an instance relative to the current state. For example, –1 restores the most recently saved state.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The stack can contain the state information for several instances of the DC. If the state specified by the specified parameter is not at the top of the stack, **RestoreDC** deletes all state information between the top of the stack and the specified instance.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **SaveDC**

SaveDC

The **SaveDC** function saves the current state of the specified device context (DC) by copying data describing selected objects and graphic modes (such as the bitmap, brush, palette, font, pen, region, drawing mode, and mapping mode) to a context stack.

```
int SaveDC(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
[in] Handle to the DC whose state is to be saved.

Return Values

If the function succeeds, the return value identifies the saved state.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **SaveDC** function can be used any number of times to save any number of instances of the DC state.

A saved state can be restored by using the **RestoreDC** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **RestoreDC**

SelectObject

The **SelectObject** function selects an object into the specified device context (DC). The new object replaces the previous object of the same type.

```
HGDIOBJ SelectObject(
  HDC hdc,          // handle to DC
  HGDIOBJ hgdiobj   // handle to object
);
```

Parameters

hdc

[in] Handle to the DC.

hgdiobj

[in] Handle to the object to be selected. The specified object must have been created by using one of the following functions:

Object	Functions
Bitmap	**CreateBitmap**, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, **CreateDIBitmap**, **CreateDIBSection** (Bitmaps can be selected for memory DCs only, and for only one DC at a time.)
Brush	**CreateBrushIndirect**, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**
Font	**CreateFont**, **CreateFontIndirect**
Pen	**CreatePen**, **CreatePenIndirect**
Region	**CombineRgn**, **CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreatePolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**

Return Values

If the selected object is not a region and the function succeeds, the return value is a handle to the object being replaced. If the selected object is a region and the function succeeds, the return value is one of the following values:

Value	Meaning
SIMPLEREGION	Region consists of a single rectangle.
COMPLEXREGION	Region consists of more than one rectangle.
NULLREGION	Region is empty.

If an error occurs and the selected object is not a region, the return value is NULL. Otherwise, it is GDI_ERROR.

Remarks

This function returns the previously selected object of the specified type. An application should always replace a new object with the original, default object after it has finished drawing with the new object.

An application cannot select a bitmap into more than one DC at a time.

ICM: If the object being selected is a brush or a pen color management is performed.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **CombineRgn**, **CreateBitmap**, **CreateBitmapIndirect**, **CreateBrushIndirect**, **CreateCompatibleBitmap**, **CreateDIBitmap**, **CreateDIBPatternBrush**, **CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreateFont**, **CreateFontIndirect**, **CreateHatchBrush**, **CreatePatternBrush**, **CreatePen**, **CreatePenIndirect**, **CreatePolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**, **CreateSolidBrush**, **SelectClipRgn**, **SelectPalette**

SetDCBrushColor

SetDCBrushColor function sets the current device context (DC) brush color to the specified color value. If the device cannot represent the specified color value, the color is set to the nearest physical color.

```
COLORREF SetDCBrushColor(
  HDC hdc,           // handle to DC
  COLORREF crColor   // new brush color
);
```

Parameters

hdc
[in] Handle to the DC.

crColor
[in] Specifies the new brush color.

Return Values

If the function succeeds, the return value specifies the previous DC brush color as a **COLORREF** value.

If the function fails, the return value is CLR_INVALID.

Remarks

When the stock DC_BRUSH is selected in a DC, all the subsequent drawings will be done using the DC brush color until the stock brush is deselected. The default DC_BRUSH color is WHITE.

The function will return the previous DC_BRUSH color, even if the stock brush DC_BRUSH is not selected in the DC: however, this will not be used in drawing operations until the stock DC_BRUSH is selected in the DC.

See *Setting the Pen or Brush Color* for an example of setting colors. The **GetStockObject** function with an argument of DC_BRUSH OR DC_PEN can be used interchangeably with the **SetDCPenColor** and **SetDCBrushColor** functions.

ICM: Color management is performed if ICM is enabled.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Device Contexts Overview, Device Context Functions, **GetDCBrushColor**, **COLORREF**

SetDCPenColor

SetDCPenColor function sets the current device context (DC) pen color to the specified color value. If the device cannot represent the specified color value, the color is set to the nearest physical color.

```
COLORREF SetDCPenColor(
  HDC hdc,          // handle to DC
  COLORREF crColor  // new pen color
);
```

Parameters

hdc
  [in] Handle to the DC.

crColor
  [in] Specifies the new pen color.

Return Values

If the function succeeds, the return value specifies the previous DC pen color as a **COLORREF** value. If the function fails, the return value is CLR_INVALID.

Remarks

The function will return the previous DC_PEN color, even if the stock pen DC_PEN is not selected in the DC: however, this will not be used in drawing operations until the stock DC_PEN is selected in the DC.

See *Setting the Pen or Brush Color* for an example of setting colors. The **GetStockObject** function with an argument of DC_BRUSH OR DC_PEN can be used interchangeably with the **SetDCPenColor** and **SetDCBrushColor** functions.

ICM: Color management is performed if ICM is enabled.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Device Contexts Overview, Device Context Functions, **GetDCPenColor**, **COLORREF**

Device Context Structures

DISPLAY_DEVICE

The **DISPLAY_DEVICE** structure receives information about the display device specified by the *iDevNum* parameter of the **EnumDisplayDevices** function.

```
typedef struct _DISPLAY_DEVICE {
  DWORD cb;
  WCHAR DeviceName[32];
  WCHAR DeviceString[128];
  DWORD StateFlags;
  WCHAR DeviceID[128];
  WCHAR DeviceKey[128];
} DISPLAY_DEVICE, *PDISPLAY_DEVICE;
```

Members

cb

Size, in bytes, of the **DISPLAY_DEVICE** structure. This must be initialized prior to calling **EnumDisplayDevices**.

DeviceName

An array of characters identifying the device name.

DeviceString

An array of characters containing the device context string.

StateFlags

Device state flags. It can be any reasonable combination of the following:

Value	Meaning
DISPLAY_DEVICE_ATTACHED_TO_DESKTOP	The device is part of the desktop.
DISPLAY_DEVICE_MIRRORING_DRIVER	The device is a pseudo-device for NetMeeting.
DISPLAY_DEVICE_MODESPRUNED	The device has more display modes than its output devices support.
DISPLAY_DEVICE_PRIMARY_DEVICE	The primary desktop is on the device. For a system with a single display card, this is always set. For a system with multiple display cards, only one device can have this set.
DISPLAY_DEVICE_VGA_COMPATIBLE	The device is VGA compatible.

DeviceID

Windows 98: A string that uniquely identifies the hardware adapter or the monitor. This is the Plug and Play identifier.

DeviceKey

Reserved.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Unicode: Declared as Unicode and ANSI structures.

See Also

Device Contexts Overview, Device Context Structures, **EnumDisplayDevices**

VIDEOPARAMETERS

The **VIDEOPARAMETERS** structure contains information for a video connection.

```
typedef struct _VIDEOPARAMETERS {
  GUID  guid;
  ULONG  dwOffset;
  ULONG  dwCommand;
  ULONG  dwFlags;
  ULONG  dwMode;
  ULONG  dwTVStandard;
  ULONG  dwAvailableModes;
  ULONG  dwAvailableTVStandard;
  ULONG  dwFlickerFilter;
  ULONG  dwOverScanX;
  ULONG  dwOverScanY;
  ULONG  dwMaxUnscaledX;
  ULONG  dwMaxUnscaledY;
  ULONG  dwPositionX;
  ULONG  dwPositionY;
  ULONG  dwBrightness;
  ULONG  dwContrast;
  ULONG  dwCPType;
  ULONG  dwCPCommand;
  ULONG  dwCPStandard;
```

(continued)

(continued)

```
  ULONG  dwCPKey;
  ULONG  bCP_APSTriggerBits;
  UCHAR  bOEMCopyProtection[256];
} VIDOEPARAMETERS, *PVIDEOPARAMETERS;
```

Members

guid

Specifies the GUID for this structure. Display drivers should verify the GUID at the start of the structure before processing the structure.

dwOffset

Reserved; must be zero.

dwCommand

Specifies whether to retrieve or set the values that are indicated by the other members of this structure. This member can be one of the following values:

Value	Meaning
VP_COMMAND_GET	Gets current video capabilities. **DwFlags** is 0 if capability is not supported.
VP_COMMAND_SET	Sets video parameters.

dwFlags

Indicates which fields contain valid data. For VP_COMMAND_GET, these are the fields to retrieve, for VP_COMMAND_SET, these are the fields to set. **dwFlags** is 0 if capability is not supported. It can be any combination of the following:

Value	Fields containing data
VP_FLAGS_TV_MODE	**dwMode**
VP_FLAGS_TV_STANDARD	**dwTVStandard**
VP_FLAGS_FLICKER	**dwFlickerFilter**
VP_FLAGS_OVERSCAN	**dwOverScanX**, **dwOverScanY**
VP_FLAGS_MAX_UNSCALED	**dwMaxUnscaledX**, **dwMaxUnscaledY**. Do not use if VP_COMMAND_SET is specified.
VP_FLAGS_POSITION	**dwPositionX**, **dwPositionY**
VP_FLAGS_BRIGHTNESS	**dwBrightness**
VP_FLAGS_CONTRAST	**dwContrast**
VP_FLAGS_COPYPROTECT	**dwCPType**, **dwCPCommand**, **dwCPStandard**

dwMode

Specifies the current playback mode. This member is valid for both VP_COMMAND_GET and VP_COMMAND_SET. It can be one of the following:

Value	Meaning
VP_MODE_WIN_GRAPHICS	Describes a set of display settings that are optimal for Windows display, with the flicker filter on and any overscan display off.
VP_MODE_TV_PLAYBACK	Describes a set of display settings for video playback, with the flicker filter off and the overscan display on.

dwTVStandard

Specifies the TV standard. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET. It can be any one of the following:

VP_TV_STANDARD_NTSC_433

VP_TV_STANDARD_NTSC_M

VP_TV_STANDARD_NTSC_M_J

VP_TV_STANDARD_PAL_60

VP_TV_STANDARD_PAL_B

VP_TV_STANDARD_PAL_D

VP_TV_STANDARD_PAL_G

VP_TV_STANDARD_PAL_H

VP_TV_STANDARD_PAL_I

VP_TV_STANDARD_PAL_M

VP_TV_STANDARD_PAL_N

VP_TV_STANDARD_SECAM_B

VP_TV_STANDARD_SECAM_D

VP_TV_STANDARD_SECAM_G

VP_TV_STANDARD_SECAM_H

VP_TV_STANDARD_SECAM_K

VP_TV_STANDARD_SECAM_K1

VP_TV_STANDARD_SECAM_L

VP_TV_STANDARD_SECAM_L1

VP_TV_STANDARD_WIN_VGA

dwAvailableModes

Specifies which modes are available. This is valid only for VP_COMMAND_GET. It can be any combination of the values specified in **dwMode**.

dwAvailableTVStandard

Specifies the TV standards that are available. This is valid only for VP_COMMAND_GET. It can be any combination of the values specified in **dwTVStandard**.

dwFlickerFilter

Specifies the flicker reduction provided by the hardware. This is a percentage value in tenths of a percent, from 0 to 1000, where 0 is no flicker reduction and 1000 is maximum flicker reduction. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwOverScanX

Specifies the amount of overscan in the horizontal direction. This is a percentage value in tenths of a percent, from 0 to 1000. A value of 0 indicates no overscan, ensuring that the entire display is visible. A value of 1000 is maximum overscan and typically causes some of the image to be off the edge of the screen. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwOverScanY

Specifies the amount of overscan in the vertical direction. This is a percentage value in tenths of a percent, from 0 to 1000. A value of 0 indicates no overscan, ensuring that the entire display is visible. A value of 1000 is maximum overscan and typically causes some of the image to be off the edge of the screen. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwMaxUnscaledX

Specifies the maximum horizontal resolution, in pixels, that is supported when the video is not scaled. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwMaxUnscaledY

Specifies the maximum vertical resolution, in pixels, that is supported when the video is not scaled. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwPositionX

Specifies the horizontal adjustment to the center of the image. Units are in pixels. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwPositionY

Specifies the vertical adjustment to the center of the image. Units are in scan lines. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwBrightness

Adjustment to the DC offset of the video signal to increase brightness on the television. It is a percentage value, 0 to 100, where 0 means no adjustment and 100 means maximum adjustment. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwContrast

Adjustment to the gain of the video signal to increase the intensity of whiteness on the television. It is a percentage value, 0 to 100, where 0 means no adjustment and 100 means maximum adjustment. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

dwCPType

Specifies the copy protection type. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET. It can be one of the following:

Value	Meaning
VP_CP_TYPE_APS_TRIGGER	only DVD trigger bits available.
VP_CP_TYPE_MACROVISION	Full macrovision data is available.

dwCPCommand

Specifies the copy protection command. This field is only valid for VP_COMMAND_SET. It can be one of the following:

Value	Meaning
VP_CP_CMD_ACTIVATE	Activate copy protection.
VP_CP_CMD_CHANGE	Change copy protection.
VP_CP_CMD_DEACTIVATE	Deactivate copy protection.

dwCPStandard

Specifies TV standards for which copy protection types are available. This field is valid only for VP_COMMAND_GET.

dwCPKey

Specifies the copy protection key returned if **dwCPCommand** is set to VP_CP_CMD_ACTIVATE. The caller must set this key when the **dwCPCommand** field is either VP_CP_CMD_DEACTIVATE or VP_CP_CMD_CHANGE. If the caller sets an incorrect key, the driver must not change the current copy protection settings. This field is valid only for VP_COMMAND_SET.

bCP_APSTriggerBits

Specifies the DVD APS trigger bit flag. This is valid only for VP_COMMAND_SET. Currently, only bits 0 and 1 are valid, the rest of the size is for DWORD alignment purposes. It can be one of the following:

Value	Meaning
VP_CP_TYPE_APS_TRIGGER	Only DVD trigger bits available.
VP_CP_TYPE_MACROVISION	Full macrovision data is available.

bOEMCopyProtection

Specifies the OEM specific copy protection data. Maximum of 256 characters. This field is valid for both VP_COMMAND_GET and VP_COMMAND_SET.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in tvout.h.

See Also

Device Contexts Overview, Device Context Structures, **ChangeDisplaySettingsEx**

Device Context Messages

WM_DEVMODECHANGE

The **WM_DEVMODECHANGE** message is sent to all top-level windows whenever the user changes device-mode settings.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_DEVMODECHANGE
  WPARAM wParam,   // not used
  LPARAM lParam    // device name (LPCTSTR)
);
```

Parameters

wParam

This parameter is not used.

lParam

Pointer to a string that specifies the device name.

Return Values

An application should return zero if it processes this message.

Remarks

This message cannot be sent directly to a window. To send the **WM_DEVMODECHANGE** message to all top-level windows, use the **SendMessageTimeout** function with the *hWnd* parameter set to HWND_BROADCAST.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Device Contexts Overview, Device Context Messages

CHAPTER 12

Filled Shapes

Filled shapes are geometric forms that are outlined by using the current pen and filled by using the current brush. There are five filled shapes:

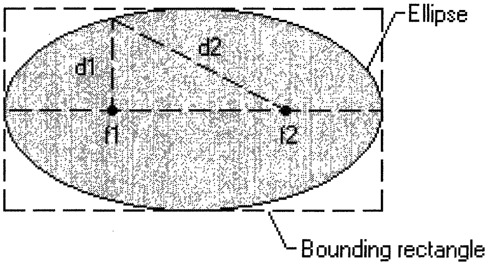
- Ellipse
- Chord
- Pie
- Polygon
- Rectangle

About Filled Shapes

A Win32-based application uses filled shapes in a variety of ways. Spreadsheet applications, for example, use filled shapes to construct charts and graphs, and drawing and painting applications use filled shapes to allow the user to draw figures and illustrations.

About Ellipses

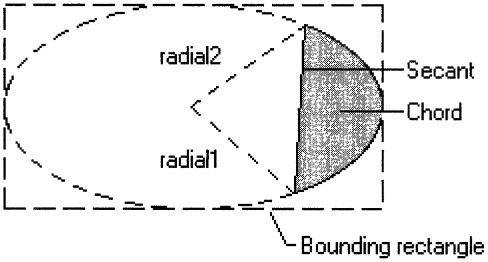
An ellipse is a closed curve defined by two fixed points (*f1* and *f2*) such that the sum of the distances (*d1* + *d2*) from any point on the curve to the two fixed points is constant. The following illustration shows an ellipse drawn by using the **Ellipse** function.



When calling **Ellipse**, an application supplies the coordinates of the upper-left and lower-right corners of the ellipse's bounding rectangle. A *bounding rectangle* is the smallest rectangle completely surrounding the ellipse. When the system draws the ellipse, it excludes the right and lower sides if no world transformations are set. Therefore, for any rectangle measuring *x* units wide by *y* units high, the associated ellipse measures *x*–1 units wide by *y*–1 units high. If the application sets a world transformation by calling the **SetWorldTransform** or **ModifyWorldTransform** function, the system includes the right and lower sides.

About Chords

A *chord* is a region bounded by the intersection of an ellipse and a line segment called a *secant*. The following illustration shows a chord drawn by using the **Chord** function.

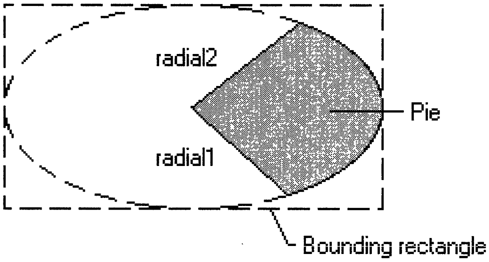


When calling **Chord**, an application supplies the coordinates of the upper-left and lower-right corners of the ellipse's bounding rectangle, as well as the coordinates of two points defining two radials. A radial is a line drawn from the center of an ellipse's bounding rectangle to a point on the ellipse.

When the system draws the curved part of the chord, it does so by using the current arc direction for the specified device context. The default arc direction is counterclockwise. You can have your application reset the arc direction by calling the **SetArcDirection** function.

About Pies

A *pie* is a region bounded by the intersection of an ellipse curve and two radials. The following illustration shows a pie drawn by using the **Pie** function.



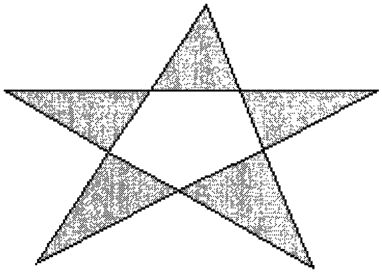
When calling **Pie**, an application supplies the coordinates of the upper-left and lower-right corners of the ellipse's bounding rectangle, as well as the coordinates of two points defining two radials.

When the system draws the curved part of the pie, it uses the current arc direction for the given device context. The default arc direction is counterclockwise. An application can reset the arc direction by calling the **SetArcDirection** function.

About Polygons

A *polygon* is a filled shape with straight sides. The sides of a polygon are drawn by using the current pen. When the system fills a polygon, it uses the current brush and the current polygon fill mode. The two fill modes—alternate (the default) and winding—determine whether regions within a complex polygon are filled or left unpainted. An application can select either mode by calling the **SetPolyFillMode** function. For more information about polygon fill modes, see Regions.

The following illustration shows a polygon drawn by using **Polygon**.



In addition to drawing a single polygon by using **Polygon**, an application can draw multiple polygons by using the **PolyPolygon** function.

Drawing Rectangles

A rectangle is a four-sided polygon whose opposing sides are parallel and equal in length. Although an application can draw a rectangle by calling the **Polygon** function, supplying the coordinates of each corner, the **Rectangle** function provides a simpler method. This function requires only the coordinates for the upper-left and the lower-right corners. When an application calls the **Rectangle** function, the system draws the rectangle, excluding the right and lower sides if no world transformation is set for the given device context.

If a world transformation has been set by using the **SetWorldTransform** or **ModifyWorldTransform** function, the system includes the right and lower edges.

In addition to drawing a traditional rectangle, you can draw rectangles with rounded corners. The **RoundRect** function requires that the application supply the coordinates of the lower-left and upper-right corners, as well as the width and height of the ellipse used to round each corner.

The Win32 API also provides three functions that applications can use to manipulate rectangles, described as follows.

Function	Description
FillRect	Repaints the interior of a rectangle.
FrameRect	Redraws the sides of a rectangle.
InvertRect	Inverts the colors that appear within the interior of a rectangle.

Filled Shape Reference

Filled Shape Functions

Chord

The **Chord** function draws a chord (a region bounded by the intersection of an ellipse and a line segment, called a secant). The chord is outlined by using the current pen and filled by using the current brush.

```
BOOL Chord(
  HDC hdc,           // handle to DC
  int nLeftRect,     // x-coord of upper-left corner of rectangle
  int nTopRect,      // y-coord of upper-left corner of rectangle
  int nRightRect,    // x-coord of lower-right corner of rectangle
  int nBottomRect,   // y-coord of lower-right corner of rectangle
  int nXRadial1,     // x-coord of first radial's endpoint
  int nYRadial1,     // y-coord of first radial's endpoint
  int nXRadial2,     // x-coord of second radial's endpoint
  int nYRadial2      // y-coord of second radial's endpoint
);
```

Parameters

hdc
[in] Handle to the device context in which the chord appears.

nLeftRect
[in] Specifies the x-coordinate of the upper-left corner of the bounding rectangle.

nTopRect
[in] Specifies the y-coordinate of the upper-left corner of the bounding rectangle.

nRightRect
[in] Specifies the x-coordinate of the lower-right corner of the bounding rectangle.

nBottomRect
[in] Specifies the y-coordinate of the lower-right corner of the bounding rectangle.

nXRadial1

[in] Specifies the x-coordinate of the endpoint of the radial defining the beginning of the chord.

nYRadial1

[in] Specifies the y-coordinate of the endpoint of the radial defining the beginning of the chord.

nXRadial2

[in] Specifies the x-coordinate of the endpoint of the radial defining the end of the chord.

nYRadial2

[in] Specifies the y-coordinate of the endpoint of the radial defining the end of the chord.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The curve of the chord is defined by an ellipse that fits the specified bounding rectangle. The curve begins at the point where the ellipse intersects the first radial and extends counterclockwise to the point where the ellipse intersects the second radial. The chord is closed by drawing a line from the intersection of the first radial and the curve to the intersection of the second radial and the curve.

If the starting point and ending point of the curve are the same, a complete ellipse is drawn.

The current position is neither used nor updated by **Chord**.

Windows 95/98: The sum of the coordinates of the bounding rectangle cannot exceed 32,767. The sum of *nLeftRect* and *nRightRect* or *nTopRect* and *nBottomRect* parameters cannot exceed 32,767.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Filled Shapes Overview, Filled Shape Functions, **AngleArc**, **Arc**, **ArcTo**, **Pie**

Ellipse

The **Ellipse** function draws an ellipse. The center of the ellipse is the center of the specified bounding rectangle. The ellipse is outlined by using the current pen and is filled by using the current brush.

```
BOOL Ellipse(
  HDC hdc,          // handle to DC
  int nLeftRect,    // x-coord of upper-left corner of rectangle
  int nTopRect,     // y-coord of upper-left corner of rectangle
  int nRightRect,   // x-coord of lower-right corner of rectangle
  int nBottomRect   // y-coord of lower-right corner of rectangle
);
```

Parameters

hdc

[in] Handle to the device context.

nLeftRect

[in] Specifies the x-coordinate of the upper-left corner of the bounding rectangle.

nTopRect

[in] Specifies the y-coordinate of the upper-left corner of the bounding rectangle.

nRightRect

[in] Specifies the x-coordinate of the lower-right corner of the bounding rectangle.

nBottomRect

[in] Specifies the y-coordinate of the lower-right corner of the bounding rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The current position is neither used nor updated by **Ellipse**.

Windows 95/98: The sum of the coordinates of the bounding rectangle cannot exceed 32,767. The sum of *nLeftRect* and *nRightRect* or *nTopRect* and *nBottomRect* parameters cannot exceed 32,767.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.

Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Filled Shapes Overview, Filled Shape Functions, **Arc**, **ArcTo**

FillRect

The **FillRect** function fills a rectangle by using the specified brush. This function includes the left and top borders, but excludes the right and bottom borders of the rectangle.

```
int FillRect(
  HDC hDC,           // handle to DC
  CONST RECT *lprc,  // rectangle
  HBRUSH hbr         // handle to brush
);
```

Parameters

hDC
[in] Handle to the device context.

lprc
[in] Pointer to a **RECT** structure that contains the logical coordinates of the rectangle to be filled.

hbr
[in] Handle to the brush used to fill the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The brush identified by the *hbr* parameter may be either a handle to a logical brush or a color value. If specifying a handle to a logical brush, call one of the following functions to obtain the handle: **CreateHatchBrush**, **CreatePatternBrush**, or **CreateSolidBrush**. Additionally, you may retrieve a handle to one of the stock brushes by using the **GetStockObject** function. If specifying a color value for the *hbr* parameter, it must be one of the standard system colors (the value 1 must be added to the chosen color). For example:

```
FillRect(hdc, &rect, (HBRUSH) (COLOR_WINDOW+1));
```

For a list of all the standard system colors, see **GetSysColor**.

When filling the specified rectangle, **FillRect** does not include the rectangle's right and bottom sides. GDI fills a rectangle up to, but not including, the right column and bottom row, regardless of the current mapping mode.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Filled Shapes Overview, Filled Shape Functions, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**, **GetStockObject**, **RECT**

FrameRect

The **FrameRect** function draws a border around the specified rectangle by using the specified brush. The width and height of the border are always one logical unit.

```
int FrameRect(
  HDC hDC,          // handle to DC
  CONST RECT *lprc, // rectangle
  HBRUSH hbr        // handle to brush
);
```

Parameters

hDC
[in] Handle to the device context in which the border is drawn.

lprc
[in] Pointer to a **RECT** structure that contains the logical coordinates of the upper-left and lower-right corners of the rectangle.

hbr
[in] Handle to the brush used to draw the border.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The brush identified by the *hbr* parameter must have been created by using the **CreateHatchBrush**, **CreatePatternBrush**, or **CreateSolidBrush** function, or retrieved by using the **GetStockObject** function.

If the **bottom** member of the **RECT** structure is less than or equal to the **top** member, or if the **right** member is less than or equal to the **left** member, the function does not draw the rectangle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Filled Shapes Overview, Filled Shape Functions, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**, **GetStockObject**, **RECT**

InvertRect

The **InvertRect** function inverts a rectangle in a window by performing a logical NOT operation on the color values for each pixel in the rectangle's interior.

```
BOOL InvertRect(
  HDC hDC,            // handle to DC
  CONST RECT *lprc    // rectangle
);
```

Parameters

hDC

[in] Handle to the device context.

lprc

[in] Pointer to a **RECT** structure that contains the logical coordinates of the rectangle to be inverted.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

On monochrome screens, **InvertRect** makes white pixels black and black pixels white. On color screens, the inversion depends on how colors are generated for the screen. Calling **InvertRect** twice for the same rectangle restores the display to its previous colors.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Filled Shapes Overview, Filled Shape Functions, **FillRect**, **RECT**

Pie

The **Pie** function draws a pie-shaped wedge bounded by the intersection of an ellipse and two radials. The pie is outlined by using the current pen and filled by using the current brush.

```
BOOL Pie(
  HDC hdc,          // handle to DC
  int nLeftRect,    // x-coord of upper-left corner of rectangle
  int nTopRect,     // y-coord of upper-left corner of rectangle
  int nRightRect,   // x-coord of lower-right corner of rectangle
  int nBottomRect,  // y-coord of lower-right corner of rectangle
  int nXRadial1,    // x-coord of first radial's endpoint
  int nYRadial1,    // y-coord of first radial's endpoint
  int nXRadial2,    // x-coord of second radial's endpoint
  int nYRadial2     // y-coord of second radial's endpoint
);
```

Parameters

hdc

[in] Handle to the device context.

nLeftRect

[in] Specifies the x-coordinate of the upper-left corner of the bounding rectangle.

nTopRect

[in] Specifies the y-coordinate of the upper-left corner of the bounding rectangle.

nRightRect

[in] Specifies the x-coordinate of the lower-right corner of the bounding rectangle.

nBottomRect

[in] Specifies the y-coordinate of the lower-right corner of the bounding rectangle.

nXRadial1

[in] Specifies the x-coordinate of the endpoint of the first radial.

nYRadial1

[in] Specifies the y-coordinate of the endpoint of the first radial.

nXRadial2

[in] Specifies the x-coordinate of the endpoint of the second radial.

nYRadial2

[in] Specifies the y-coordinate of the endpoint of the second radial.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The curve of the pie is defined by an ellipse that fits the specified bounding rectangle. The curve begins at the point where the ellipse intersects the first radial and extends counterclockwise to the point where the ellipse intersects the second radial.

The current position is neither used nor updated by the **Pie** function.

Windows 95/98: The sum of the coordinates of the bounding rectangle cannot exceed 32,767. The sum of *nLeftRect* and *nRightRect* or *nTopRect* and *nBottomRect* parameters cannot exceed 32,767.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Filled Shapes Overview, Filled Shape Functions, **AngleArc**, **Arc**, **ArcTo**, **Chord**

Polygon

The **Polygon** function draws a polygon consisting of two or more vertices connected by straight lines. The polygon is outlined by using the current pen and filled by using the current brush and polygon fill mode.

```
BOOL Polygon(
  HDC hdc,                // handle to DC
  CONST POINT *lpPoints,  // polygon vertices
  int nCount              // count of polygon vertices
);
```

Parameters

hdc

[in] Handle to the device context.

lpPoints

[in] Pointer to an array of **POINT** structures that specify the vertices of the polygon.

nCount

[in] Specifies the number of vertices in the array. This value must be greater than or equal to 2.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The polygon is closed automatically by drawing a line from the last vertex to the first.

The current position is neither used nor updated by the **Polygon** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Filled Shapes Overview, Filled Shape Functions, **GetPolyFillMode**, **POINT**, **Polyline**, **PolylineTo**, **PolyPolygon**, **SetPolyFillMode**

PolyPolygon

The **PolyPolygon** function draws a series of closed polygons. Each polygon is outlined by using the current pen and filled by using the current brush and polygon fill mode. The polygons drawn by this function can overlap.

```
BOOL PolyPolygon(
  HDC hdc,                   // handle to DC
  CONST POINT *lpPoints,     // array of vertices
  CONST INT *lpPolyCounts,   // array of count of vertices
  int nCount                 // count of polygons
);
```

Parameters

hdc

[in] Handle to the device context.

lpPoints

[in] Pointer to an array of **POINT** structures that define the vertices of the polygons. The polygons are specified consecutively. Each polygon is closed automatically by drawing a line from the last vertex to the first. Each vertex should be specified once.

lpPolyCounts

[in] Pointer to an array of integers, each of which specifies the number of points in the corresponding polygon. Each integer must be greater than or equal to 2.

nCount

[in] Specifies the total number of polygons.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The current position is neither used nor updated by this function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Filled Shapes Overview, Filled Shape Functions, **GetPolyFillMode**, **POINT**, **Polygon**, **Polyline**, **PolylineTo**, **SetPolyFillMode**

Rectangle

The **Rectangle** function draws a rectangle. The rectangle is outlined by using the current pen and filled by using the current brush.

```
BOOL Rectangle(
  HDC hdc,          // handle to DC
  int nLeftRect,    // x-coord of upper-left corner of rectangle
  int nTopRect,     // y-coord of upper-left corner of rectangle
  int nRightRect,   // x-coord of lower-right corner of rectangle
  int nBottomRect   // y-coord of lower-right corner of rectangle
);
```

Parameters

hdc

[in] Handle to the device context.

nLeftRect

[in] Specifies the logical x-coordinate of the upper-left corner of the rectangle.

nTopRect

[in] Specifies the logical y-coordinate of the upper-left corner of the rectangle.

nRightRect

[in] Specifies the logical x-coordinate of the lower-right corner of the rectangle.

nBottomRect

[in] Specifies the logical y-coordinate of the lower-right corner of the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The current position is neither used nor updated by **Rectangle**.

The rectangle that is drawn excludes the bottom and right edges.

If a PS_NULL pen is used, the dimensions of the rectangle are 1 pixel less in height and 1 pixel less in width.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Filled Shapes Overview, Filled Shape Functions, **RoundRect**

RoundRect

The **RoundRect** function draws a rectangle with rounded corners. The rectangle is outlined by using the current pen and filled by using the current brush.

```
BOOL RoundRect(
  HDC hdc,           // handle to DC
  int nLeftRect,     // x-coord of upper-left corner of rectangle
  int nTopRect,      // y-coord of upper-left corner of rectangle
  int nRightRect,    // x-coord of lower-right corner of rectangle
  int nBottomRect,   // y-coord of lower-right corner of rectangle
  int nWidth,        // width of ellipse
  int nHeight        // height of ellipse
);
```

Parameters

hdc
[in] Handle to the device context.

nLeftRect
[in] Specifies the x-coordinate of the upper-left corner of the rectangle.

nTopRect
[in] Specifies the y-coordinate of the upper-left corner of the rectangle.

nRightRect
[in] Specifies the x-coordinate of the lower-right corner of the rectangle.

nBottomRect
[in] Specifies the y-coordinate of the lower-right corner of the rectangle.

nWidth
[in] Specifies the width of the ellipse used to draw the rounded corners.

nHeight
[in] Specifies the height of the ellipse used to draw the rounded corners.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The current position is neither used nor updated by this function.

Windows 95/98: The sum of the coordinates of the bounding rectangle cannot exceed 32,767. The sum of *nLeftRect* and *nRightRect* or *nTopRect* and *nBottomRect* parameters cannot exceed 32,767.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Filled Shapes Overview, Filled Shape Functions, **Rectangle**

CHAPTER 13

Lines and Curves

Lines and curves are used to draw graphics output on raster devices. As discussed in this overview, a *line* is a set of highlighted pixels on a raster display (or a set of dots on a printed page) identified by two points: a starting point and an ending point. A *regular curve* is a set of highlighted pixels on a raster display (or dots on a printed page) that defines the perimeter (or part of the perimeter) of a conic section. An *irregular curve* is a set of pixels that defines a curve that does not fit the perimeter of a conic section.

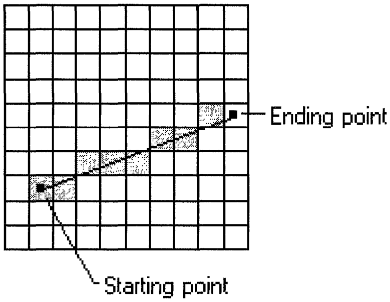
About Lines and Curves

All types of Win32-based applications use lines and curves to draw graphics output on raster devices. Computer-aided design (CAD) and drawing applications use lines and curves to outline objects, specify the centers of objects, the dimensions of objects, and so on. Spreadsheet applications use lines and curves to draw grids, charts, and graphs. Word processing applications use lines to create rules and borders on a page of text.

Lines

A line is a set of highlighted pixels on a raster display (or a set of dots on a printed page) identified by two points: a starting point and an ending point. The pixel located at the starting point is always included in the line, and the pixel located at the ending point is always excluded. (This kind of line is sometimes called inclusive-exclusive.)

When an application calls one of the Win32 line-drawing functions, graphical device interface (GDI), or in some cases a device driver, determines which pixels should be highlighted. GDI is a dynamic link library (DLL) that processes graphics function calls from a Win32-based application and passes those calls to a device driver. A device driver is a DLL that receives input from GDI, converts the input to device commands, and passes those commands to the appropriate device. GDI uses a digital differential analyzer (DDA) to determine the set of pixels that define a line. A DDA determines the set of pixels by examining each point on the line and identifying those pixels on the display surface (or dots on a printed page) that correspond to the points. The following illustration shows a line, its starting point, its ending point, and the pixels highlighted by using a simple DDA.

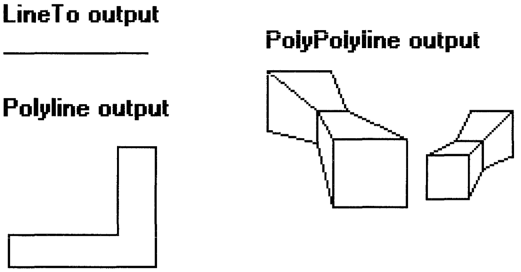


The simplest and most common DDA is the Bresenham, or incremental, DDA. A modified version of this algorithm draws lines in Windows. The incremental DDA is noted for its simplicity, but it is also noted for its inaccuracy. Because it rounds off to the nearest integer value, it sometimes fails to represent the original line requested by the application. The DDA used by GDI does not round off to the nearest integer. As a result, this new DDA produces output that is sometimes much closer in appearance to the original line requested by the application.

Note If an application requires line output that cannot be achieved with the new DDA, it can draw its own lines by calling the **LineDDA** function and supplying a private DDA (**LineDDAProc**). However, the **LineDDA** function draws lines much slower than the Win32 line-drawing functions. Do *not* use this function within an application if speed is a primary concern.

An application can use the new DDA to draw single lines and multiple, connected line segments. An application can draw a single line by calling the **LineTo** function. This function draws a line from the current position up to, but not including, a specified ending point. An application can draw a series of connected line segments by calling the **Polyline** function, supplying an array of points that specify the ending point of each line segment. An application can draw multiple, disjointed series of connected line segments by calling the **PolyPolyline** function, supplying the required ending points.

The following illustration shows line output created by calling the **LineTo**, **Polyline**, and **PolyPolyline** functions.

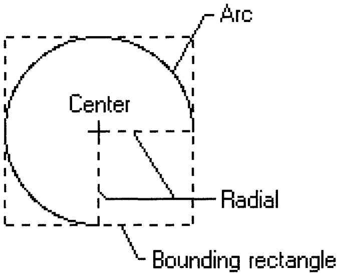


Curves

A regular curve is a set of highlighted pixels on a raster display (or dots on a printed page) that define the perimeter (or part of the perimeter) of a conic section. An irregular curve is a set of pixels that define a curve that does not fit the perimeter of a conic section. In Win32-based applications, the ending point is excluded from a curve just as it is excluded from a line.

When an application calls one of the Win32 curve-drawing functions, GDI breaks the curve into a number of extremely small, discrete line segments. After determining the endpoints (starting point and ending point) for each of these line segments, GDI determines which pixels (or dots) define each line by applying its DDA.

An application can draw an ellipse or part of an ellipse by calling the **Arc** function. This function draws the curve within the perimeter of an invisible rectangle called a bounding rectangle. The size of the ellipse is specified by two invisible radials extending from the center of the rectangle to the sides of the rectangle. The following illustration shows an arc (part of an ellipse) drawn by using the **Arc** function.

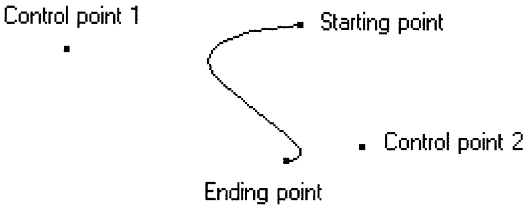


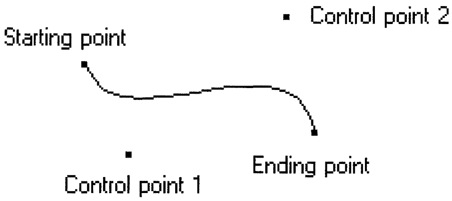
When calling the **Arc** function, an application specifies the coordinates of the bounding rectangle and radials. The preceding illustration shows the rectangle and radials with dashed lines while the actual arc was drawn using a solid line.

When drawing the arc of another object, the application can call the **SetArcDirection** and **GetArcDirection** functions to control the direction (clockwise or counterclockwise)

in which the object is drawn. The default direction for drawing arcs and other objects is counterclockwise.

In addition to drawing ellipses or parts of ellipses, Win32-based applications can draw irregular curves called Bézier curves. A *Bézier curve* is an irregular curve whose curvature is defined by four control points (*p1*, *p2*, *p3*, and *p4*). The control points *p1* and *p4* define the starting and ending points of the curve, and the control points *p2* and *p3* define the shape of the curve by marking points where the curve reverses orientation.





An application can draw irregular curves by calling the **PolyBezier** function, supplying the appropriate control points.

Combined Lines and Curves

In addition to drawing lines or curves, Win32-based applications can draw combinations of line and curve output by calling a single function. For example, an application can draw the outline of a pie chart by calling the **AngleArc** function.

The **AngleArc** function draws an arc along a circle's perimeter and draws a line connecting the starting point of the arc to the circle's center. In addition to using the **AngleArc** function, a Win32-based application can also combine line and irregular curve output by using the **PolyDraw** function.

Line and Curve Attributes

A device context (DC) contains attributes that affect line and curve output. The *line and curve attributes* include the current position, brush style, brush color, pen style, pen color, transformation, and so on.

The default current position for any DC is located at the point (0,0) in logical (or world) space. You can set these coordinates to a new position by calling the **MoveToEx** function and passing a new set of coordinates.

Note The Win32 API provides two sets of line- and curve-drawing functions. The first set retains the current position in a DC, and the second set alters the position. You can identify the functions that alter the current position by examining the function name. If the function name ends with the preposition "To", the function sets the current position to the ending point of the last line drawn (**LineTo**, **ArcTo**, **PolylineTo**, or **PolyBezierTo**). If the function name does not end with this preposition, it leaves the current position intact (**Arc**, **Polyline**, or **PolyBezier**).

The default brush is a solid white brush. An application can create a new brush by calling the **CreateBrushIndirect** function. After creating a brush, the application can select it into its DC by calling the **SelectObject** function. The Win32 API provides a complete set of functions to create, select, and alter the brush in an application's DC. For more information about these functions and about brushes in general, see *Brushes*.

The default pen is a cosmetic, solid black pen that is one pixel wide. An application can create a pen by using the **ExtCreatePen** function. After creating a pen, your application can select it into its DC by calling the **SelectObject** function. The Win32 API provides a complete set of functions to create, select, and alter the pen in an application's DC. For more information about these functions and about pens in general, see *Pens*.

The default transformation is the unity transformation (specified by the identity matrix). An application can specify a new transformation by calling the **SetWorldTransform** function. The Win32 API provides a complete set of functions to transform lines and curves by altering their width, location, and general appearance. For more information about these functions, see *Coordinate Spaces and Transformations*.

Line and Curve Reference

Line and Curve Functions

AngleArc

The **AngleArc** function draws a line segment and an arc. The line segment is drawn from the current position to the beginning of the arc. The arc is drawn along the perimeter of a circle with the given radius and center. The length of the arc is defined by the given start and sweep angles.

```
BOOL AngleArc(
  HDC hdc,            // handle to device context
  int X,              // x-coordinate of circle's center
  int Y,              // y-coordinate of circle's center
  DWORD dwRadius,     // circle's radius
  FLOAT eStartAngle,  // arc's start angle
  FLOAT eSweepAngle   // arc's sweep angle
);
```

Parameters

hdc

[in] Handle to a device context.

X

[in] Specifies the logical x-coordinate of the center of the circle.

Y
[in] Specifies the logical y-coordinate of the center of the circle.

dwRadius
[in] Specifies the radius, in logical units, of the circle. This value must be positive.

eStartAngle
[in] Specifies the start angle, in degrees, relative to the x-axis.

eSweepAngle
[in] Specifies the sweep angle, in degrees, relative to the starting angle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **AngleArc** function moves the current position to the ending point of the arc.

The arc drawn by this function may appear to be elliptical, depending on the current transformation and mapping mode. Before drawing the arc, **AngleArc** draws the line segment from the current position to the beginning of the arc.

The arc is drawn by constructing an imaginary circle around the specified center point with the specified radius. The starting point of the arc is determined by measuring counterclockwise from the x-axis of the circle by the number of degrees in the start angle. The ending point is similarly located by measuring counterclockwise from the starting point by the number of degrees in the sweep angle.

If the sweep angle is greater than 360 degrees, the arc is swept multiple times.

This function draws lines by using the current pen. The figure is not filled.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Lines and Curves Overview, Line and Curve Functions, **Arc**, **ArcTo**, **MoveToEx**

Arc

The **Arc** function draws an elliptical arc.

```
BOOL Arc(
  HDC hdc,          // handle to device context
  int nLeftRect,    // x-coord of rectangle's upper-left corner
  int nTopRect,     // y-coord of rectangle's upper-left corner
  int nRightRect,   // x-coord of rectangle's lower-right corner
  int nBottomRect,  // y-coord of rectangle's lower-right corner
  int nXStartArc,   // x-coord of first radial ending point
  int nYStartArc,   // y-coord of first radial ending point
  int nXEndArc,     // x-coord of second radial ending point
  int nYEndArc      // y-coord of second radial ending point
);
```

Parameters

hdc

[in] Handle to the device context where drawing takes place.

nLeftRect

[in] Specifies the logical x-coordinate of the upper-left corner of the bounding rectangle.

Windows 95/98: The sum of *nLeftRect* plus *nRightRect* must be less than 32768.

nTopRect

[in] Specifies the logical y-coordinate of the upper-left corner of the bounding rectangle.

Windows 95/98: The sum of *nTopRect* plus *nBottomRect* must be less than 32768.

nRightRect

[in] Specifies the logical x-coordinate of the lower-right corner of the bounding rectangle.

Windows 95/98: The sum of *nLeftRect* plus *nRightRect* must be less than 32768.

nBottomRect

[in] Specifies the logical y-coordinate of the lower-right corner of the bounding rectangle.

Windows 95/98: The sum of *nTopRect* plus *nBottomRect* must be less than 32768.

nXStartArc

[in] Specifies the logical x-coordinate of the ending point of the radial line defining the starting point of the arc.

nYStartArc

[in] Specifies the logical y-coordinate of the ending point of the radial line defining the starting point of the arc.

nXEndArc
[in] Specifies the logical x-coordinate of the ending point of the radial line defining the ending point of the arc.

nYEndArc
[in] Specifies the logical y-coordinate of the ending point of the radial line defining the ending point of the arc.

Return Values

If the arc is drawn, the return value is nonzero.

If the arc is not drawn, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The points (*nLeftRect, nTopRect*) and (*nRightRect, nBottomRect*) specify the bounding rectangle. An ellipse formed by the specified bounding rectangle defines the curve of the arc. The arc extends in the current drawing direction from the point where it intersects the radial from the center of the bounding rectangle to the *(nXStartArc, nYStartArc)* point. The arc ends where it intersects the radial from the center of the bounding rectangle to the *(nXEndArc, nYEndArc)* point. If the starting point and ending point are the same, a complete ellipse is drawn.

The arc is drawn using the current pen; it is not filled.

The current position is neither used nor updated by **Arc**.

Windows 95/98: The drawing direction is always counterclockwise.

Windows NT/2000: Use the **GetArcDirection** and **SetArcDirection** functions to get and set the current drawing direction for a device context. The default drawing direction is counterclockwise.

Windows 95/98: The sum of the coordinates of the bounding rectangle cannot exceed 32,767. The sum of *nLeftRect* and *nRightRect* or *nTopRect* and *nBottomRect* parameters cannot exceed 32,767.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **AngleArc**, **ArcTo**, **Chord**, **Ellipse**, **GetArcDirection**, **Pie**, **SetArcDirection**

ArcTo

The **ArcTo** function draws an elliptical arc.

```
BOOL ArcTo(
  HDC hdc,            // handle to device context
  int nLeftRect,      // x-coord of rectangle's upper-left corner
  int nTopRect,       // y-coord of rectangle's upper-left corner
  int nRightRect,     // x-coord of rectangle's lower-right corner
  int nBottomRect,    // y-coord of rectangle's lower-right corner
  int nXRadial1,      // x-coord of first radial ending point
  int nYRadial1,      // y-coord of first radial ending point
  int nXRadial2,      // x-coord of second radial ending point
  int nYRadial2       // y-coord of second radial ending point
);
```

Parameters

hdc

[in] Handle to the device context where drawing takes place.

nLeftRect

[in] Specifies the logical x-coordinate of the upper-left corner of the bounding rectangle.

nTopRect

[in] Specifies the logical y-coordinate of the upper-left corner of the bounding rectangle.

nRightRect

[in] Specifies the logical x-coordinate of the lower-right corner of the bounding rectangle.

nBottomRect

[in] Specifies the logical y-coordinate of the lower-right corner of the bounding rectangle.

nXRadial1

[in] Specifies the logical x-coordinate of the endpoint of the radial defining the starting point of the arc.

nYRadial1

[in] Specifies the logical y-coordinate of the endpoint of the radial defining the starting point of the arc.

nXRadial2

[in] Specifies the logical x-coordinate of the endpoint of the radial defining the ending point of the arc.

nYRadial2

[in] Specifies the logical y-coordinate of the endpoint of the radial defining the ending point of the arc.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

ArcTo is similar to the **Arc** function, except that the current position is updated.

The points (*nLeftRect, nTopRect*) and (*nRightRect, nBottomRect*) specify the bounding rectangle. An ellipse formed by the specified bounding rectangle defines the curve of the arc. The arc extends counterclockwise from the point where it intersects the radial line from the center of the bounding rectangle to the *(nXRadial1, nYRadial1)* point. The arc ends where it intersects the radial line from the center of the bounding rectangle to the *(nXRadial2, nYRadial2)* point. If the starting point and ending point are the same, a complete ellipse is drawn.

A line is drawn from the current position to the starting point of the arc. If no error occurs, the current position is set to the ending point of the arc.

The arc is drawn using the current pen; it is not filled.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **AngleArc**, **Arc**, **SetArcDirection**

GetArcDirection

The **GetArcDirection** function returns the current arc direction for the specified device context. Arc and rectangle functions use the arc direction.

```
int GetArcDirection(
  HDC hdc   // handle to device context
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

The return value specifies the current arc direction; it can be any one of the following values:

Value	Meaning
AD_COUNTERCLOCKWISE	Arcs and rectangles are drawn counterclockwise.
AD_CLOCKWISE	Arcs and rectangles are drawn clockwise.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **SetArcDirection**

LineDDA

The **LineDDA** function determines which pixels should be highlighted for a line defined by the specified starting and ending points.

```
BOOL LineDDA(
  int nXStart,             // x-coordinate of starting point
  int nYStart,             // y-coordinate of starting point
  int nXEnd,               // x-coordinate of ending point
  int nYEnd,               // y-coordinate of ending point
  LINEDDAPROC lpLineFunc,  // callback function
  LPARAM lpData            // application-defined data
);
```

Parameters

nXStart

[in] Specifies the x-coordinate of the line's starting point.

nYStart

[in] Specifies the y-coordinate of the line's starting point.

nXEnd
[in] Specifies the x-coordinate of the line's ending point.

nYEnd
[in] Specifies the y-coordinate of the line's ending point.

lpLineFunc
[in] Pointer to an application-defined callback function. For more information, see the ***LineDDAProc*** callback function.

lpData
[in] Pointer to the application-defined data.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **LineDDA** function passes the coordinates for each point along the line, except for the line's ending point, to the application-defined callback function. In addition to passing the coordinates of a point, this function passes any existing application-defined data.

The coordinates passed to the callback function match pixels on a video display only if the default transformations and mapping modes are used.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **LineDDAProc**

LineDDAProc

The **LineDDAProc** function is an application-defined callback function used with the **LineDDA** function. It is used to process coordinates. The **LINEDDAPROC** type defines a pointer to this callback function. **LineDDAProc** is a placeholder for the application-defined function name.

```
VOID CALLBACK LineDDAProc(
  int X,          // x-coordinate of point
  int Y,          // y-coordinate of point
  LPARAM lpData   // application-defined data
);
```

Parameters

X

[in] Specifies the x-coordinate of the current point.

Y

[in] Specifies the y-coordinate of the current point.

lpData

[in] Pointer to the application-defined data.

Return Values

This function does not return a value.

Remarks

An application registers a **LineDDAProc** function by passing its address to the **LineDDA** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Lines and Curves Overview, Line and Curve Functions, **LineDDA**

LineTo

The **LineTo** function draws a line from the current position up to, but not including, the specified point.

```
BOOL LineTo(
  HDC hdc,      // device context handle
  int nXEnd,    // x-coordinate of ending point
  int nYEnd     // y-coordinate of ending point
);
```

Parameters

hdc

[in] Handle to a device context.

nXEnd

[in] Specifies the x-coordinate of the line's ending point.

nYEnd

[in] Specifies the y-coordinate of the line's ending point.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The coordinates of the line's ending point are specified in logical units.

The line is drawn by using the current pen and, if the pen is a geometric pen, the current brush.

If **LineTo** succeeds, the current position is set to the specified ending point.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Lines and Curves Overview, Line and Curve Functions, **MoveToEx**, **Polyline**, **PolylineTo**

MoveToEx

The **MoveToEx** function updates the current position to the specified point and optionally returns the previous position.

```
BOOL MoveToEx(
  HDC hdc,          // handle to device context
  int X,            // x-coordinate of new current position
  int Y,            // y-coordinate of new current position
  LPPOINT lpPoint   // old current position
);
```

Parameters

hdc

[in] Handle to a device context.

X

[in] Specifies the x-coordinate of the new position, in logical units.

Y

[in] Specifies the y-coordinate of the new position, in logical units.

lpPoint

[out] Pointer to a **POINT** structure that receives the previous current position. If this parameter is a NULL pointer, the previous position is not returned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **MoveToEx** function affects all drawing functions.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Lines and Curves Overview, Line and Curve Functions, **AngleArc**, **LineTo**, **POINT**, **PolyBezierTo**, **PolylineTo**

PolyBezier

The **PolyBezier** function draws one or more Bézier curves.

```
BOOL PolyBezier(
  HDC hdc,              // handle to device context
  CONST POINT *lppt,    // endpoints and control points
  DWORD cPoints         // count of endpoints and control points
);
```

Parameters

hdc

[in] Handle to a device context.

lppt

[in] Pointer to an array of **POINT** structures that contain the endpoints and control points of the curve(s).

cPoints

[in] Specifies the number of points in the *lppt* array. This value must be one more than three times the number of curves to be drawn, because each Bézier curve requires two control points and an endpoint, and the initial curve requires an additional starting point.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Th **Polybezier** function draws cubic Bézier curves by using the endpoints and control points specified by the *lppt* parameter. The first curve is drawn from the first point to the fourth point by using the second and third points as control points. Each subsequent curve in the sequence needs exactly three more points: the ending point of the previous curve is used as the starting point, the next two points in the sequence are control points, and the third is the ending point.

The current position is neither used nor updated by the **PolyBezier** function. The figure is not filled.

This function draws lines by using the current pen.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **MoveToEx**, **POINT**, **PolyBezierTo**

PolyBezierTo

The **PolyBezierTo** function draws one or more Bézier curves.

```
BOOL PolyBezierTo(
  HDC hdc,              // handle to device context
  CONST POINT *lppt,    // endpoints and control points
  DWORD cCount          // count of endpoints and control points
);
```

Parameters

hdc

[in] Handle to a device context.

lppt

[in] Pointer to an array of **POINT** structures that contains the endpoints and control points.

cCount

[in] Specifies the number of points in the *lppt* array. This value must be three times the number of curves to be drawn, because each Bézier curve requires two control points and an ending point.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This function draws cubic Bézier curves by using the control points specified by the *lppt* parameter. The first curve is drawn from the current position to the third point by using the first two points as control points. For each subsequent curve, the function needs exactly three more points, and uses the ending point of the previous curve as the starting point for the next.

PolyBezierTo moves the current position to the ending point of the last Bézier curve. The figure is not filled.

This function draws lines by using the current pen.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **MoveToEx**, **POINT**, **PolyBezier**

PolyDraw

The **PolyDraw** function draws a set of line segments and Bézier curves.

```
BOOL PolyDraw(
  HDC hdc,               // handle to device context
  CONST POINT *lppt,     // array of points
  CONST BYTE *lpbTypes,  // line and curve identifiers
  int cCount             // count of points
);
```

Parameters

hdc
[in] Handle to a device context.

lppt
[in] Pointer to an array of **POINT** structures that contains the endpoints for each line segment and the endpoints and control points for each Bézier curve.

lpbTypes
[in] Pointer to an array that specifies how each point in the *lppt* array is used. This parameter can be one of the following values:

Type	Meaning
PT_MOVETO	Specifies that this point starts a disjoint figure. This point becomes the new current position.
PT_LINETO	Specifies that a line is to be drawn from the current position to this point, which then becomes the new current position.
PT_BEZIERTO	Specifies that this point is a control point or ending point for a Bézier curve. PT_BEZIERTO types always occur in sets of three. The current position defines the starting point for the Bézier curve. The first two PT_BEZIERTO points are the control points, and the third PT_BEZIERTO point is the ending point. The ending point becomes the new current position. If there are not three consecutive PT_BEZIERTO points, an error results.

A PT_LINETO or PT_BEZIERTO type can be combined with the following value by using the bitwise operator OR to indicate that the corresponding point is the last point in a figure and the figure is closed.

Value	Meaning
PT_CLOSEFIGURE	Specifies that the figure is automatically closed after the PT_LINETO or PT_BEZIERTO type for this point is done. A line is drawn from this point to the most recent PT_MOVETO or **MoveToEx** point. This value is combined with the PT_LINETO type for a line, or with the PT_BEZIERTO type of the ending point for a Bézier curve, by using the bitwise operator OR. The current position is set to the ending point of the closing line.

cCount

[in] Specifies the total number of points in the *lppt* array, the same as the number of bytes in the *lpbTypes* array.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **PolyDraw** function can be used in place of consecutive calls to **MoveToEx**, **LineTo**, and **PolyBezierTo** functions to draw disjoint figures. The lines and curves are drawn using the current pen and figures are not filled. If there is an active path started by calling **BeginPath**, **PolyDraw** adds to the path.

The points contained in the *lppt* array and in the *lpbTypes* array indicate whether each point is part of a **MoveTo**, **LineTo**, or **PolyBezierTo** operation. It is also possible to close figures.

This function updates the current position.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **BeginPath**, **EndPath**, **LineTo**, **MoveToEx**, **POINT**, **PolyBezierTo**, **PolyLine**

Polyline

The **Polyline** function draws a series of line segments by connecting the points in the specified array.

```
BOOL Polyline(
  HDC hdc,             // handle to device context
  CONST POINT *lppt,   // array of endpoints
  int cPoints          // number of points in array
);
```

Parameters

hdc
[in] Handle to a device context.

lppt
[in] Pointer to an array of **POINT** structures. Each structure in the array identifies a point in logical space.

cPoints
[in] Specifies the number of points in the array. This number must be greater than or equal to two.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The lines are drawn from the first point through subsequent points by using the current pen. Unlike the **LineTo** function, the **Polyline** function neither uses nor updates the current position.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **LineTo**, **MoveToEx**, **POINT**, **PolylineTo**, **PolyPolyline**

PolylineTo

The **PolylineTo** function draws one or more straight lines.

```
BOOL PolylineTo(
  HDC hdc,            // handle to device context
  CONST POINT *lppt,  // array of points
  DWORD cCount        // number of points in array
);
```

Parameters

hdc
[in] Handle to the device context.

lppt
[in] Pointer to an array of **POINT** structures that contains the vertices of the line.

cCount
[in] Specifies the number of points in the array.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A line is drawn from the current position to the first point specified by the *lppt* parameter by using the current pen. For each additional line, the function draws from the ending point of the previous line to the next point specified by *lppt*.

PolylineTo moves the current position to the ending point of the last line.

If the line segments drawn by this function form a closed figure, the figure is not filled.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **LineTo**, **MoveToEx**, **POINT**, **Polyline**

PolyPolyline

The **PolyPolyline** function draws multiple series of connected line segments.

```
BOOL PolyPolyline(
  HDC hdc,                      // handle to device context
  CONST POINT *lppt,            // array of points
  CONST DWORD *lpdwPolyPoints,  // array of values
  DWORD cCount                  // number of entries in values array
);
```

Parameters

hdc
: [in] Handle to the device context.

lppt
: [in] Pointer to an array of **POINT** structures that contains the vertices of the polylines. The polylines are specified consecutively.

lpdwPolyPoints
[in] Pointer to an array of variables specifying the number of points in the *lppt* array for the corresponding polyline. Each entry must be greater than or equal to two.

cCount
[in] Specifies the total number of entries in the *lpdwPolyPoints* array.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The line segments are drawn by using the current pen. The figures formed by the segments are not filled.

The current position is neither used nor updated by this function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions, **POINT**, **Polyline**, **PolylineTo**

SetArcDirection

The **SetArcDirection** sets the drawing direction to be used for arc and rectangle functions.

```
int SetArcDirection(
  HDC hdc,           // handle to device context
  int ArcDirection   // new arc direction
);
```

Parameters

hdc
[in] Handle to the device context.

ArcDirection

[in] Specifies the new arc direction. This parameter can be one of the following values:

Value	Meaning
AD_COUNTERCLOCKWISE	Figures drawn counterclockwise.
AD_CLOCKWISE	Figures drawn clockwise.

Return Values

If the function succeeds, the return value specifies the old arc direction.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The default direction is counterclockwise.

The **SetArcDirection** function specifies the direction in which the following functions draw:

Arc
ArcTo
Chord
Ellipse
Pie
Rectangle
RoundRect

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Lines and Curves Overview, Line and Curve Functions

CHAPTER 14

Metafiles

A *metafile* is a collection of structures that stores a picture in a device-independent format. Device independence is the one feature that sets metafiles apart from bitmaps. Unlike a bitmap, a metafile guarantees device independence. There is a drawback to metafiles, however; they are generally drawn more slowly than bitmaps. Therefore, if an application requires fast drawing, and if device independence is not an issue, it should use bitmaps instead of metafiles.

About Metafiles

Internally, a metafile is an array of variable-length structures called *metafile records.* The first records in the metafile specify general information such as the resolution of the device on which the picture was created, the dimensions of the picture, and so on. The remaining records, which constitute the bulk of any metafile, correspond to the graphical device interface (GDI) functions required to draw the picture. These records are stored in the metafile after a special metafile device context (DC) is created. This metafile device context is then used for all drawing operations required to create the picture. When the system processes a GDI function associated with a metafile DC, it converts the function into the appropriate data and stores this data in a record appended to the metafile.

After a picture is complete and the last record is stored in the metafile, you can pass the metafile to another application by:

- Using the clipboard
- Embedding it within another file
- Storing it on disk
- Playing it repeatedly

A metafile is *played* when its records are converted to device commands and processed by the appropriate device.

There are two types of metafiles:

- Enhanced-format metafiles
- Windows-format metafiles

Enhanced-Format Metafiles

An *enhanced-format metafile* is used by Win32-based applications. The enhanced format consists of the following elements:

- A header
- A table of handles to GDI objects
- A private palette
- An array of metafile records

Enhanced metafiles provide true device independence. You can think of the picture stored in an enhanced metafile as a "snapshot" of the video display taken at a particular moment. This "snapshot" maintains its dimensions no matter where it appears—on a printer, a plotter, the desktop, or in the client area of any Win32-based application.

You can use enhanced metafiles to store a picture created by using the Win32 GDI functions (including new path and transformation functions). Because the enhanced metafile format is standardized, pictures that are stored in this format can be copied from one Win32-based application to another; and, because the pictures are truly device independent, they are guaranteed to maintain their shape and proportion on any output device.

Enhanced Metafile Records

An enhanced metafile is an array of records. A metafile record is a variable-length **ENHMETARECORD** structure. At the beginning of every enhanced metafile record is an **EMR** structure, which contains two members. The first member, **iType**, identifies the record type—that is, the GDI function whose parameters are contained in the record. Because the structures are variable in length, the other member, **nSize**, contains the size of the record. Immediately following the **nSize** member are the remaining parameters, if any, of the GDI function. The remainder of the structure contains additional data that is dependent on the record type.

The first record in an enhanced metafile is always the **ENHMETAHEADER** structure, which is the enhanced-metafile header. The header specifies the following information:

- Size of the metafile, in bytes
- Dimensions of the picture frame, in device units
- Dimensions of the picture frame, in .01-millimeter units
- Number of records in the metafile
- Offset to an optional text description
- Size of the optional palette
- Resolution of the original device, in pixels
- Resolution of the original device, in millimeters

An optional text description can follow the header record. The text description describes the picture and the author's name. The optional palette specifies the colors used to create the enhanced metafile. The remaining records identify the GDI functions used to create the picture. The following hexadecimal output corresponds to a record generated for a call to the **SetMapMode** function:

```
00000011 0000000C 00000004
```

The value 0x00000011 specifies the record type (corresponds to the EMR_SETMAPMODE constant defined in the file Wingdi.h). The value 0x0000000C specifies the length of the record, in bytes. The value 0x00000004 identifies the mapping mode (corresponds to the MM_LOENGLISH constant defined in the **SetMapMode** function).

For a list of additional record types, see *Enhanced Metafile Structures*.

Enhanced Metafile Creation

You create an enhanced metafile by using the **CreateEnhMetaFile** function, supplying the appropriate arguments. The system uses these arguments to maintain picture dimensions, determine whether the metafile should be stored on a disk or in memory, and so on.

To maintain picture dimensions across output devices, **CreateEnhMetaFile** requires the resolution of the reference device. This *reference device* is the device on which the picture first appeared, and the *reference DC* is the *device context* associated with the reference device. When calling the **CreateEnhMetaFile** function, you must supply a handle that identifies this DC. You can get this handle by calling the **GetDC** or **CreateDC** function. You can also specify NULL as the handle to use the current display device for the reference device.

Most applications store pictures permanently and therefore create an enhanced metafile that is stored on a disk; however, there are some instances when this is not necessary. For example, a word-processing application that provides chart-drawing capabilities could store a user-defined chart in memory as an enhanced metafile and then copy the enhanced metafile bits from memory into the user's document file. An application that requires a metafile that is stored permanently on a disk must supply the file name when it calls **CreateEnhMetaFile**. If you do not supply a file name, the system automatically treats the metafile as a temporary file and stores it in memory.

You can add an optional text description to a metafile containing information about the picture and the author. An application can display these strings in the File Open dialog box to provide the user with information about metafile content that will help in selecting the appropriate file. If an application includes the text description, it must supply a pointer to the string when it calls **CreateEnhMetaFile**.

When **CreateEnhMetaFile** succeeds, it returns a handle that identifies a special metafile device context. A metafile device context is unique in that it is associated with a file rather than with an output device. When the system processes a GDI function that received a handle to a metafile device context, it converts the GDI function into an enhanced-metafile record and appends the record to the end of the enhanced metafile.

After a picture is complete and the last record is appended to the enhanced metafile, the application can close the file by calling the **CloseEnhMetaFile** function. This function closes and deletes the special metafile device context and returns a handle identifying the enhanced metafile.

To delete an enhanced-format metafile or an enhanced-format metafile handle, call the **DeleteEnhMetaFile** function.

Enhanced Metafile Operations

You can use the handle to an enhanced metafile to accomplish the following tasks:

- Display the picture stored in an enhanced metafile.
- Create copies of an enhanced metafile.
- Edit an enhanced metafile.
- Retrieve the optional description stored in an enhanced metafile.
- Retrieve a copy of an enhanced-metafile header.
- Retrieve a binary version of an enhanced metafile.
- Enumerate the colors in the optional palette.

These tasks are discussed in the sections in the remainder of this topic.

Display the Picture Stored in an Enhanced Metafile

You can display the picture stored in an enhanced metafile using the **PlayEnhMetaFile** function. Pass the function a handle to the enhanced metafile, without being concerned with the format of the enhanced metafile records. However, it is sometimes desirable to enumerate the records in the enhanced metafile to search for a particular GDI function and modify the parameters of the function in some manner. To do this, you can use **EnumEnhMetaFile** and provide a callback function, **EnhMetaFileProc**, to process the enhanced metafile records. To modify the parameters for an enhanced metafile record, you must know the format of the parameters within the record.

Create Copies of an Enhanced Metafile

Some applications create temporary backup (or duplicate) copies of a file before enabling the user to alter the original. An application can create a backup copy of an enhanced metafile by calling the **CopyEnhMetaFile** function, supplying a handle that identifies the enhanced metafile, and supplying a pointer to the name of the new file.

To create a memory-based enhanced-format metafile, call the **SetEnhMetaFileBits** function.

Most drawing, illustration, and computer-aided design (CAD) applications require a means of editing a picture stored in an enhanced metafile. Although editing an enhanced metafile is a complex task, you can use the **EnumEnhMetaFile** function in combination with other functions to provide this capability in your application. The **EnumEnhMetaFile** function and its associated callback function, **EnhMetaFileProc**, enable the application to process individual records in an enhanced metafile.

Retrieve the Optional Description Stored in an Enhanced Metafile

Some applications display the text description of an enhanced metafile with the corresponding file name in the **Open** dialog box. You can determine whether this string exists in an enhanced metafile by retrieving the metafile header with the **GetEnhMetaFileHeader** function and examining one of its members. If the string exists, the application retrieves it by calling the **GetEnhMetaFileDescription** function.

Retrieve a Binary Version of an Enhanced Metafile

You can retrieve the contents of a metafile by calling the **GetEnhMetaFileBits** function; however, before retrieving the contents, you must specify the size of the file. To get the size, you can use the **GetEnhMetaFileHeader** function and examine the appropriate member.

Enumerate the Colors in the Optional Palette

To achieve consistent colors when a picture is displayed on various output devices, you can call the **CreatePalette** function and store a logical palette in an enhanced metafile. An application that displays the picture stored in the enhanced metafile retrieves this palette and calls the **RealizePalette** function before displaying the picture. To determine whether a palette is stored in an enhanced metafile, retrieve the metafile header and examine the appropriate member. If a palette exists, you can call the **GetEnhMetaFilePaletteEntries** function to retrieve the logical palette.

Windows-Format Metafiles

Windows-format metafiles are limited in their capabilities and should rarely be used—the Windows-format functions are supported to maintain backward compatibility with applications that were written to run as 16-bit Windows-based applications. Instead, you should use the enhanced-format functions.

A *Windows-format metafile* is used by 16-bit Windows-based applications. The format consists of a header and an array of metafile records.

The following are the limitations of this format:

- A Windows-format metafile is application and device dependent. Changes in the application's mapping modes or the device resolution affect the appearance of metafiles created in this format.
- A Windows-format metafile does not contain a comprehensive header that describes the original picture dimensions, the resolution of the device on which the picture was created, an optional text description, or an optional palette.

- A Windows-format metafile does not support the new curve, path, and transformation functions. See the list of supported functions in the table that follows.
- Some Windows-format metafile records cannot be scaled.
- The metafile device context associated with a Windows-format metafile cannot be queried (that is, an application cannot retrieve device-resolution data, font metrics, and so on).

Following are the only functions that are supported in Windows-format metafiles:

AnimatePalette
Arc
BitBlt
Chord
CreateBrushIndirect
CreateDIBPatternBrush
CreateFontIndirect
CreatePalette
CreatePatternBrush
CreatePenIndirect
DeleteObject
Ellipse
Escape
ExcludeClipRect
ExtFloodFill
ExtTextOut
FillRgn
FloodFill
FrameRgn
IntersectClipRect
InvertRgn
LineTo
MoveToEx
OffsetClipRgn
OffsetViewportOrgEx
OffsetWindowOrgEx
PaintRgn
PatBlt
Pie
Polygon
Polyline
PolyPolygon
RealizePalette
Rectangle
ResizePalette
RestoreDC
RoundRect
SaveDC
ScaleViewportExtEx
ScaleWindowExtEx
SelectClipRgn
SelectObject
SelectPalette
SetBkColor
SetBkMode
SetDIBitsToDevice
SetMapMode
SetMapperFlags
SetPaletteEntries
SetPixel
SetPolyFillMode
SetROP2
SetStretchBltMode
SetTextAlign
SetTextCharacterExtra
SetTextColor
SetTextJustification
SetViewportOrgEx
SetWindowExtEx
SetWindowOrgEx
StretchBlt
StretchDIBits
TextOut

To convert a Windows-format metafile to an enhanced-format metafile, call the **GetMetaFileBitsEx** function to retrieve the data from the Windows-format metafile and then call the **SetWinMetaFileBits** function to convert this data into an enhanced-format metafile. To convert an enhanced-format record into a Windows-format record, call the **GetWinMetaFileBits** function.

Metafile Reference

Metafile Functions

CloseEnhMetaFile

The **CloseEnhMetaFile** function closes an enhanced-metafile device context and returns a handle that identifies an enhanced-format metafile.

```
HENHMETAFILE CloseEnhMetaFile(
  HDC hdc   // handle to enhanced-metafile DC
);
```

Parameters

hdc

[in] Handle to an enhanced-metafile device context.

Return Values

If the function succeeds, the return value is a handle to an enhanced metafile.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can use the enhanced-metafile handle returned by the **CloseEnhMetaFile** function to perform the following tasks:

- Display a picture stored in an enhanced metafile.
- Create copies of the enhanced metafile.
- Enumerate, edit, or copy individual records in the enhanced metafile.
- Retrieve an optional description of the metafile contents from the enhanced-metafile header.
- Retrieve a copy of the enhanced-metafile header.
- Retrieve a binary copy of the enhanced metafile.
- Enumerate the colors in the optional palette.
- Convert an enhanced-format metafile into a Windows-format metafile.

When the application no longer needs the enhanced metafile handle, it should release the handle by calling the **DeleteEnhMetaFile** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **CopyEnhMetaFile**, **CreateEnhMetaFile**, **DeleteEnhMetaFile**, **EnumEnhMetaFile**, **GetEnhMetaFileBits**, **GetWinMetaFileBits**, **PlayEnhMetaFile**

CopyEnhMetaFile

The **CopyEnhMetaFile** function copies the contents of an enhanced-format metafile to a specified file.

```
HENHMETAFILE CopyEnhMetaFile(
  HENHMETAFILE hemfSrc,    // handle to enhanced metafile
  LPCTSTR lpszFile         // file name
);
```

Parameters

hemfSrc
[in] Handle to the source-enhanced metafile.

lpszFile
[in] Pointer to the name of the destination file. If this parameter is NULL, the source metafile is copied to memory.

Return Values

If the function succeeds, the return value is a handle to the copy of the enhanced metafile.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Where text arguments must use Unicode characters, use the **CopyEnhMetaFile** function as a wide-character function. Where text arguments must use characters from the Windows character set, use this function as an ANSI function.

Applications can use metafiles stored in memory for temporary operations.

When the application no longer needs the enhanced-metafile handle, it should delete the handle by calling the **DeleteEnhMetaFile** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Metafiles Overview, Metafile Functions, **DeleteEnhMetaFile**

CreateEnhMetaFile

The **CreateEnhMetaFile** function creates a device context for an enhanced-format metafile. This device context can be used to store a device-independent picture.

```
HDC CreateEnhMetaFile(
  HDC hdcRef,              // handle to reference DC
  LPCTSTR lpFilename,      // file name
  CONST RECT *lpRect,      // bounding rectangle
  LPCTSTR lpDescription    // description string
);
```

Parameters

hdcRef
[in] Handle to a reference device for the enhanced metafile.

lpFilename
[in] Pointer to the file name for the enhanced metafile to be created. If this parameter is NULL, the enhanced metafile is memory based and its contents are lost when it is deleted by using the **DeleteEnhMetaFile** function.

lpRect
[in] Pointer to a **RECT** structure that specifies the dimensions (in .01-millimeter units) of the picture to be stored in the enhanced metafile.

lpDescription
[in] Pointer to a string that specifies the name of the application that created the picture, as well as the picture's title.

Return Values

If the function succeeds, the return value is a handle to the device context for the enhanced metafile.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Where text arguments must use Unicode characters, use the **CreateEnhMetaFile** function as a wide-character function. Where text arguments must use characters from the Windows character set, use this function as an ANSI function.

The system uses the reference device identified by the *hdcRef* parameter to record the resolution and units of the device on which a picture originally appeared. If the *hdcRef* parameter is NULL, it uses the current display device for reference.

The **left** and **top** members of the **RECT** structure pointed to by the *lpRect* parameter must be less than the **right** and **bottom** members, respectively. Points along the edges of the rectangle are included in the picture. If *lpRect* is NULL, the graphical device interface (GDI) computes the dimensions of the smallest rectangle that surrounds the picture drawn by the application. The *lpRect* parameter should be provided where possible.

The string pointed to by the *lpDescription* parameter must contain a null character between the application name and the picture name and must terminate with two null characters—for example, "XYZ Graphics Editor\0Bald Eagle\0\0", where \0 represents the null character. If *lpDescription* is NULL, there is no corresponding entry in the enhanced-metafile header.

Applications use the device context created by this function to store a graphics picture in an enhanced metafile. The handle identifying this device context can be passed to any GDI function.

After an application stores a picture in an enhanced metafile, it can display the picture on any output device by calling the **PlayEnhMetaFile** function. When displaying the picture, the system uses the rectangle pointed to by the *lpRect* parameter and the resolution data from the reference device to position and scale the picture.

The device context returned by this function contains the same default attributes associated with any new device context.

Applications must use the **GetWinMetaFileBits** function to convert an enhanced metafile to the older Windows metafile format.

The file name for the enhanced metafile should use the .emf extension.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Metafiles Overview, Metafile Functions, **CloseEnhMetaFile**, **DeleteEnhMetaFile**, **GetEnhMetaFileDescription**, **GetEnhMetaFileHeader**, **GetWinMetaFileBits**, **PlayEnhMetaFile**, **RECT**

DeleteEnhMetaFile

The **DeleteEnhMetaFile** function deletes an enhanced-format metafile or an enhanced-format metafile handle.

```
BOOL DeleteEnhMetaFile(
  HENHMETAFILE hemf   // handle to an enhanced metafile
);
```

Parameters

hemf

[in] Handle to an enhanced metafile.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If the *hemf* parameter identifies an enhanced metafile stored in memory, the **DeleteEnhMetaFile** function deletes the metafile. If *hemf* identifies a metafile stored on a disk, the function deletes the metafile handle but does not destroy the actual metafile. An application can retrieve the file by calling the **GetEnhMetaFile** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.

Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **CopyEnhMetaFile**, **CreateEnhMetaFile**, **GetEnhMetaFile**

EnhMetaFileProc

The **EnhMetaFileProc** function is an application-defined callback function used with the **EnumEnhMetaFile** function. The **ENHMFENUMPROC** type defines a pointer to this callback function. **EnhMetaFileProc** is a placeholder for the application-defined function name.

```
int CALLBACK EnhMetaFileProc(
  HDC hDC,                       // handle to DC
  HANDLETABLE *lpHTable,         // metafile handle table
  CONST ENHMETARECORD *lpEMFR,   // metafile record
  int nObj,                      // count of objects
  LPARAM lpData                  // optional data
);
```

Parameters

hDC

[in] Handle to the device context passed to **EnumEnhMetaFile**.

lpHTable

[in] Pointer to a **HANDLETABLE** structure representing the table of handles associated with the graphics objects (pens, brushes, and so on) in the metafile. The first entry contains the enhanced-metafile handle.

lpEMFR

[in] Pointer to one of the records in the metafile. This record should not be modified. (If modification is necessary, it should be performed on a copy of the record.)

nObj

[in] Specifies the number of objects with associated handles in the handle table.

lpData

[in] Pointer to optional data.

Return Values

This function must return a nonzero value to continue enumeration; to stop enumeration, it must return zero.

Remarks

An application must register the callback function by passing its address to the **EnumEnhMetaFile** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Metafile Functions, **ENHMETARECORD**, **EnumEnhMetaFile**, **HANDLETABLE**

EnumEnhMetaFile

The **EnumEnhMetaFile** function enumerates the records within an enhanced-format metafile by retrieving each record and passing it to the specified callback function. The application-supplied callback function processes each record as required. The enumeration continues until the last record is processed or when the callback function returns zero.

```
BOOL EnumEnhMetaFile(
  HDC hdc,                        // handle to DC
  HENHMETAFILE hemf,              // handle to enhanced metafile
  ENHMFENUMPROC lpEnhMetaFunc,    // callback function
  LPVOID lpData,                  // callback-function data
  CONST RECT *lpRect              // bounding rectangle
);
```

Parameters

hdc
[in] Handle to a device context. This handle is passed to the callback function.

hemf
[in] Handle to an enhanced metafile.

lpEnhMetaFunc
[in] Pointer to the application-supplied callback function. For more information, see ***EnhMetaFileProc***.

lpData
[in] Pointer to optional callback-function data.

lpRect

[in] Pointer to a **RECT** structure that specifies the coordinates of the picture's upper-left and lower-right corners. The dimensions of this rectangle are specified in logical units.

Return Values

If the callback function successfully enumerates all the records in the enhanced metafile, the return value is nonzero.

If the callback function does not successfully enumerate all the records in the enhanced metafile, the return value is zero.

Remarks

Points along the edge of the rectangle pointed to by the *lpRect* parameter are included in the picture. If the *hdc* parameter is NULL, the system ignores *lpRect*.

If the callback function calls the **PlayEnhMetaFileRecord** function, *hdc* must identify a valid device context. The system uses the device context's transformation and mapping mode to transform the picture displayed by the **PlayEnhMetaFileRecord** function.

You can use the **EnumEnhMetaFile** function to embed one enhanced-metafile within another.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Metafiles Overview, Metafile Functions, **EnhMetaFileProc**, **PlayEnhMetaFile**, **PlayEnhMetaFileRecord**, **RECT**

GdiComment

The **GdiComment** function copies a comment from a buffer into a specified enhanced-format metafile.

```
BOOL GdiComment(
  HDC hdc,              // handle to a device context
  UINT cbSize,          // size of text buffer
  CONST BYTE *lpData    // text buffer
);
```

Parameters

hdc

[in] Handle to an enhanced-metafile device context.

cbSize

[in] Specifies the length of the comment buffer, in bytes.

lpData

[in] Pointer to the buffer that contains the comment.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A comment can include any kind of private information—for example, the source of a picture and the date it was created. A comment should begin with an application signature, followed by the data.

Comments should not contain application-specific or position-specific data. Position-specific data specifies the location of a record, and it should not be included because one metafile may be embedded within another metafile.

A public comment is a comment that begins with the comment signature identifier GDICOMMENT_IDENTIFIER. The following public comments are defined:

Public comment	Definition
GDICOMMENT_WINDOWS_METAFILE	The GDICOMMENT_WINDOWS_METAFILE public comment contains a Windows-format metafile that is equivalent to an enhanced-format metafile. This comment is written only by the **SetWinMetaFileBits** function. The comment record, if given, follows the **ENHMETAHEADER** metafile record. The comment has the following form:

```
DWORD ident;        // This contains GDICOMMENT_IDENTIFIER.
DWORD iComment;     // This contains GDICOMMENT_WINDOWS_METAFILE.
DWORD nVersion;     // This contains the version number of
                    // the Windows-format metafile.
DWORD nChecksum;    // This is the additive DWORD checksum
                    // for the enhanced metafile.  The
                    // checksum for the enhanced metafile
                    // data including this comment record
                    // must be zero.  Otherwise, the
```

(continued)

(continued)

```
                    // enhanced metafile has been modified
                    // and the Windows-format metafile is
                    //  no longer valid.
DWORD fFlags;       // This must be zero.
DWORD cbWinMetaFile; // This is the size, in bytes, of the
                    // Windows-format metafile data that
                    // follows.
```

GDICOMMENT_BEGINGROUP

The GDICOMMENT_BEGINGROUP public comment identifies the beginning of a group of drawing records. It identifies an object within an enhanced metafile. The comment has the following form:

```
DWORD   ident;        // This contains GDICOMMENT_IDENTIFIER.
DWORD   iComment;     // This contains GDICOMMENT_BEGINGROUP.
RECTL   rclOutput;    // This is the bounding rectangle
                      // for the object in logical
                      // coordinates.
DWORD   nDescription; // This is the number of characters
                      // in the optional unicode
                      // description string that follows.
                      // This is zero if there is no
                      // description string.
```

GDICOMMENT_ENDGROUP

The GDICOMMENT_ENDGROUP public comment identifies the end of a group of drawing records. The GDICOMMENT_BEGINGROUP comment and the GDICOMMENT_ENDGROUP comment must be included in a pair and may be nested. The comment has the following form:

```
DWORD   ident;     // This contains GDICOMMENT_IDENTIFIER.
DWORD   iComment;  // This contains GDICOMMENT_ENDGROUP.
```

GDICOMMENT_MULTIFORMATS

Windows NT 4.0 SP4 and earlier, Windows 95/98: The GDICOMMENT_MULTIFORMATS public comment allows multiple definitions of a picture to be included in an enhanced metafile. Using this comment, for example, an application can include an encapsulated PostScript definition as well as an enhanced metafile definition of a picture. When the record is played back, GDI selects and renders the first format recognized by the device. The comment has the following form:

```
DWORD   ident;            // This contains GDICOMMENT_IDENTIFIER.
DWORD   iComment;         // This contains GDICOMMENT_MULTIFORMATS.
RECTL   rclOutput;        // This is the bounding rectangle
                          // for the picture in logical
                          // coordinates.
DWORD   nFormats;         // This contains the number of
                          // formats in the comment.
EMRFORMAT aemrformat[1];// This is an array of EMRFORMAT
                          // structures in the order of
                          // preference. The data for each
                          // format follows the last
                          // EMRFORMAT structure.
```

Windows 2000: The GDICOMMENT_MULTIFORMATS flag is not supported for EPS data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **CreateEnhMetaFile**, **SetWinMetaFileBits**

GetEnhMetaFile

The **GetEnhMetaFile** function creates a handle that identifies the enhanced-format metafile stored in the specified file.

```
HENHMETAFILE GetEnhMetaFile(
  LPCTSTR lpszMetaFile   // file name
);
```

Parameters

lpszMetaFile
[in] Pointer to a null-terminated string that specifies the name of an enhanced metafile.

Return Values

If the function succeeds, the return value is a handle to the enhanced metafile.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When the application no longer needs an enhanced-metafile handle, it should delete the handle by calling the **DeleteEnhMetaFile** function.

A Windows-format metafile must be converted to the enhanced format before it can be processed by the **GetEnhMetaFile** function. To convert the file, use the **SetWinMetaFileBits** function.

Where text arguments must use Unicode characters, use this function as a wide-character function. Where text arguments must use characters from the Windows character set, use this function as an ANSI function.

Windows 95/98: The maximum length of the description string for an enhanced metafile is 16,384 bytes.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Metafiles Overview, Metafile Functions, **DeleteEnhMetaFile**, **GetEnhMetaFile**, **SetWinMetaFileBits**

GetEnhMetaFileBits

The **GetEnhMetaFileBits** function retrieves the contents of the specified enhanced-format metafile and copies them into a buffer.

```
UINT GetEnhMetaFileBits(
  HENHMETAFILE hemf,  // handle to metafile
  UINT cbBuffer,      // size of data buffer
  LPBYTE lpbBuffer    // data buffer
);
```

Parameters

hemf

[in] Handle to the enhanced metafile.

cbBuffer

[in] Specifies the size, in bytes, of the buffer to receive the data.

lpbBuffer

[out] Pointer to a buffer that receives the metafile data. The buffer must be sufficiently large to contain the data. If *lpbBuffer* is NULL, the function returns the size necessary to hold the data.

Return Values

If the function succeeds and the buffer pointer is NULL, the return value is the size of the enhanced metafile, in bytes.

If the function succeeds and the buffer pointer is a valid pointer, the return value is the number of bytes copied to the buffer.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

After the enhanced-metafile bits are retrieved, they can be used to create a memory-based metafile by calling the **SetEnhMetaFileBits** function.

The **GetEnhMetaFileBits** function does not invalidate the enhanced-metafile handle. The application must call the **DeleteEnhMetaFile** function to delete the handle when it is no longer needed.

The metafile contents retrieved by this function are in the enhanced format. To retrieve the metafile contents in the Windows format, use the **GetWinMetaFileBits** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Metafiles Overview, Metafile Functions, **DeleteEnhMetaFile**, **GetWinMetaFileBits**, **SetEnhMetaFileBits**

GetEnhMetaFileDescription

The **GetEnhMetaFileDescription** function retrieves an optional text description from an enhanced-format metafile and copies the string to the specified buffer.

```
UINT GetEnhMetaFileDescription(
  HENHMETAFILE hemf,        // handle to enhanced metafile
  UINT cchBuffer,           // size of text buffer
  LPTSTR lpszDescription    // text buffer
);
```

Parameters

hemf

[in] Handle to the enhanced metafile.

cchBuffer

[in] Specifies the size, in characters, of the buffer to receive the data. Only this many characters will be copied.

lpszDescription

[out] Pointer to a buffer that receives the optional text description.

Return Values

If the optional text description exists and the buffer pointer is NULL, the return value is the length of the text string, in characters.

If the optional text description exists and the buffer pointer is a valid pointer, the return value is the number of characters copied into the buffer.

If the optional text description does not exist, the return value is zero.

If the function fails, the return value is GDI_ERROR.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The optional text description contains two strings, the first identifying the application that created the enhanced metafile and the second identifying the picture contained in the metafile. The strings are separated by a null character and terminated with two null characters—for example, “XYZ Graphics Editor\0Bald Eagle\0\0” where \0 represents the null character.

Where text arguments must use Unicode characters, use this function as a wide-character function. Where text arguments must use characters from the Windows character set, use this function as an ANSI function.

Windows 95/98: The maximum length of the description string for an enhanced metafile is 16,384 bytes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Metafiles Overview, Metafile Functions, **CreateEnhMetaFile**

GetEnhMetaFileHeader

The **GetEnhMetaFileHeader** function retrieves the record containing the header for the specified enhanced-format metafile.

```
UINT GetEnhMetaFileHeader(
  HENHMETAFILE hemf,        // handle to enhanced metafile
  UINT cbBuffer,            // size of buffer
  LPENHMETAHEADER lpemh     // data buffer
);
```

Parameters

hemf

[in] Handle to the enhanced metafile for which the header is to be retrieved.

cbBuffer

[in] Specifies the size, in bytes, of the buffer to receive the data. Only this many bytes will be copied.

lpemh

[out] Pointer to an **ENHMETAHEADER** structure that receives the header record. If this parameter is NULL, the function returns the size of the header record.

Return Values

If the function succeeds and the structure pointer is NULL, the return value is the size of the record that contains the header; if the structure pointer is a valid pointer, the return value is the number of bytes copied. Otherwise, it is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An enhanced-metafile header contains such information as the metafile's size, in bytes; the dimensions of the picture stored in the metafile; the number of records stored in the metafile; the offset to the optional text description; the size of the optional palette, and the resolution of the device on which the picture was created.

The record that contains the enhanced-metafile header is always the first record in the metafile.

Windows 95/98: The maximum length of the description string for an enhanced metafile is 16,384 bytes.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **ENHMETAHEADER**, **PlayEnhMetaFile**

GetEnhMetaFilePaletteEntries

The **GetEnhMetaFilePaletteEntries** function retrieves optional palette entries from the specified enhanced metafile.

```
UINT GetEnhMetaFilePaletteEntries(
  HENHMETAFILE hemf,    // handle to enhanced metafile
  UINT cEntries,        // count of palette entries
  LPPALETTEENTRY lppe   // array of palette entries
);
```

Parameters

hemf

[in] Handle to the enhanced metafile.

cEntries

[in] Specifies the number of entries to be retrieved from the optional palette.

lppe

[out] Pointer to an array of **PALETTEENTRY** structures that receives the palette colors. The array must contain at least as many structures as there are entries specified by the *cEntries* parameter.

Return Values

If the array pointer is NULL and the enhanced metafile contains an optional palette, the return value is the number of entries in the enhanced metafile's palette; if the array pointer is a valid pointer and the enhanced metafile contains an optional palette, the return value is the number of entries copied; if the metafile does not contain an optional palette, the return value is zero. Otherwise, the return value is GDI_ERROR.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can store an optional palette in an enhanced metafile by calling the **CreatePalette** and **SetPaletteEntries** functions before creating the picture and storing it in the metafile. By doing this, the application can achieve consistent colors when the picture is displayed on a variety of devices.

An application that displays a picture stored in an enhanced metafile can call the **GetEnhMetaFilePaletteEntries** function to determine whether the optional palette exists. If it does, the application can call the **GetEnhMetaFilePaletteEntries** function a second time to retrieve the palette entries and then create a logical palette (by using the **CreatePalette** function), select it into its device context (by using the **SelectPalette** function), and then realize it (by using the **RealizePalette** function). After the logical palette has been realized, calling the **PlayEnhMetaFile** function displays the picture using its original colors.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **CreatePalette**, **PALETTEENTRY**, **PlayEnhMetaFile**, **RealizePalette**, **SelectPalette**

GetWinMetaFileBits

The **GetWinMetaFileBits** function converts the enhanced-format records from a metafile into Windows-format records and stores the converted records in the specified buffer.

```
UINT GetWinMetaFileBits(
  HENHMETAFILE hemf, // handle to the enhanced metafile
  UINT cbBuffer,     // buffer size
  LPBYTE lpbBuffer,  // records buffer
  INT fnMapMode,     // mapping mode
  HDC hdcRef         // handle to reference DC
);
```

Parameters

hemf

[in] Handle to the enhanced metafile.

cbBuffer

[in] Specifies the size, in bytes, of the buffer into which the converted records are to be copied.

lpbBuffer

[out] Pointer to the buffer that receives the converted records. If *lpbBuffer* is NULL, **GetWinMetaFileBits** returns the number of bytes required to store the converted metafile records.

fnMapMode

[in] Specifies the mapping mode to use in the converted metafile.

hdcRef

[in] Handle to the reference device context.

Return Values

If the function succeeds and the buffer pointer is NULL, the return value is the number of bytes required to store the converted records; if the function succeeds and the buffer pointer is a valid pointer, the return value is the size of the metafile data in bytes.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This function converts an enhanced metafile into a Windows-format metafile so that its picture can be displayed in an application that recognizes the older format.

The system uses the reference device context to determine the resolution of the converted metafile.

The **GetWinMetaFileBits** function does not invalidate the enhanced metafile handle. An application should call the **DeleteEnhMetaFile** function to release the handle when it is no longer needed.

Due to the limitations of the Windows-format metafile, some information can be lost in the retrieved metafile contents. For example, an original call to the **PolyBezier** function in the enhanced metafile may be converted into a call to the **Polyline** function in the

Windows-format metafile, because there is no equivalent **PolyBezier** function in the Windows format.

16-bit Windows-based applications define the viewport origin and extents of a picture stored in a Windows-format metafile. As a result, the Windows-format records created by **GetWinMetaFileBits** do not contain the **SetViewportOrgEx** and **SetViewportExtEx** functions. However, **GetWinMetaFileBits** does create Windows-format records for the **SetWindowExtEx** and **SetMapMode** functions.

To create a scalable Windows-format metafile, specify MM_ANISOTROPIC as the *fnMapMode* parameter.

The upper-left corner of the metafile picture is always mapped to the origin of the reference device.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **DeleteEnhMetaFile**, **PolyBezier**, **Polyline**, **SetMapMode**, **SetViewportOrgEx**, **SetViewportExtEx**, **SetWindowExtEx**, **SetWinMetaFileBits**

PlayEnhMetaFile

The **PlayEnhMetaFile** function displays the picture stored in the specified enhanced-format metafile.

```
BOOL PlayEnhMetaFile(
  HDC hdc,              // handle to DC
  HENHMETAFILE hemf,    // handle to an enhanced metafile
  CONST RECT *lpRect    // bounding rectangle
);
```

Parameters

hdc

[in] Handle to the device context for the output device on which the picture will appear.

hemf

[in] Handle to the enhanced metafile.

lpRect

[in] Pointer to a **RECT** structure that contains the coordinates of the bounding rectangle used to display the picture. The coordinates are specified in logical units.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When an application calls the **PlayEnhMetaFile** function, the system uses the picture frame in the enhanced-metafile header to map the picture onto the rectangle pointed to by the *lpRect* parameter. (This picture may be sheared or rotated by setting the world transform in the output device before calling **PlayEnhMetaFile**.) Points along the edges of the rectangle are included in the picture.

An enhanced-metafile picture can be clipped by defining the clipping region in the output device before playing the enhanced metafile.

If an enhanced metafile contains an optional palette, an application can achieve consistent colors by setting up a color palette on the output device before calling **PlayEnhMetaFile**. To retrieve the optional palette, use the **GetEnhMetaFilePaletteEntries** function.

An enhanced metafile can be embedded in a newly created enhanced metafile by calling **PlayEnhMetaFile** and playing the source enhanced metafile into the device context for the new enhanced metafile.

The states of the output device context are preserved by this function. Any object created but not deleted in the enhanced metafile is deleted by this function.

To stop this function, an application can call the **CancelDC** function from another thread to terminate the operation. In this case, the function returns FALSE.

Windows 95/98: PlayEnhMetaFile is subject to the limitations of the GDI. For example, Windows 95/98 supports only 16-bit signed coordinates. For records that contain 32-bit values, Windows 95/98 fails to play the record if the values are not in the range –32,768 to 32,767.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Metafiles Overview, Metafile Functions, **CancelDC**, **GetEnhMetaFileHeader**, **GetEnhMetaFilePaletteEntries**, **RECT**, **SetWorldTransform**

PlayEnhMetaFileRecord

The **PlayEnhMetaFileRecord** function plays an enhanced-metafile record by executing the graphics device interface (GDI) functions identified by the record.

```
BOOL PlayEnhMetaFileRecord(
  HDC hdc,                              // handle to DC
  LPHANDLETABLE lpHandletable,          // metafile handle table
  CONST ENHMETARECORD *lpEnhMetaRecord, // metafile record
  UINT nHandles                         // count of handles
);
```

Parameters

hdc

[in] Handle to the device context passed to the **EnumEnhMetaFile** function.

lpHandletable

[in] Pointer to a table of handles to GDI objects used when playing the metafile. The first entry in this table contains the enhanced-metafile handle.

lpEnhMetaRecord

[in] Pointer to the enhanced-metafile record to be played.

nHandles

[in] Specifies the number of handles in the handle table.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This is an enhanced-metafile function.

An application typically uses **PlayEnhMetaFileRecord** in conjunction with the **EnumEnhMetaFile** function to process and play an enhanced-format metafile one record at a time.

The *hdc*, *lpHandletable*, and *nHandles* parameters must be exactly those passed to the *EnhMetaFileProc* callback procedure by the **EnumEnhMetaFile** function.

If **PlayEnhMetaFileRecord** does not recognize a record, it ignores the record and returns TRUE.

Windows 95/98: PlayEnhMetaFileRecord is subject to the limitations of GDI. For example, Windows 95/98 supports only 16-bit signed coordinates. For records that contain 32-bit values, Windows 95/98 fails to play the record if the values are not in the range –32,768 to 32,767.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Metafiles Overview, Metafile Functions, **EnumEnhMetaFile**, **PlayEnhMetaFile**

SetEnhMetaFileBits

The **SetEnhMetaFileBits** function creates a memory-based enhanced-format metafile from the supplied data.

```
HENHMETAFILE SetEnhMetaFileBits(
  UINT cbBuffer,      // buffer size
  CONST BYTE *lpData  // enhanced metafile data buffer
);
```

Parameters

cbBuffer
  [in] Specifies the size, in bytes, of the data provided.

lpData
  [in] Pointer to a buffer that contains enhanced-metafile data. (It is assumed that the data in the buffer was obtained by calling the **GetEnhMetaFileBits** function.)

Return Values

If the function succeeds, the return value is a handle to a memory-based enhanced metafile.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When the application no longer needs the enhanced-metafile handle, it should delete the handle by calling the **DeleteEnhMetaFile** function.

The **SetEnhMetaFileBits** function does not accept metafile data in the Windows format. To import Windows-format metafiles, use the **SetWinMetaFileBits** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Metafiles Overview, Metafile Functions, **DeleteEnhMetaFile**, **GetEnhMetaFileBits**, **SetWinMetaFileBits**

SetWinMetaFileBits

The **SetWinMetaFileBits** function converts a metafile from the older Windows format to the new enhanced format and stores the new metafile in memory.

```
HENHMETAFILE SetWinMetaFileBits(
  UINT cbBuffer,                // size of buffer
  CONST BYTE *lpbBuffer,        // metafile data buffer
  HDC hdcRef,                   // handle to reference DC
  CONST METAFILEPICT *lpmfp     // size of metafile picture
);
```

Parameters

cbBuffer
[in] Specifies the size, in bytes, of the buffer that contains the Windows-format metafile.

lpbBuffer
[in] Pointer to a buffer that contains the Windows-format metafile data. (It is assumed that the data was obtained by using the **GetMetaFileBitsEx** or **GetWinMetaFileBits** function.)

hdcRef
[in] Handle to a reference device context.

lpmfp
[in] Pointer to a **METAFILEPICT** structure that contains the suggested size of the metafile picture and the mapping mode that was used when the picture was created.

Return Values

If the function succeeds, the return value is a handle to a memory-based enhanced metafile.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The Win32 API uses the reference device context's resolution data and the data in the **METAFILEPICT** structure to scale a picture. If the *hdcRef* parameter is NULL, the system uses resolution data for the current output device. If the *lpmfp* parameter is NULL, the system uses the MM_ANISOTROPIC mapping mode to scale the picture so that it fits the entire device surface. The *hMF* field in the **METAFILEPICT** structure is not used.

When the application no longer needs the enhanced metafile handle, it should delete it by calling the **DeleteEnhMetaFile** function.

The handle returned by this function can be used with other enhanced-metafile functions.

If the reference device context is not identical to the device in which the metafile was originally created, some GDI functions that use device units may not draw the picture correctly.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Metafiles Overview, Metafile Functions, **DeleteEnhMetaFile**, **GetWinMetaFileBits**, **GetMetaFileBitsEx**, **METAFILEPICT**, **PlayEnhMetaFile**

Metafile Structures

Enhanced Metafile Structures

The following structures are used with enhanced metafile records. Note that the first structure, **EMR**, is used as the first member of the remaining structures.

EMR

The **EMR** structure provides the base structure for all enhanced metafile records. An enhanced metafile record contains the parameters for a specific GDI function used to create part of a picture in an enhanced format metafile.

```
typedef struct tagEMR {
  DWORD iType;
  DWORD nSize;
} EMR, *PEMR;
```

Members

iType

Specifies the record type. The parameter can be one of the following (with a link to the associated record structure):

EMR_ABORTPATH
EMR_ANGLEARC
EMR_ARC
EMR_ARCTO
EMR_BEGINPATH
EMR_BITBLT
EMR_CHORD
EMR_CLOSEFIGURE
EMR_CREATEBRUSHINDIRECT
EMR_CREATEDIBPATTERNBRUSHPT
EMR_CREATEMONOBRUSH
EMR_CREATEPALETTE
EMR_CREATEPEN
EMR_DELETEOBJECT
EMR_ELLIPSE
EMR_POLYLINE16
EMR_POLYLINETO
EMR_POLYLINETO16
EMR_POLYPOLYGON
EMR_POLYPOLYGON16
EMR_POLYPOLYLINE
EMR_POLYPOLYLINE16
EMR_POLYTEXTOUTA
EMR_POLYTEXTOUTW
EMR_REALIZEPALETTE
EMR_RECTANGLE
EMR_RESIZEPALETTE
EMR_RESTOREDC
EMR_ROUNDRECT
EMR_SAVEDC

(continued)

(continued)

EMR_ENDPATH
EMR_EOF
EMR_EXCLUDECLIPRECT
EMR_EXTCREATEFONTINDIRECTW
EMR_EXTCREATEPEN
EMR_EXTFLOODFILL
EMR_EXTSELECTCLIPRGN
EMR_EXTTEXTOUTA
EMR_EXTTEXTOUTW
EMR_FILLPATH
EMR_FILLRGN
EMR_FLATTENPATH
EMR_FRAMERGN
EMR_GDICOMMENT
EMR_INTERSECTCLIPRECT
EMR_INVERTRGN
EMR_LINETO
EMR_MASKBLT
EMR_MODIFYWORLDTRANSFORM
EMR_MOVETOEX
EMR_OFFSETCLIPRGN
EMR_PAINTRGN
EMR_PIE
EMR_PLGBLT
EMR_POLYBEZIER
EMR_POLYBEZIER16
EMR_POLYBEZIERTO
EMR_POLYBEZIERTO16
EMR_POLYDRAW
EMR_POLYDRAW16
EMR_POLYGON
EMR_POLYGON16
EMR_POLYLINE
EMR_SCALEVIEWPORTEXTEX
EMR_SCALEWINDOWEXTEX
EMR_SELECTCLIPPATH
EMR_SELECTOBJECT
EMR_SELECTPALETTE
EMR_SETARCDIRECTION
EMR_SETBKCOLOR
EMR_SETBKMODE
EMR_SETBRUSHORGEX
EMR_SETCOLORADJUSTMENT
EMR_SETDIBITSTODEVICE
EMR_SETMAPMODE
EMR_SETMAPPERFLAGS
EMR_SETMETARGN
EMR_SETMITERLIMIT
EMR_SETPALETTEENTRIES
EMR_SETPIXELV
EMR_SETPOLYFILLMODE
EMR_SETROP2
EMR_SETSTRETCHBLTMODE
EMR_SETTEXTALIGN
EMR_SETTEXTCOLOR
EMR_SETVIEWPORTEXTEX
EMR_SETVIEWPORTORGEX
EMR_SETWINDOWEXTEX
EMR_SETWINDOWORGEX
EMR_SETWORLDTRANSFORM
EMR_STRETCHBLT
EMR_STRETCHDIBITS
EMR_STROKEANDFILLPATH
EMR_STROKEPATH
EMR_WIDENPATH

The following record types are valid for Windows 95 and Windows NT 4.0 and later:

EMR_CREATECOLORSPACE
EMR_DELETECOLORSPACE
EMR_GLSBOUNDEDRECORD
EMR_GLSRECORD
EMR_PIXELFORMAT
EMR_SETCOLORSPACE
EMR_SETICMMODE

The following record types are valid for Windows 98 and Windows 2000 and later:

EMR_ALPHABLEND
EMR_COLORCORRECTPALETTE
EMR_COLORMATCHTOTARGETW
EMR_CREATECOLORSPACEW
EMR_GRADIENTFILL
EMR_SETICMPROFILEA
EMR_SETICMPROFILEW
EMR_SETLAYOUT
EMR_TRANSPARENTBLT

nSize

Size of the record, in bytes. This member must be a multiple of four.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRALPHABLEND

The **EMRALPHABLEND** structure contains members for the **AlphaBlend** enhanced metafile record.

```
typedef struct tagEMRALPHABLEND {
  EMR      emr;
  RECTL    rclBounds;
  LONG     xDest;
  LONG     yDest;
  LONG     cxDest;
  LONG     cyDest;
  DWORD    dwRop;
  LONG     xSrc;
  LONG     ySrc;
  XFORM    xformSrc;
```

(continued)

(continued)

```
COLORREF crBkColorSrc;
  DWORD    iUsageSrc;
  DWORD    offBmiSrc;
  DWORD    cbBmiSrc;
  DWORD    offBitsSrc;
  DWORD    cbBitsSrc;
  LONG     cxSrc;
  LONG     cySrc;
} EMRALPHABLEND, *PEMRALPHABLEND;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

xDest

Specifies the x coordinate, in logical units, of the upper-left corner of the destination rectangle.

yDest

Specifies the y coordinate, in logical units, of the upper-left corner of the destination rectangle.

cxDest

Logical width of the destination rectangle.

cyDest

Logical height of the destination rectangle.

dwRop

Stores the **BLENDFUNCTION** structure.

xSrc

Logical x coordinate of the upper-left corner of the source rectangle.

ySrc

Logical y coordinate of the upper-left corner of the source rectangle.

xformSrc

World-space to page-space transformation of the source device context.

crBkColorSrc

Background color (the RGB value) of the source device context. To make a **COLORREF** value, use the **RGB** macro.

iUsageSrc

Source bitmap information color table usage (DIB_RGB_COLORS).

offBmiSrc

Offset to the source **BITMAPINFO** structure.

cbBmiSrc

Size of the source **BITMAPINFO** structure.

offBitsSrc
Offset to the source bitmap bits.

cbBitsSrc
Size of the source bitmap bits.

cxSrc
Width of source rectangle.

cySrc
Height of the source rectangle.

Remarks

This structure is to be used during metafile playback.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **Metafiles**, **BITMAPINFO**, **AlphaBlend**, **COLORREF**, RGB

EMRANGLEARC

The **EMRANGLEARC** structure contains members for the **AngleArc** enhanced metafile record.

```
typedef struct tagEMRANGLEARC {
  EMR     emr;
  POINTL  ptlCenter;
  DWORD   nRadius;
  FLOAT   eStartAngle;
  FLOAT   eSweepAngle;
} EMRANGLEARC, *PEMRANGLEARC;
```

Members

emr
Base structure for all record types.

ptlCenter
Logical coordinates of a circle's center.

nRadius
A circle's radius, in logical units.

eStartAngle
An arc's start angle, in degrees.

eSweepAngle
An arc's sweep angle, in degrees.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **AngleArc**

EMRARC, EMRARCTO, EMRCHORD, EMRPIE

The **EMRARC, EMRARCTO, EMRCHORD,** and **EMRPIE** structures contain members for the **Arc**, **ArcTo**, **Chord**, and **Pie** enhanced metafile records.

```
typedef struct tagEMRARC {
  EMR    emr;
  RECTL  rclBox;
  POINTL ptlStart;
  POINTL ptlEnd;
} EMRARC,    *PEMRARC,
  EMRARCTO,  *PEMRARCTO,
  EMRCHORD,  *PEMRCHORD,
  EMRPIE,    *PEMRPIE;
```

Members

emr
Base structure for all record types.

rclBox
Bounding rectangle.

ptlStart
Coordinates of first radial ending point.

ptlEnd
Coordinates of second radial ending point.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRBITBLT

The **EMRBITBLT** structure contains members for the **BitBlt** enhanced metafile record. Note that graphics device interface (GDI) converts the device-dependent bitmap into a device-independent bitmap (DIB) before storing it in the metafile record.

```
typedef struct tagEMRBITBLT {
  EMR      emr;
  RECTL    rclBounds;
  LONG     xDest;
  LONG     yDest;
  LONG     cxDest;
  LONG     cyDest;
  DWORD    dwRop;
  LONG     xSrc;
  LONG     ySrc;
  XFORM    xformSrc;
  COLORREF crBkColorSrc;
  DWORD    iUsageSrc;
  DWORD    offBmiSrc;
  DWORD    offBitsSrc;
  DWORD    cbBitsSrc;
} EMRBITBLT, *PEMRBITBLT;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

xDest

Logical x-coordinate of the upper-left corner of the destination rectangle.

yDest

Logical y-coordinate of the upper-left corner of the destination rectangle.

cxDest
Logical width of the destination rectangle.

cyDest
Logical height of the destination rectangle.

dwRop
Raster-operation code. These codes define how the color data of the source rectangle is to be combined with the color data of the destination rectangle to achieve the final color.

xSrc
Logical x-coordinate of the upper-left corner of the source rectangle.

ySrc
Logical y-coordinate of the upper-left corner of the source rectangle.

xformSrc
World-space to page-space transformation of the source device context.

crBkColorSrc
Background color (the RGB value) of the source device context. To make a **COLORREF** value, use the **RGB** macro.

iUsageSrc
Value of the **bmiColors** member of the **BITMAPINFO** structure. The **iUsageSrc** member can be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

offBmiSrc
Offset to source **BITMAPINFO** structure.

cbBmiSrc
Size of source **BITMAPINFO** structure.

offBitsSrc
Offset to source bitmap bits.

cbBitsSrc
Size of source bitmap bits.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **BitBlt**, **BITMAPINFO**, **COLORREF**, RGB

EMRCOLORCORRECTPALETTE

The EMRCOLORCORRECTPALETTE structure contains members for the **ColorCorrectPalette** enhanced metafile record.

```
typedef struct tagCOLORCORRECTPALETTE {
  EMR      emr;
  DWORD    ihPalette;
  DWORD    nFirstEntry;
  DWORD    nPalEntries;
  DWORD    nReserved;
} EMRCOLORCORRECTPALETTE, *PEMRCOLORCORRECTPALETTE;
```

Members

emr

Base structure for all record types.

ihPalette

Index of the palette handle to color correct.

nFirstEntry

Index of the first entry in the palette to color correct.

nPalEntries

Number of palette entries to color correct.

nReserved

Reserved.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **ColorCorrectPalette**

EMRCOLORMATCHTOTARGET

The **EMRCOLORMATCHTOTARGET** structure contains members for the **ColorMatchToTarget** enhanced metafile record.

```
typedef struct tagCOLORMATCHTOTARGET {
  EMR   emr;
  DWORD dwAction;
  DWORD dwFlags;
  DWORD cbName;
  DWORD cbData;
  BYTE  Data[1];
} EMRCOLORMATCHTOTARGET, *PEMRCOLORMATCHTOTARGET;
```

Members

emr

Base structure for all record types.

dwAction

Action to be taken. This member can be one of the following values:

Action	Meaning
CS_ENABLE	Maps colors to the target device's color gamut. This enables color proofing. All subsequent draw commands to the DC will render colors as they would appear on the target device.
CS_DISABLE	Disables color proofing.
CS_DELETE_TRANSFORM	If color management is enabled for the target profile, disables it and deletes the concatenated transform.

dwFlags

This parameter can be the following value:

Flag	Meaning
COLORMATCHTOTARGET_EMBEDED	Indicates that a color profile has been embedded in the metafile.

cbName

Size of the desired target profile name, in bytes.

cbData

Size of the raw target profile data in bytes, if it is attached.

Data

An array containing the target profile name and the raw target profile data. The size of the array is **cbName** + **cbData**. If **cbData** is nonzero the raw target profile data is attached and follows the target profile name at location **Data**[**cbName**].

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **ColorMatchToTarget**

EMRCREATEBRUSHINDIRECT

The **EMRCREATEBRUSHINDIRECT** structure contains members for the **CreateBrushIndirect** enhanced metafile record.

```
typedef struct tagEMRCREATEBRUSHINDIRECT {
  EMR      emr;
  DWORD    ihBrush;
  LOGBRUSH lb;
} EMRCREATEBRUSHINDIRECT, *PEMRCREATEBRUSHINDIRECT;
```

Members

emr

Base structure for all record types.

ihBrush

Index of brush in handle table.

lb

LOGBRUSH structure containing information about the brush. The **lbStyle** member must be either the BS_SOLID, BS_HOLLOW, BS_NULL, or BS_HATCHED value.

Note that if your code is used on both 32-bit and 64-bit platforms, you must use the **LOGBRUSH32** structure. This maintains compatibility between the platforms when you record the metafile on one platform and use it on the other platform. If your code remains on one platform, it is sufficient to use **LOGBRUSH**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **CreateBrushIndirect**, **LOGBRUSH**, **LOGBRUSH32**

EMRCREATECOLORSPACE

The **EMRCREATECOLORSPACE** structure contains members for the **CreateColorSpace** enhanced metafile record.

```
typedef struct tagEMRCREATECOLORSPACE {
  EMR           emr;
  DWORD         ihCS;
  LOGCOLORSPACE lcs;
} EMRCREATECOLORSPACE, *PEMRCREATECOLORSPACE;
```

Members

emr
Base structure for all record types.

ihCS
Index of the color space in handle table.

lcs
Logical color space.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **CreateColorSpace,** **EMRCREATECOLORSPACEW**EMRCREATECOLORSPACEW

EMRCREATECOLORSPACEW

The **EMRCREATECOLORSPACEW** structure contains members for the **CreateColorSpace** enhanced metafile record. It differs from **EMRCREATECOLORSPACE** in that it has a Unicode logical color space and also has an optional array containing raw source profile data.

```
typedef struct tagEMRCREATECOLORSPACEW {
  EMR            emr;
  DWORD          ihCS;
  LOGCOLORSPACEW lcs;
  DWORD          dwFlags;
  DWORD          cbData;
  BYTE           Data[1];
} EMRCREATECOLORSPACEW, *PEMRCREATECOLORSPACEW;
```

Members

emr

Base structure for all record types.

ihCS

Index of the color cpace in handle table.

lcs

Logical color space. Note that this is the Unicode version of the structure.

dwFlags

Can be the following:

Flag	Meaning
CREATECOLORSPACE_EMBEDED	Indicates that a color space is embedded in the metafile.

cbData

Size of the raw source profile data in bytes, if it is attached.

Data

An array containing the source profile data. The size of the array is **cbData.**

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **CreateColorSpace** , **EMRCREATECOLORSPACE**

EMRCREATEDIBPATTERNBRUSHPT

The **EMRCREATEDIBPATTERNBRUSHPT** structure contains members for the **CreateDIBPatternBrushPt** enhanced metafile record. The **BITMAPINFO** structure is followed by the bitmap bits that form a packed device-independent bitmap (DIB).

```
typedef struct tagEMRCREATEDIBPATTERNBRUSHPT {
  EMR   emr;
  DWORD ihBrush;
  DWORD iUsage;
  DWORD offBmi;
  DWORD cbBmi;
  DWORD offBits;
  DWORD cbBits;
} EMRCREATEDIBPATTERNBRUSHPT,
PEMRCREATEDIBPATTERNBRUSHPT;
```

Members

emr

Base structure for all record types.

ihBrush

Index of brush in handle table.

iUsage

Value specifying whether the **bmiColors** member of the **BITMAPINFO** structure was provided and, if so, whether **bmiColors** contains explicit red, green, blue (RGB) values or indices. The **iUsage** member must be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

offBmi

Offset to **BITMAPINFO** structure.

cbBmi

Size of **BITMAPINFO** structure.

offBits

Offset to bitmap bits.

cbBits

Size of bitmap bits.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **CreateDIBPatternBrushPt**, RGB

EMRCREATEMONOBRUSH

The **EMRCREATEMONOBRUSH** structure contains members for the **CreatePatternBrush** (when passed a monochrome bitmap) or **CreateDIBPatternBrush** (when passed a monochrome DIB) enhanced metafile records.

```
typedef struct tagEMRCREATEMONOBRUSH {
  EMR   emr;
  DWORD ihBrush;
  DWORD iUsage;
  DWORD offBmi;
  DWORD cbBmi;
  DWORD offBits;
  DWORD cbBits;
} EMRCREATEMONOBRUSH, *PEMRCREATEMONOBRUSH;
```

Members

emr
Base structure for all record types.

ihBrush
Index of brush in handle table.

iUsage
Value specifying whether the **bmiColors** member of the **BITMAPINFO** structure was provided and, if so, whether **bmiColors** contains explicit red, green, blue (RGB) values or indices. The **iUsage** member must be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

offBmi
Offset to **BITMAPINFO** structure.

cbBmi
Size of **BITMAPINFO** structure.

offBits
Offset to bitmap bits.

cbBits
Size of bitmap bits.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **CreateDIBPatternBrush**, **CreatePatternBrush**, RGB

EMRCREATEPALETTE

The **EMRCREATEPALETTE** structure contains members for the **CreatePalette** enhanced metafile record.

```
typedef struct tagEMRCREATEPALETTE {
  EMR        emr;
  DWORD      ihPal;
  LOGPALETTE lgpl;
} EMRCREATEPALETTE, *PEMRCREATEPALETTE;
```

Members

emr
Base structure for all record types.

ihPal
Index of palette in handle table.

lgpl
LOGPALETTE structure that contains information about the palette. Note that **peFlags** members in the **PALETTEENTRY** structures do not contain any flags.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **CreatePalette**, **LOGPALETTE**, **PALETTEENTRY**

EMRCREATEPEN

The **EMRCREATEPEN** structure contains members for the **CreatePen** enhanced metafile record.

```
typedef struct tagEMRCREATEPEN {
  EMR    emr;
  DWORD  ihPen;
  LOGPEN lopn;
} EMRCREATEPEN, *PEMRCREATEPEN;
```

Members

emr
Base structure for all record types

ihPen
Index to pen in handle table

lopn
Logical pen

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **CreatePen**

EMRELLIPSE, EMRRECTANGLE

The **EMRELLIPSE** and **EMARRECTANGLE** structures contain members for the **Ellipse** and **Rectangle** enhanced metafile records.

```
typedef struct tagEMRELLIPSE {
  EMR   emr;
  RECTL rclBox;
} EMRELLIPSE,  *PEMRELLIPSE,
  EMRRECTANGLE, *PEMRRECTANGLE;
```

Members

emr
Base structure for all record types.

rclBox
Bounding rectangle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures

EMREOF

The **EMREOF** structure contains data for the enhanced metafile record that indicates the end of the metafile.

```
typedef struct tagEMREOF {
  EMR     emr;
  DWORD   nPalEntries;
  DWORD   offPalEntries;
  DWORD   nSizeLast;
} EMREOF, *PEMREOF;
```

Members

emr
Base structure for all record types.

nPalEntries
Number of palette entries.

offPalEntries
Offset to palette entries.

nSizeLast
Same size as the **nSize** member of the **EMR** structure. This member must be the last double word of the record. If palette entries exist, they precede this member.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **EMR**

EMREXCLUDECLIPRECT, EMRINTERSECTCLIPRECT

The **EMREXCLUDECLIPRECT** and **EMRINTERSECTCLIPRECT**structures contain members for the **ExcludeClipRect** and **IntersectClipRect** enhanced metafile records.

```
typedef struct tagEMREXCLUDECLIPRECT {
  EMR   emr;
  RECTL rclClip;
} EMREXCLUDECLIPRECT,   *PEMREXCLUDECLIPRECT,
  EMRINTERSECTCLIPRECT, *PEMRINTERSECTCLIPRECT;
```

Members

emr
Base structure for all record types.

rclClip
Clipping rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMREXTCREATEFONTINDIRECTW

The **EMREXTCREATEFONTINDIRECTW** structure contains members for the **CreateFontIndirect** enhanced metafile record.

```
typedef struct tagEMREXTCREATEFONTINDIRECTW {
  EMR          emr;
  DWORD        ihFont;
  EXTLOGFONTW elfw;
} EMREXTCREATEFONTINDIRECTW,
PEMREXTCREATEFONTINDIRECTW;
```

Members

emr
Base structure for all record types.

ihFont
Index to the font in handle table.

elfw
Logical font.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **CreateFontIndirect**

EMREXTCREATEPEN

The **EMREXTCREATEPEN** structure contains members for the **ExtCreatePen** enhanced metafile record. If the record contains a **BITMAPINFO** structure, it is followed by the bitmap bits that form a packed device-independent bitmap (DIB).

```
typedef struct tagEMREXTCREATEPEN {
  EMR       emr;
  DWORD     ihPen;
  DWORD     offBmi;
  DWORD     cbBmi;
  DWORD     offBits;
  DWORD     cbBits;
  EXTLOGPEN elp;
} EMREXTCREATEPEN, *PEMREXTCREATEPEN;
```

Members

emr
Base structure for all record types.

ihPen
Index to pen in handle table.

offBmi
Offset to **BITMAPINFO** structure, if any.

cbBmi
Size of **BITMAPINFO** structure, if any.

offBits
Offset to brush bitmap bits, if any.

cbBits
Size of brush bitmap bits, if any.

elp
Extended logical pen, including the **elpStyleEntry** member of the **EXTLOGPEN** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **ExtCreatePen**, **EXTLOGPEN**

EMREXTFLOODFILL

The **EMREXTFLOODFILL** structure contains members for the **ExtFloodFill** enhanced metafile record.

```
typedef struct tagEMREXTFLOODFILL {
  EMR      emr;
  POINTL   ptlStart;
  COLORREF crColor;
  DWORD    iMode;
} EMREXTFLOODFILL, *PEMREXTFLOODFILL;
```

Members

emr
Base structure for all record types.

ptlStart
Coordinates where filling begins.

crColor
Color of fill. To make a **COLORREF** value, use the **RGB** macro.

iMode
Type of fill operation to be performed. This member must be either the FLOODFILLBORDER or FLOODFILLSURFACE value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **ExtFloodFill**, **COLORREF**, RGB

EMREXTSELECTCLIPRGN

The **EMREXTSELECTCLIPRGN** structure contains members for the **ExtSelectClipRgn** enhanced metafile record.

```
typedef struct tagEMREXTSELECTCLIPRGN {
  EMR   emr;
  DWORD cbRgnData;
  DWORD iMode;
  BYTE  RgnData[1];
} EMREXTSELECTCLIPRGN, *PEMREXTSELECTCLIPRGN;
```

Members

emr
Base structure for all record types.

cbRgnData
Size of region data, in bytes.

iMode
Operation to be performed. This member must be one of the following values: RGN_AND, RGN_COPY, RGN_DIFF, RGN_OR, or RGN_XOR.

RgnData
Buffer containing **RGNDATA** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **ExtSelectClipRgn**

EMREXTTEXTOUTA, EMREXTTEXTOUTW

The **EMREXTTEXTOUTA** and **EMREXTTEXTOUTW** structures contain members for the **ExtTextOut**, **TextOut**, or **DrawText** enhanced metafile records.

```
typedef struct tagEMREXTTEXTOUTA {
  EMR     emr;
  RECTL   rclBounds;
  DWORD   iGraphicsMode;
  FLOAT   exScale;
  FLOAT   eyScale;
  EMRTEXT emrtext;
} EMREXTTEXTOUTA, *PEMREXTTEXTOUTA,
  EMREXTTEXTOUTW, *PEMREXTTEXTOUTW;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

iGraphicsMode
Current graphics mode. This member can be either the GM_COMPATIBLE or GM_ADVANCED value.

exScale
X-scaling factor from page units to .01mm units if the graphics mode is the GM_COMPATIBLE value.

eyScale
Y-scaling factor from page units to .01mm units if the graphics mode is the GM_COMPATIBLE value.

emrtext
EMRTEXT structure, which is followed by the string and the intercharacter spacing array.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRFILLPATH, EMRSTROKEANDFILLPATH, EMRSTROKEPATH

The **EMRFILLPATH, EMRSTROKEANDFILLPATH,** and **EMRSTROKEPATH** structures contain members for the **FillPath**, **StrokeAndFillPath**, and **StrokePath** enhanced metafile records.

```
typedef struct tagEMRFILLPATH {
  EMR   emr;
  RECTL rclBounds;
} EMRFILLPATH,          *PEMRFILLPATH,
  EMRSTROKEANDFILLPATH, *PEMRSTROKEANDFILLPATH,
  EMRSTROKEPATH,        *PEMRSTROKEPATH;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures

EMRFILLRGN

The **EMRFILLRGN** structure contains members for the **FillRgn** enhanced metafile record.

```
typedef struct tagEMRFILLRGN {
  EMR   emr;
  RECTL rclBounds;
  DWORD cbRgnData;
  DWORD ihBrush;
  BYTE  RgnData[1];
} EMRFILLRGN, *PEMRFILLRGN;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

cbRgnData
Size of region data, in bytes.

ihBrush
Index of brush, in handle table.

RgnData
Buffer containing **RGNDATA** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **FillRgn**, **RGNDATA**

EMRFORMAT

The **EMRFORMAT** structure contains information that identifies graphics data in an enhanced metafile. A GDICOMMENT_MULTIFORMATS enhanced metafile public comment contains an array of **EMRFORMAT** structures.

```
typedef struct tagEMRFORMAT {
  DWORD   dSignature;
  DWORD   nVersion;
  DWORD   cbData;
  DWORD   offData;
} EMRFORMAT;
```

Members

dSignature
Contains a picture format identifier. The following identifier values are defined:

Identifier	Meaning
ENHMETA_SIGNATURE	The picture is in enhanced metafile format.
EPS_SIGNATURE	The picture is in encapsulated PostScript file format.

nVersion

Contains a picture version number. The following version number value is defined:

Version	Meaning
1	This is the version number of a level 1 encapsulated PostScript file.

cbData

Specifies the size, in bytes, of the picture data.

offData

Specifies an offset to the picture data. The offset is figured from the start of the GDICOMMENT_MULTIFORMATS public comment within which this **EMRFORMAT** structure is embedded. The offset must be a **DWORD** offset.

Remarks

The reference page for **GdiComment** discusses enhanced metafile public comments in general, and the GDICOMMENT_MULTIFORMATS public comment in particular.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **GdiComment**

EMRFRAMERGN

The **EMRFRAMERGN** structure contains members for the **FrameRgn** enhanced metafile record.

```
typedef struct tagEMRFRAMERGN {
  EMR    emr;
  RECTL  rclBounds;
  DWORD  cbRgnData;
  DWORD  ihBrush;
  SIZEL  szlStroke;
  BYTE   RgnData[1];
} EMRFRAMERGN, *PEMRFRAMERGN;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

cbRgnData
Size of region data, in bytes.

ihBrush
Index of brush, in handle table.

szlStroke
Width and height of region frame.

RgnData
Buffer containing **RGNDATA** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **FrameRgn**, **RGNDATA**

EMRGDICOMMENT

The **EMRGDICOMMENT** structure contains application-specific data. This enhanced metafile record is only meaningful to applications that know the format of the data and how to utilize it. This record is ignored by graphics device interface (GDI) during playback of the enhanced metafile.

```
typedef struct tagEMRGDICOMMENT {
  EMR   emr;
  DWORD cbData;
  BYTE  Data[1];
} EMRGDICOMMENT, *PEMRGDICOMMENT;
```

Members

emr
Base structure for all record types.

cbData
Size of data buffer, in bytes.

Data[1]
Application-specific data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRGLSBOUNDEDRECORD

The **EMRGLSBOUNDEDRECORD** structure contains members for an enhanced metafile record generated by OpenGL functions. It contains data for OpenGL functions with information in pixel units that must be scaled when playing the metafile.

```
typedef struct tagEMRGLSBOUNDEDRECORD {
  EMR    emr;          // base structure
  RECTL  rclBounds;    // bounds in recording coordinates
  DWORD  cbData;       // size of Data[], in bytes
  BYTE   Data[1];
} EMRGLSBOUNDEDRECORD, *PEMRGLSBOUNDEDRECORD;
```

Members

emr
Base structure for all record types

rclBounds
Bounds of the rectangle, in recording coordinates, within which to perform the OpenGL function.

cbData
Size of **Data**, in bytes.

Data
Array of data representing the OpenGL function to be performed.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, OpenGL on Windows NT, Windows 2000, and Windows 95/98

EMRGLSRECORD

The **EMRGLSRECORD** structure contains members for an enhanced metafile record generated by OpenGL functions, It contains data for OpenGL functions that scale automatically to the OpenGL viewport.

```
typedef struct tagEMRGLSRECORD {
  EMR    emr;          // base structure
  DWORD  cbData;       // size of Data[], in bytes
  BYTE   Data[1];
} EMRGLSRECORD, *PEMRGLSRECORD;
```

Members

emr

Base structure for all records.

cbData

Size of **Data**, in bytes.

Data

Array of data representing the OpenGL function to be performed.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, OpenGL on Windows NT, Windows 2000, and Windows 95/98

EMRGRADIENTFILL

The **EMRGRADIENTFILL** structure contains members for the **GradientFill** enhanced metafile record.

```
typedef struct tagEMRGRADIENTFILL {
  EMR       emr;
  RECTL     rclBounds;
  DWORD     nVer;
  DWORD     nTri;
  ULONG     ulMode;
  TRIVERTEX Ver[1];
}EMRGRADIENTFILL,*PEMRGRADIENTFILL;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

nVer

Number of vertices.

nTri

Number of rectangles or triangles to pass to **GradientFill**.

ulMode

Specifies the gradient fill mode.

Ver[1]

Pointer to an array of **TRIVERTEX** structures that each define a triangle vertex.

Remarks

This is a variable-length structure. The **nVer** element designates the beginning of the variable-length area. First comes an array of **nVer TRIVERTEX** structures to pass the vertices. Next comes an array of either **nTri GRADIENT_TRIANGLE** structures or **nTri GRADIENT_RECT** structures, depending on the value of **jlMode** (triangles or rectangles).

This structure is to be used during metafile playback.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **Metafiles**, **BITMAPINFO**, **GradientFill**, **GRADIENT_TRIANGLE**, **GRADIENT_RECT**

EMRINVERTRGN, EMRPAINTRGN

The **EMRINVERTRGN** and **EMRPAINTRGN** structures contain members for the **InvertRgn** and **PaintRgn** enhanced metafile records.

```
typedef struct tagEMRINVERTRGN {
  EMR    emr;
  RECTL  rclBounds;
  DWORD  cbRgnData;
  BYTE   RgnData[1];
} EMRINVERTRGN, *PEMRINVERTRGN,
  EMRPAINTRGN,  *PEMRPAINTRGN;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

cbRgnData

Size of region data, in bytes.

RgnData

Buffer containing an **RGNDATA** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **InvertRgn**, **PaintRgn**

EMRLINETO, EMRMOVETOEX

The **EMRLINETO** and **EMRMOVETOEX** structures contains members for the **LineTo** and **MoveToEx** enhanced metafile records.

```
typedef struct tagEMRLINETO {
  EMR     emr;
  POINTL  ptl;
} EMRLINETO,    *PEMRLINETO,
  EMRMOVETOEX,  *PEMRMOVETOEX;
```

Members

emr

Base structure for all record types.

ptl

Coordinates of the line's ending point for the **LineTo** function or coordinates of the new current position for the **MoveToEx** function.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRMASKBLT

The **EMRMASKBLT** structure contains members for the **MaskBlt** enhanced metafile record. Note that graphics device interface (GDI) converts the device-dependent bitmap into a device-independent bitmap (DIB) before storing it in the metafile record.

```
typedef struct tagEMRMASKBLT {
  EMR     emr;
  RECTL   rclBounds;
  LONG    xDest;
  LONG    yDest;
  LONG    cxDest;
  LONG    cyDest;
  DWORD   dwRop;
  LONG    xSrc;
  LONG    ySrc;
XFORM    xformSrc;
  COLORREF crBkColorSrc;
  DWORD    iUsageSrc;
  DWORD    offBmiSrc;
  DWORD    cbBmiSrc;
  DWORD    offBitsSrc;
  DWORD    cbBitsSrc;
  LONG     xMask;
  LONG     yMask;
  DWORD    iUsageMask;
  DWORD    offBmiMask;
  DWORD    cbBmiMask;
  DWORD    offBitsMask;
  DWORD    cbBitsMask;
} EMRMASKBLT, *PEMRMASKBLT;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

xDest
Logical x-coordinate of the upper-left corner of the destination rectangle.

yDest
Logical y-coordinate of the upper-left corner of the destination rectangle.

cxDest
Logical width of the destination rectangle

cyDest
Logical height of the destination rectangle

dwRop
Raster-operation code. These codes define how the color data of the source rectangle is to be combined with the color data of the destination rectangle to achieve the final color.

xSrc
Logical x-coordinate of the upper-left corner of the source rectangle.

ySrc
Logical y-coordinate of the upper-left corner of the source rectangle.

xformSrc
World-space to page-space transformation of the source device context.

crBkColorSrc
Background color (the RGB value) of the source device context. To make a **COLORREF** value, use the **RGB** macro.

iUsageSrc
Value of the **bmiColors** member of the source **BITMAPINFO** structure. The **iUsageSrc** member can be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

offBmiSrc
Offset to source **BITMAPINFO** structure.

cbBmiSrc
Size of source **BITMAPINFO** structure.

offBitsSrc
Offset to source bitmap bits.

cbBitsSrc
Size of source bitmap bits.

xMask
Horizontal pixel offset into mask bitmap

yMask
Vertical pixel offset into mask bitmap

iUsageMask
Value of the **bmiColors** member of the mask **BITMAPINFO** structure.

offBmiMask
Offset to mask **BITMAPINFO** structure.

cbBmiMask
Size of mask **BITMAPINFO** structure.

offBitsMask
Offset to mask bitmap bits.

cbBitsMask
Size of mask bitmap bits.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **MaskBlt**, **COLORREF**, RGB

EMRMODIFYWORLDTRANSFORM

The **EMRMODIFYWORLDTRANSFORM** structure contains members for the **ModifyWorldTransform** enhanced metafile record.

```
typedef struct tagEMRMODIFYWORLDTRANSFORM {
  EMR   emr;
  XFORM xform;
  DWORD iMode;
transformation
} EMRMODIFYWORLDTRANSFORM,
*PEMRMODIFYWORLDTRANSFORM;
```

Members

emr

Base structure for all record types.

xform

World-space to page-space transformation data.

iMode

Value specifying how the transformation data modifies the current world transformation. This member can be one of the following values:

MWT_IDENTITY, MWT_LEFTMULTIPLY, or MWT_RIGHTMULTIPLY.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **ModifyWorldTransform**

EMROFFSETCLIPRGN

The **EMROFFSETCLIPRGN** structure contains members for the **OffsetClipRgn** enhanced metafile record.

```
typedef struct tagEMROFFSETCLIPRGN {
  EMR    emr;
  POINTL ptlOffset;
} EMROFFSETCLIPRGN, *PEMROFFSETCLIPRGN;
```

Members

emr

Base structure for all record types.

ptlOffset

Logical coordinates of offset.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **OffsetClipRgn**

EMRPIXELFORMAT

The **EMRPIXELFORMAT** structure contains the members for the **SetPixelFormat** enhanced metafile record. The pixel format information in **ENHMETAHEADER** refers to this structure.

```
typedef struct tagEMRPIXELFORMAT {
  EMR                   emr;
  PIXELFORMATDESCRIPTOR pfd;
} EMRPIXELFORMAT, *PEMRPIXELFORMAT;
```

Members

emr

Base structure for all record types.

pfd

PIXELFORMATDESCRIPTOR structure, which describes the pixel format.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SetPixelFormat**, **ENHMETAHEADER**, **PIXELFORMATDESCRIPTOR**

EMRPLGBLT

The **EMRPLGBLT** structure contains members for the **PlgBlt** enhanced metafile record. Note that graphical device interface (GDI) converts the device-dependent bitmap into a device-independent bitmap (DIB) before storing it in the metafile record.

```
typedef struct tagEMRPLGBLT {
  EMR       emr;
  RECTL     rclBounds;
  POINTL    aptlDest[3];
  LONG      xSrc;
  LONG      ySrc;
  LONG      cxSrc;
  LONG      cySrc;
  XFORM     xformSrc;
  COLORREF  crBkColorSrc;
  DWORD     iUsageSrc;
  DWORD     offBmiSrc;
  DWORD     cbBmiSrc;
  DWORD     offBitsSrc;
  DWORD     cbBitsSrc;
  LONG      xMask;
  LONG      yMask;
  DWORD     iUsageMask;
  DWORD     offBmiMask;
  DWORD     cbBmiMask;
  DWORD     offBitsMask;
  DWORD     cbBitsMask;
} EMRPLGBLT, *PEMRPLGBLT;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

aptlDest

Array of three points in logical space that identify three corners of the destination parallelogram. The upper-left corner of the source rectangle is mapped to the first point in this array, the upper-right corner to the second point in this array, and the lower-left corner to the third point. The lower-right corner of the source rectangle is mapped to the implicit fourth point in the parallelogram.

xSrc

Logical x-coordinate of the upper-left corner of the source rectangle.

ySrc
Logical y-coordinate of the upper-left corner of the source rectangle.

cxSrc
Logical width of the source.

cySrc
Logical height of the source.

xformSrc
World-space to page-space transformation of the source device context.

crBkColorSrc
Background color (the RGB value) of the source device context. To make a **COLORREF** value, use the **RGB** macro.

iUsageSrc
Value of the **bmiColors** member of the **BITMAPINFO** structure. The **iUsageSrc** member can be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

offBmiSrc
Offset to source **BITMAPINFO** structure.

cbBmiSrc
Size of source **BITMAPINFO** structure.

offBitsSrc
Offset to source bitmap bits.

cbBitsSrc
Size of source bitmap bits.

xMask
Horizontal pixel offset into mask bitmap.

yMask
Vertical pixel offset into mask bitmap.

iUsageMask
Value of the **bmiColors** member of the mask **BITMAPINFO** structure.

offBmiMask
Offset to mask **BITMAPINFO** structure.

cbBmiMask
Size of mask **BITMAPINFO** structure.

offBitsMask
Offset to mask bitmap bits.

cbBitsMask
Size of mask bitmap bits.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **PlgBlt**, **COLORREF**, RGB

EMRPOLYDRAW

The **EMRPOLYDRAW** structure contains members for the **PolyDraw** enhanced metafile record.

```
typedef struct tagEMRPOLYDRAW {
  EMR     emr;
  RECTL   rclBounds;
  DWORD   cptl;
  POINTL  aptl[1];
  BYTE    abTypes[1];
} EMRPOLYDRAW, *PEMRPOLYDRAW;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

cptl
Number of points.

aptl
Array of 32-bit points.

abTypes
Array of values that specifies how each point in the **aptl** array is used. This member can be one of the following values: PT_MOVETO, PT_LINETO, or PT_BEZIERTO. The PT_LINETO or PT_BEZIERTO value can be combined with the PT_CLOSEFIGURE value by using the bitwise-xOR operator.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **PolyDraw**

EMRPOLYDRAW16

The **EMRPOLYDRAW16** structure contains members for the **PolyDraw** enhanced metafile record.

```
typedef struct tagEMRPOLYDRAW16 {
  EMR     emr;
  RECTL   rclBounds;
  DWORD   cpts;
  POINTS  apts[1];
  BYTE    abTypes[1];
} EMRPOLYDRAW16, *PEMRPOLYDRAW16;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

cpts
Number of points.

apts
Array of 16-bit points.

abTypes
Array of values that specifies how each point in the **apts** array is used. This member can be one of the following values: PT_MOVETO, PT_LINETO, or PT_BEZIERTO. The PT_LINETO or PT_BEZIERTO value can be combined with the PT_CLOSEFIGURE value by using the bitwise-OR operator.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **PolyDraw**

EMRPOLYLINE, EMRPOLYBEZIER, EMRPOLYGON, EMRPOLYBEZIERTO, EMRPOLYLINETO

The **EMRPOLYLINE, EMRPOLYBEZIER, EMRPOLYGON, EMRPOLYBEZIERTO,** and **EMRPOLYLINETO** structures contain members for the **Polyline**, **PolyBezier**, **Polygon**, **PolyBezierTo**, and **PolylineTo** enhanced metafile records.

```
typedef struct tagEMRPOLYLINE {
  EMR    emr;
  RECTL  rclBounds;
  DWORD  cptl;
  POINTL aptl[1];
} EMRPOLYLINE,      *PEMRPOLYLINE,
  EMRPOLYBEZIER,    *PEMRPOLYBEZIER,
  EMRPOLYGON,       *PEMRPOLYGON,
  EMRPOLYBEZIERTO,  *PEMRPOLYBEZIERTO,
  EMRPOLYLINETO,    *PEMRPOLYLINETO;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

cptl

Number of points array.

aptl

Array of 32-bit points.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRPOLYLINE16, EMRPOLYBEZIER16, EMRPOLYGON16, EMRPOLYBEZIERTO16, EMRPOLYLINETO16

The **EMRPOLYLINE16, EMRPOLYBEZIER16, EMRPOLYGON16, EMRPOLYBEZIERTO16,** and **EMRPOLYLINETO16** structures contain members for the **Polyline**, **PolyBezier**, **Polygon**, **PolyBezierTo**, and **PolylineTo** enhanced metafile records.

```
typedef struct tagEMRPOLYLINE {
  EMR    emr;
  RECTL  rclBounds;
  DWORD  cpts;
  POINTL apts[1];
} EMRPOLYLINE16,     *PEMRPOLYLINE16,
  EMRPOLYBEZIER16,   *PEMRPOLYBEZIER16,
  EMRPOLYGON16,      *PEMRPOLYGON16,
  EMRPOLYBEZIERTO16, *PEMRPOLYBEZIERTO16,
  EMRPOLYLINETO16,   *PEMRPOLYLINETO16;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

cpts

Number of points in the array.

apts

Array of 16-bit points.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRPOLYPOLYLINE, EMRPOLYPOLYGON

The **EMRPOLYPOLYLINE** and **EMRPOLYPOLYGON** structures contain members for the **PolyPolyline** and **PolyPolygon** enhanced metafile records.

```
typedef struct tagEMRPOLYPOLYLINE {
  EMR     emr;
  RECTL   rclBounds;
  DWORD   nPolys;
  DWORD   cptl;
  DWORD   aPolyCounts[1];
  POINTL  aptl[1];
} EMRPOLYPOLYLINE, *PEMRPOLYPOLYLINE,
  EMRPOLYPOLYGON,  *PEMRPOLYPOLYGON;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

nPolys
Number of polys.

cptl
Total number of points in all polys.

aPolyCounts
Array of point counts for each poly.

aptl
Array of 32-bit points.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures

EMRPOLYPOLYLINE16, EMRPOLYPOLYGON16

The **EMRPOLYPOLYLINE16** and **EMRPOLYPOLYGON16** structures contain members for the **PolyPolyline** and **PolyPolygon** enhanced metafile records.

```
typedef struct tagEMRPOLYPOLYLINE16 {
  EMR     emr;
  RECTL   rclBounds;
  DWORD   nPolys;
  DWORD   cpts;
  DWORD   aPolyCounts[1];
  POINTS  apts[1];
} EMRPOLYPOLYLINE16, *PEMRPOLYPOLYLINE16,
  EMRPOLYPOLYGON16,  *PEMRPOLYPOLYGON16;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

nPolys
Number of polys.

cpts
Total number of points in all polys.

aPolyCounts
Array of point counts for each poly.

apts
Array of 16-bit points.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRPOLYTEXTOUTA, EMRPOLYTEXTOUTW

The **EMRPOLYTEXTOUTA** and **EMRPOLYTEXTOUTW** structures contain members for the **PolyTextOut** enhanced metafile record.

```
typedef struct tagEMRPOLYTEXTOUTA {
  EMR     emr;
  RECTL   rclBounds;
  DWORD   iGraphicsMode;
  FLOAT   exScale;
  FLOAT   eyScale;
  LONG    cStrings;
  EMRTEXT aemrtext[1];
} EMRPOLYTEXTOUTA, *PEMRPOLYTEXTOUTA,
  EMRPOLYTEXTOUTW, *PEMRPOLYTEXTOUTW;
```

Members

emr

Base structure for all record types.

rclBounds

Bounding rectangle, in device units.

iGraphicsMode

Current graphics mode. This member can be either the GM_COMPATIBLE or GM_ADVANCED value.

exScale

X-scaling factor from page units to .01mm units if the graphics mode is the GM_COMPATIBLE value.

eyScale

Y-scaling factor from page units to .01mm units if the graphics mode is the GM_COMPATIBLE value.

cStrings

Number of strings.

aemrtext

EMRTEXT structure, which is followed by the string and the intercharacter spacing array.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRRESIZEPALETTE

The **EMRRESIZEPALETTE** structure contains members for the **ResizePalette** enhanced metafile record.

```
typedef struct tagEMRRESIZEPALETTE {
  EMR   emr;
  DWORD ihPal;
  DWORD cEntries;
} EMRRESIZEPALETTE, *PEMRRESIZEPALETTE;
```

Members

emr

Base structure for all record types.

ihPal

Index of the palette in the handle table.

cEntries

Number of entries in palette after resizing.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **ResizePalette**

EMRRESTOREDC

The **EMRRESTOREDC** structure contains members for the **RestoreDC** enhanced metafile record.

```
typedef struct tagEMRRESTOREDC {
  EMR  emr;
  LONG iRelative;
} EMRRESTOREDC, *PEMRRESTOREDC;
```

Members

emr

Base structure for all record types.

iRelative

Relative instance to restore.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **RestoreDC**

EMRROUNDRECT

The **EMRROUNDRECT** structure contains members for the **RoundRect** enhanced metafile record.

```
typedef struct tagEMRROUNDRECT {
  EMR   emr;
  RECTL rclBox;
  SIZEL szlCorner;
} EMRROUNDRECT, *PEMRROUNDRECT;
```

Members

emr
Base structure for all record types.

rclBox
Bounding rectangle.

szlCorner
Width and height of the ellipse used to draw rounded corners.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **RoundRect**

EMRSCALEVIEWPORTEXTEX, EMRSCALEWINDOWEXTEX

The **EMRSCALEVIEWPORTEXTEX** and **EMRSCALEWINDOWEXTEX** structures contain members for the **ScaleViewportExtEx** and **ScaleWindowExtEx** enhanced metafile records.

```
typedef struct tagEMRSCALEVIEWPORTEXTEX {
  EMR  emr;
  LONG xNum;
  LONG xDenom;
  LONG yNum;
  LONG yDenom;
} EMRSCALEVIEWPORTEXTEX, *PEMRSCALEVIEWPORTEXTEX,
  EMRSCALEWINDOWEXTEX,   *PEMRSCALEWINDOWEXTEX;
```

Members

emr

Base structure for all record types.

xNum

Horizontal multiplicand.

xDenom

Horizontal divisor.

yNum

Vertical multiplicand.

yDenom

Vertical divisor.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRSETCOLORSPACE, EMRSELECTCOLORSPACE, EMRDELETECOLORSPACE

The **EMRSETCOLORSPACE**, **EMRSELECTCOLORSPACE**, and **EMRDELETECOLORSPACE** structures contain members for the **SetColorSpace** and **DeleteColorSpace** enhanced metafile records.

```
typedef struct tagEMRSETCOLORSPACE {
  EMR     emr;
  DWORD   ihCS;
} EMRSETCOLORSPACE, *PEMRSETCOLORSPACE,
  EMRSELECTCOLORSPACE, *PEMRSELECTCOLORSPACE,
  EMRDELETECOLORSPACE, *PEMRDELETECOLORSPACE;
```

Members

emr

Base structure for all record types.

ihCS

Color space handle index.

Remarks

There is no function that generates an enhanced metafile record with the **EMRSELECTCOLORSPACE** structure.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRSELECTOBJECT, EMRDELETEOBJECT

The **EMRSELECTOBJECT** and **EMRDELETEOBJECT** structures contain members for the **SelectObject** and **DeleteObject** enhanced metafile records.

```
typedef struct tagEMRSELECTOBJECT {
  EMR   emr;
  DWORD ihObject;
} EMRSELECTOBJECT, *PEMRSELECTOBJECT,
  EMRDELETEOBJECT, *PEMRDELETEOBJECT;
```

Members

emr

Base structure for all record types.

ihObject

Index of an object in the handle table.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRSELECTPALETTE

The **EMRSELECTPALETTE** structure contains members for the **SelectPalette** enhanced metafile record. Note that the *bForceBackground* parameter in **SelectPalette** is always recorded as TRUE, which causes the palette to be realized as a background palette.

```
typedef struct tagEMRSELECTPALETTE {
  EMR   emr;
  DWORD ihPal;
} EMRSELECTPALETTE, *PEMRSELECTPALETTE;
```

Members

emr

Base structure for all record types.

ihPal

Index to logical palette in the handle table.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SelectPalette**

EMRSETARCDIRECTION

The **EMRSETARCDIRECTION** structure contains members for the **SetArcDirection** enhanced metafile record.

```
typedef struct tagEMRSETARCDIRECTION {
  EMR   emr;
  DWORD iArcDirection;
} EMRSETARCDIRECTION, *PEMRSETARCDIRECTION;
```

Members

emr

Base structure for all record types.

iArcDirection

Arc direction. This member can be either the AD_CLOCKWISE or AD_COUNTERCLOCKWISE value.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, Arc, **SetArcDirection**

EMRSETBKCOLOR, EMRSETTEXTCOLOR

The **EMRSETBKCOLOR** and **EMRSETTEXTCOLOR** structures contain members for the **SetBkColor** and **SetTextColor** enhanced metafile records.

```
typedef struct tagEMRSETTEXTCOLOR {
  EMR      emr;
  COLORREF crColor;
} EMRSETBKCOLOR,   *PEMRSETBKCOLOR,
  EMRSETTEXTCOLOR, *PEMRSETTEXTCOLOR;
```

Members

emr

Base structure for all record types.

crColor

Color value.To make a **COLORREF** value, use the **RGB** macro.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **COLORREF**, RGB

EMRSETCOLORADJUSTMENT

The **EMRSETCOLORADJUSTMENT** structure contains members for the **SetColorAdjustment** enhanced metafile record.

```
typedef struct tagEMRSETCOLORADJUSTMENT {
  EMR  emr;
  COLORADJUSTMENT ColorAdjustment;
} EMRSETCOLORADJUSTMENT, *PEMRSETCOLORADJUSTMENT;
```

Members

emr
Base structure for all record types.

ColorAdjustment
COLORADJUSTMENT structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SetColorAdjustment**

EMRSETDIBITSTODEVICE

The **EMRSETDIBITSTODEVICE** structure contains members for the **SetDIBitsToDevice** enhanced metafile record.

```
typedef struct tagEMRSETDIBITSTODEVICE {
  EMR   emr;
  RECTL rclBounds;
  LONG  xDest;
  LONG  yDest;
  LONG  xSrc;
  LONG  ySrc;
  LONG  cxSrc;
  LONG  cySrc;
  DWORD offBmiSrc;
  DWORD cbBmiSrc;
  DWORD offBitsSrc;
  DWORD cbBitsSrc;
  DWORD iUsageSrc;
  DWORD iStartScan;
  DWORD cScans;
} EMRSETDIBITSTODEVICE, *PEMRSETDIBITSTODEVICE;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

xDest
Logical x-coordinate of the upper-left corner of the destination rectangle.

yDest
Logical y-coordinate of the upper-left corner of the destination rectangle.

xSrc
Logical x-coordinate of the lower-left corner of the source device-independent bitmap (DIB).

ySrc
Logical y-coordinate of the lower-left corner of the source DIB.

cxSrc
Width of the source rectangle.

cySrc
Height of the source rectangle.

offBmiSrc
Offset to the source **BITMAPINFO** structure.

cbBmiSrc
Size of the source **BITMAPINFO** structure.

offBitsSrc
Offset to source bitmap bits.

cbBitsSrc
Size of source bitmap bits.

iUsageSrc
Value of the **bmiColors** member of the **BITMAPINFO** structure. The **iUsageSrc** member can be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

iStartScan
First scan line in the array.

cScans
Number of scan lines.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **SetDIBitsToDevice**

EMRSETICMPROFILE

The **EMRSETICMPROFILE** structure contains members for the **SetICMProfile** enhanced metafile record..

```
typedef struct tagEMRSETICMPROFILE {
  EMR     emr;
  DWORD   dwFlags;
  DWORD   cbName;
  DWORD   cbData;
  BYTE    Data[1];
} EMRSETICMPROFILE, *PEMRSETICMPROFILE,
  EMRSETICMPROFILEA, *PEMRSETICMPROFILEA,
  EMRSETICMPROFILEW, *PEMRSETICMPROFILEW;
```

Members

emr
Base structure for all record types.

dwFlags
Profile Flags.

cbName
Size of the desired profile name.

cbData
Size of profile data, if attached.

Data[1]
Array size is **cbName** and **cbData**.

Remarks

This structure is to be used during metafile playback.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **Metafiles**, **SetICMProfile**

EMRSETMAPPERFLAGS

The **EMRSETMAPPERFLAGS** structure contains members for the **SetMapperFlags** enhanced metafile record.

```
typedef struct tagEMRSETMAPPERFLAGS {
  EMR   emr;
  DWORD dwFlags;
} EMRSETMAPPERFLAGS, *PEMRSETMAPPERFLAGS;
```

Members

emr
Base structure for all record types.

dwFlags
Font mapper flag.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SetMapperFlags**

EMRSETMITERLIMIT

The **EMRSETMITERLIMIT** structure contains members for the **SetMiterLimit** enhanced metafile record.

```
typedef struct tagEMRSETMITERLIMIT {
  EMR   emr;
  FLOAT eMiterLimit;
} EMRSETMITERLIMIT, *PEMRSETMITERLIMIT;
```

Members

emr

Base structure for all record types.

eMiterLimit

New miter limit.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SetMiterLimit**

EMRSETPALETTEENTRIES

The **EMRSETPALETTEENTRIES** structure contains members for the **SetPaletteEntries** enhanced metafile record.

```
typedef struct tagEMRSETPALETTEENTRIES {
  EMR          emr;
  DWORD        ihPal;
  DWORD        iStart;
  DWORD        cEntries;
  PALETTEENTRY aPalEntries[1];
} EMRSETPALETTEENTRIES, *PEMRSETPALETTEENTRIES;
```

Members

emr

Base structure for all record types.

ihPal

Palette handle index.

iStart
Index of first entry to set.

cEntries
Number of entries.

aPalEntries
Array of **PALETTEENTRY** structures. Note that **peFlags** members in the structures do not contain any flags.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **PALETTEENTRY**, **SetPaletteEntries**

EMRSETPIXELV

The **EMRSETPIXELV** structure contains members for the **SetPixelV** enhanced metafile record. When an enhanced metafile is created, calls to **SetPixel** are also recorded in this record.

```
typedef struct tagEMRSETPIXELV {
  EMR      emr;
  POINTL   ptlPixel;
  COLORREF crColor;
} EMRSETPIXELV, *PEMRSETPIXELV;
```

Members

emr
Base structure for all record types.

ptlPixel
Logical coordinates of pixel.

crColor
Color value.To make a **COLORREF** value, use the **RGB** macro.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SetPixelV**, **SetPixel** , **COLORREF**, RGB

EMRSETVIEWPORTEXTEX, EMRSETWINDOWEXTEX

The **EMRSETVIEWPORTEXTEX** and **EMRSETWINDOWEXTEX** structures contains members for the **SetViewportExtEx** and **SetWindowExtEx** enhanced metafile records.

```
typedef struct tagEMRSETVIEWPORTEXTEX {
  EMR   emr;
  SIZEL szlExtent;
} EMRSETVIEWPORTEXTEX, *PEMRSETVIEWPORTEXTEX,
  EMRSETWINDOWEXTEX,   *PEMRSETWINDOWEXTEX;
```

Members

emr

Base structure for all record types.

szlExtent

Horizontal and vertical extents. For **SetViewportExtEx**, the extents are in device units, and for **SetWindowExtEx**, the extents are in logical units.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRSETVIEWPORTORGEX, EMRSETWINDOWORGEX, EMRSETBRUSHORGEX

The **EMRSETVIEWPORTORGEX, EMRSETWINDOWORGEX,** and **EMRSETBRUSHORGEX** structures contain members for the **SetViewportOrgEx**, **SetWindowOrgEx**, and **SetBrushOrgEx** enhanced metafile records.

```
typedef struct tagEMRSETVIEWPORTORGEX {
  EMR     emr;
  POINTL ptlOrigin;
} EMRSETVIEWPORTORGEX, *PEMRSETVIEWPORTORGEX,
  EMRSETWINDOWORGEX,   *PEMRSETWINDOWORGEX,
  EMRSETBRUSHORGEX,    *PEMRSETBRUSHORGEX;
```

Members

emr

Base structure for all record types.

ptlOrigin

Coordinate of origin.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **SetBrushOrgEx**, **SetViewportOrgEx**, **SetWindowOrgEx**

EMRSETWORLDTRANSFORM

The **EMRSETWORLDTRANSFORM** structure contains members for the **SetWorldTransform** enhanced metafile record.

```
typedef struct tagEMRSETWORLDTRANSFORM {
  EMR   emr;
  XFORM xform;
} EMRSETWORLDTRANSFORM, *PEMRSETWORLDTRANSFORM;
```

Members

emr

Base structure for all record types.

xform

World-space to page-space transformation data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SetWorldTransform**

EMRSTRETCHBLT

The **EMRSTRETCHBLT** structure contains members for the **StretchBlt** enhanced metafile record. Note that graphics device interface (GDI) converts the device-dependent bitmap into a device-independent bitmap (DIB) before storing it in the metafile record.

```
typedef struct tagEMRSTRETCHBLT {
  EMR      emr;
  RECTL    rclBounds;
  LONG     xDest;
  LONG     yDest;
  LONG     cxDest;
  LONG     cyDest;
  DWORD    dwRop;
  LONG     xSrc;
  LONG     ySrc;
  XFORM    xformSrc;
  COLORREF crBkColorSrc;
  DWORD    iUsageSrc;
  DWORD    offBmiSrc;
  DWORD    cbBmiSrc;
  DWORD    offBitsSrc;
  DWORD    cbBitsSrc;
  LONG     cxSrc;
  LONG     cySrc;
} EMRSTRETCHBLT, *PEMRSTRETCHBLT;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

xDest
Logical x-coordinate of the upper-left corner of the destination rectangle.

yDest
Logical y-coordinate of the upper-left corner of the destination rectangle.

cxDest
Logical width of the destination rectangle.

cyDest
Logical height of the destination rectangle.

dwRop
Raster-operation code. These codes define how the color data of the source rectangle is to be combined with the color data of the destination rectangle to achieve the final color.

xSrc
Logical x-coordinate of the upper-left corner of the source rectangle.

ySrc
Logical y-coordinate of the upper-left corner of the source rectangle.

xformSrc
World-space to page-space transformation of the source device context.

crBkColorSrc
Background color (the RGB value) of the source device context.To make a **COLORREF** value, use the **RGB** macro.

iUsageSrc
Value of the **bmiColors** member of the **BITMAPINFO** structure. The **iUsageSrc** member can be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

offBmiSrc
Offset to the source **BITMAPINFO** structure.

cbBmiSrc
Size of the source **BITMAPINFO** structure.

offBitsSrc
Offset to source bitmap bits.

cbBitsSrc
Size of source bitmap bits.

cxSrc
Width of the source rectangle.

cySrc
Height of the source rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **COLORREF**, RGB, **StretchBlt**

EMRSTRETCHDIBITS

The **EMRSTRETCHDIBITS** structure contains members for the **StretchDIBits** enhanced metafile record.

```
typedef struct tagEMRSTRETCHDIBITS {
  EMR    emr;
  RECTL  rclBounds;
  LONG   xDest;
  LONG   yDest;
  LONG   xSrc;
  LONG   ySrc;
  LONG   cxSrc;
  LONG   cySrc;
  DWORD  offBmiSrc;
  DWORD  cbBmiSrc;
  DWORD  offBitsSrc;
  DWORD  cbBitsSrc;
  DWORD  iUsageSrc;
  DWORD  dwRop;
  LONG   cxDest;
  LONG   cyDest;
} EMRSTRETCHDIBITS, *PEMRSTRETCHDIBITS;
```

Members

emr
Base structure for all record types.

rclBounds
Bounding rectangle, in device units.

xDest
Logical x-coordinate of the upper-left corner of the destination rectangle.

yDest
Logical y-coordinate of the upper-left corner of the destination rectangle.

xSrc

Logical x-coordinate of the upper-left corner of the source rectangle.

ySrc

Logical y-coordinate of the upper-left corner of the source rectangle.

cxSrc

Width of the source rectangle.

cySrc

Height of the source rectangle.

offBmiSrc

Offset to the source **BITMAPINFO** structure.

cbBmiSrc

Size of the source **BITMAPINFO** structure.

offBitsSrc

Offset to source bitmap bits.

cbBitsSrc

Size of source bitmap bits.

iUsageSrc

Value of the **bmiColors** member of the **BITMAPINFO** structure. The **iUsageSrc** member can be either the DIB_PAL_COLORS or DIB_RGB_COLORS value.

dwRop

Raster-operation code. These codes define how the color data of the source rectangle is to be combined with the color data of the destination rectangle to achieve the final color.

cxDest

Logical width of the destination rectangle.

cyDest

Logical height of the destination rectangle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **BITMAPINFO**, **StretchDIBits**

EMRTEXT

The **EMRTEXT** structure contains members for text output.

```
typedef struct tagEMRTEXT {
  POINTL ptlReference;
  DWORD  nChars;
  DWORD  offString;
  DWORD  fOptions;
  RECTL  rcl;
  DWORD  offDx;
} EMRTEXT, *PEMRTEXT;
```

Members

ptlReference
Logical coordinates of the reference point used to position the string.

nChars
Number of characters in string.

offString
Offset to string.

fOptions
Value specifying how to use the application-defined rectangle. This member can be a combination of the ETO_CLIPPED and ETO_OPAQUE values.

rcl
Optional clipping and/or opaquing rectangle.

offDx
Offset to intercharacter spacing array.

Remarks

The **EMRTEXT** structure is used as a member in the **EMREXTTEXTOUT** and **EMRPOLYTEXTOUT** structures.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

EMRTRANSPARENTBLT

The **EMRTRANSPARENTBLT** structure contains members for the**TransparentBLT** enhanced metafile record.

```
typedef struct tagEMRTRANSPARENTBLT {
  EMR      emr;
  RECTL    rclBounds;
  LONG     xDest;
  LONG     yDest;
  LONG     cxDest;
  LONG     cyDest;
  DWORD    dwRop;
  LONG     xSrc;
  LONG     ySrc;
  XFORM    xformSrc;
  COLORREF crBkColorSrc;
  DWORD    iUsageSrc;
  DWORD    offBmiSrc;
  DWORD    cbBmiSrc;
  DWORD    offBitsSrc;
  DWORD    cbBitsSrc;
  LONG     cxSrc;
  LONG     cySrc;
} EMREMRTRANSPARENTBLT, *PEMREMRTRANSPARENTBLT;
```

Members

emr

Base structure for all record types.

rclBounds

Inclusive bounds, in device units

xDest

Logical x coordinate of the upper-left corner of the destination rectangle.

yDest

Logical y coordinate of the upper-left corner of the destination rectangle.

cxDest

Logical width of the destination rectangle.

cyDest

Logical height of the destination rectangle.

dwRop

Stores the transparent color.

xSrc

Logical x coordinate of the upper-left corner of the source rectangle.

ySrc
Logical y coordinate of the upper-left corner of the source rectangle.

xformSrc
World-space to page-space transformation of the source device context.

crBkColorSrc
Background color (the RGB value) of the source device context. To make a **COLORREF** value, use the **RGB** macro.

iUsageSrc
Source bitmap information color table usage (DIB_RGB_COLORS).

offBmiSrc
Offset to the source **BITMAPINFO** structure.

cbBmiSrc
Size of the source **BITMAPINFO** structure.

offBitsSrc
Offset to the source bitmap bits.

cbBitsSrc
Size of the source bitmap bits.

cxSrc
Width of the source rectangle.

cySrc
Height of the source rectangle.

Remarks

This structure is to be used during metafile playback.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **Metafiles**, **TransparentBLT**, **BITMAPINFO**, **COLORREF**, RGB

Enhanced Metafile Records with No Parameters

Contains data for the **AbortPath**, **BeginPath**, **EndPath**, **CloseFigure**, **FlattenPath**, **WidenPath**, **SetMetaRgn**, **SaveDC**, and **RealizePalette** enhanced metafile records.

```
typedef struct tagABORTPATH {
  EMR emr;
} EMRABORTPATH,       *PEMRABORTPATH,
  EMRBEGINPATH,       *PEMRBEGINPATH,
  EMRENDPATH,         *PEMRENDPATH,
  EMRCLOSEFIGURE,     *PEMRCLOSEFIGURE,
  EMRFLATTENPATH,     *PEMRFLATTENPATH,
  EMRWIDENPATH,       *PEMRWIDENPATH,
  EMRSETMETARGN,      *PEMRSETMETARGN,
  EMRSAVEDC,          *PEMRSAVEDC,
  EMRREALIZEPALETTE,  *PEMRREALIZEPALETTE;
```

Members

emr

Base structure for all record types.

+ See Also

Metafiles Overview, Enhanced Metafile Structures

Enhanced Metafile Records with One Parameter

Contains parameters for the **SelectClipPath**, **SetBkMode**, **SetMapMode**, **SetPolyFillMode**, **SetROP2**, **SetStretchBltMode**, **SetTextAlign**, **SetICMMode**, and **SetLayout** enhanced metafile records.

```
typedef struct tagEMRSELECTCLIPPATH {
  EMR   emr;
  DWORD iMode;
} EMRSELECTCLIPPATH,     *PEMRSELECTCLIPPATH,
  EMRSETBKMODE,          *PEMRSETBKMODE,
  EMRSETMAPMODE,         *PEMRSETMAPMODE,
  EMRSETPOLYFILLMODE,    *PEMRSETPOLYFILLMODE,
  EMRSETROP2,            *PEMRSETROP2,
  EMRSETSTRETCHBLTMODE,  *PEMRSETSTRETCHBLTMODE,
  EMRSETTEXTALIGN,           *PEMRSETTEXTALIGN,
  EMRSETICMMODE,             *PEMRSETICMMODE,
  EMRSETLAYOUT,           *PEMRSETLAYOUT
```

Members

emr

Base structure for all record types.

iMode

Value and meaning that varies depending on the function contained in the enhanced metafile record. For a description of this member, see the documentation of the functions contained in this record.

See Also

Metafiles Overview, Enhanced Metafile Structures, **SelectClipPath**, **SetBkMode**, **SetMapMode**, **SetPolyFillMode**, **SetROP2**, **SetStretchBltMode**, **SetTextAlign**

ENHMETAHEADER

The **ENHMETAHEADER** structure contains enhanced-metafile data such as the dimensions of the picture stored in the enhanced metafile, the count of records in the enhanced metafile, the resolution of the device on which the picture was created, and so on.

This structure is always the first record in an enhanced metafile.

```
typedef struct tagENHMETAHEADER {
  DWORD iType;
  DWORD nSize;
  RECTL rclBounds;
  RECTL rclFrame;
  DWORD dSignature;
  DWORD nVersion;
  DWORD nBytes;
  DWORD nRecords;
  WORD  nHandles;
  WORD  sReserved;
  DWORD nDescription;
  DWORD offDescription;
  DWORD nPalEntries;
  SIZEL szlDevice;
  SIZEL szlMillimeters;
```

(continued)

(continued)

```
#if (WINVER >= 0x0400)
  DWORD cbPixelFormat;
  DWORD offPixelFormat;
  DWORD bOpenGL;
#endif /* WINVER >= 0x0400 */
#if (WINVER >= 0x0500)
  SIZEL szlMicrometers
#endif /* WINVER >= 0x0500 */
} ENHMETAHEADER, *PENHMETAHEADER;
```

Members

iType

Specifies the record type. This member must specify the value assigned to the EMR_HEADER constant.

nSize

Specifies the structure size, in bytes.

rclBounds

Specifies the dimensions, in device units, of the smallest rectangle that can be drawn around the picture stored in the metafile. This rectangle is supplied by graphics device interface (GDI). Its dimensions include the right and bottom edges.

rclFrame

Specifies the dimensions, in .01-millimeter units, of a rectangle that surrounds the picture stored in the metafile. This rectangle must be supplied by the application that creates the metafile. Its dimensions include the right and bottom edges.

dSignature

Specifies a double word signature. This member must specify the value assigned to the ENHMETA_SIGNATURE constant.

nVersion

Specifies the metafile version. The current version value is 0x10000.

nBytes

Specifies the size of the enhanced metafile, in bytes.

nRecords

Specifies the number of records in the enhanced metafile.

nHandles

Specifies the number of handles in the enhanced-metafile handle table. (Index zero in this table is reserved.)

sReserved

Reserved; must be zero.

nDescription

Specifies the number of characters in the array that contains the description of the enhanced metafile's contents. This member should be set to zero if the enhanced metafile does not contain a description string.

offDescription

Specifies the offset from the beginning of the **ENHMETAHEADER** structure to the array that contains the description of the enhanced metafile's contents. This member should be set to zero if the enhanced metafile does not contain a description string.

nPalEntries

Specifies the number of entries in the enhanced metafile's palette.

szlDevice

Specifies the resolution of the reference device, in pixels.

szlMillimeters

Specifies the resolution of the reference device, in millimeters.

cbPixelFormat

Windows 95/98, Windows NT4.0 and later: Specifies the size of the last recorded pixel format in a metafile. If a pixel format is set in a reference DC at the start of recording, *cbPixelFormat* is set to the size of the **PIXELFORMATDESCRIPTOR**. When no pixel format is set when a metafile is recorded, this member is set to zero. If more than a single pixel format is set, the header points to the last pixel format.

offPixelFormat

Windows 95/98, Windows NT4.0 and later: Specifies the offset of pixel format used when recording a metafile. If a pixel format is set in a reference DC at the start of recording or during recording, *offPixelFormat* is set to the offset of the **PIXELFORMATDESCRIPTOR** in the metafile. If no pixel format is set when a metafile is recorded, this member is set to zero. If more than a single pixel format is set, the header points to the last pixel format.

bOpenGL

Windows 95/98, Windows NT4.0 and later: Specifies whether any OpenGL records are present in a metafile. *bOpenGL* is a simple Boolean flag that you can use to determine whether an enhanced metafile requires OpenGL handling. When a metafile contains OpenGL records, *bOpenGL* is TRUE; otherwise it is FALSE.

szlMicrometers

Windows 98, Windows 2000: Size of the reference device in micrometers.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Metafiles Overview, Enhanced Metafile Structures, **ENHMETARECORD**

ENHMETARECORD

The **ENHMETARECORD** structure contains data that describes a graphics device interface (GDI) function used to create part of a picture in an enhanced-format metafile.

```
typedef struct tagENHMETARECORD {
  DWORD iType;
  DWORD nSize;
  DWORD dParm[1];
} ENHMETARECORD, *PENHMETARECORD;
```

Members

iType
Specifies the record type.

nSize
Specifies the size of the record, in bytes.

dParm
Specifies an array of parameters passed to the GDI function identified by the record.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **ENHMETAHEADER**

HANDLETABLE

The **HANDLETABLE** structure is an array of handles, each of which identifies a graphics device interface (GDI) object.

```
typedef struct tagHANDLETABLE {
  HGDIOBJ objectHandle[1];
} HANDLETABLE, *PHANDLETABLE;
```

Members

objectHandle
Contains an array of handles.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **EnhMetaFileProc**, **EnumMetaFileProc**

POINTL

The **POINTL** structure contains the coordinates of a point.

```
typedef struct _POINTL {
  LONG x;
  LONG y;
} POINTL, *PPOINTL;
```

Members

x

Specifies the horizontal (x) coordinate of the point.

y

Specifies the vertical (y) coordinate of the point.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in windef.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures

RECTL

The **RECTL** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct _RECTL {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECTL, *PRECTL;
```

Members

left

Specifies the x-coordinate of the upper-left corner of the rectangle.

top

Specifies the y-coordinate of the upper-left corner of the rectangle.

right

Specifies the x-coordinate of the lower-right corner of the rectangle.

bottom

Specifies the y-coordinate of the lower-right corner of the rectangle.

Remarks

When **RECTL** is passed to the **FillRect** function, the rectangle is filled up to, but not including, the right column and bottom row of pixels. This structure is identical to the **RECT** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in windef.h; include windows.h.

See Also

Metafiles Overview, Enhanced Metafile Structures, **FillRect**, **RECT**, **SMALL_RECT**

CHAPTER 15

Painting and Drawing

This overview describes how the system manages output to the screen and explains what applications must do to draw in a window. In particular, this overview describes *display device contexts* (or, more simply, *display DCs*) and how to prepare and use them. The overview does not explain how to use graphical device interface (GDI) functions to generate output, or how to print.

About Painting and Drawing

Nearly all applications use the screen to display the data they manipulate. An application paints images, draws figures, and writes text so that the user can view data as it is created, edited, and printed. The Win32 API provides rich support for painting and drawing, but, because of the nature of multitasking operating systems, applications must cooperate with one another when accessing the screen.

To keep all applications functioning smoothly and cooperatively, the system manages all output to the screen. Applications use windows as their primary output device instead of the screen itself. The system supplies display device contexts that uniquely correspond to the windows. Applications use display device contexts to direct their output to the specified windows. Drawing in a window (directing output to it) prevents an application from interfering with the output of other applications and allows applications to coexist with one another while still taking full advantage of the graphics capabilities of the system.

When to Draw in a Window

An application draws in a window at a variety of times: when first creating a window, when changing the size of the window, when moving the window from behind another window, when minimizing or maximizing the window, when displaying data from an opened file, and when scrolling, changing, or selecting a portion of the displayed data.

The system manages actions, such as moving and sizing a window. If an action affects the content of the window, the system marks the affected portion of the window as ready for updating and, at the next opportunity, sends a **WM_PAINT** message to the window procedure of the window. The message is a signal to the application to determine what must be updated and to carry out the necessary drawing.

Some actions are managed by the application, such as displaying open files and selecting displayed data. For these actions, an application can mark for updating the portion of the window affected by the action, causing a **WM_PAINT** message to be sent at the next opportunity. If an action requires immediate feedback, the application can draw while the action takes place, without waiting for **WM_PAINT**. For example, a typical application highlights the area the user selects, instead of waiting for the next **WM_PAINT** message to update the area.

In all cases, an application can draw in a window as soon as it is created. To draw in the window, the application must first retrieve a handle to a display device context for the window. Ideally, an application carries out most of its drawing operations during the processing of **WM_PAINT** messages. In this case, the application retrieves a display device context by calling the **BeginPaint** function. If an application draws at any other time, such as from within **WinMain** or during the processing of keyboard or mouse messages, it calls the **GetDC** or **GetDCEx** function to retrieve the display DC.

The WM_PAINT Message

Typically, an application draws in a window in response to a **WM_PAINT** message. The system sends this message to a window procedure when changes to the window have altered the content of the client area. The system sends the message only if there are no other messages in the application message queue.

Upon receiving a **WM_PAINT** message, an application can call **BeginPaint** to retrieve the display device context for the client area, and use it in calls to GDI functions to carry out whatever drawing operations are necessary to update the client area. After completing the drawing operations, the application calls the **EndPaint** function to release the display device context.

Before **BeginPaint** returns the display device context, the system prepares the device context for the specified window. It first sets the clipping region for the device context to be equal to the intersection of the portion of the window that needs updating and the portion that is visible to the user. Only those portions of the window that have changed are redrawn. Attempts to draw outside this region are clipped and do not appear on the screen.

The system also can send **WM_NCPAINT** and **WM_ERASEBKGND** messages to the window procedure before **BeginPaint** returns. These messages direct the application to draw the nonclient area and window background. The nonclient *area* is the part of a window that is outside of the client area. The area includes features such as the title bar, window menu (also known as the **System** menu), and scroll bars. Most applications rely on the default window function, **DefWindowProc**, to draw this area and therefore pass the **WM_NCPAINT** message to this function. The *window background* is the color or pattern that a window is filled with before other drawing operations begin. The background covers any images previously in the window or on the screen under the window. If a window belongs to a window class having a class background brush, the **DefWindowProc** function draws the window background automatically.

BeginPaint fills a **PAINTSTRUCT** structure with information, such as the dimensions of the portion of the window to be updated and a flag indicating whether the window background has been drawn. The application can use this information to optimize drawing. For example, it can use the dimensions of the update region, specified by the

rcPaint member, to limit drawing to only those portions of the window that need updating. If an application has very simple output, it can ignore the update region and draw in the entire window, relying on the system to discard (clip) any unneeded output. Because the system clips drawing that extends outside the clipping region, only drawing that is in the update region is visible.

BeginPaint sets the update region of a window to NULL. This clears the region, preventing it from generating subsequent **WM_PAINT** messages. If an application processes a **WM_PAINT** message but does not call **BeginPaint** or, otherwise, clear the update region, the application continues to receive **WM_PAINT** messages as long as the region is not empty. In all cases, an application must clear the update region before returning from the **WM_PAINT** message.

After the application finishes drawing, it should call **EndPaint**. For most windows, **EndPaint** releases the display device context, making it available to other windows. **EndPaint** also shows the caret, if it was previously hidden by **BeginPaint**. **BeginPaint** hides the caret to prevent drawing operations from corrupting it.

The Update Region

The *update region* identifies the portion of a window that is out-of-date or invalid and in need of redrawing. The system uses the update region to generate **WM_PAINT** messages for applications and to minimize the time applications spend bringing the contents of their windows up to date. The system adds only the invalid portion of the window to the update region, requiring only that portion to be drawn.

When the system determines that a window needs updating, it sets the dimensions of the update region to the invalid portion of the window. Setting the update region does not immediately cause the application to draw. Instead, the application continues retrieving messages from the application message queue until no messages remain. The system then checks the update region, and if the region is not empty (non-NULL), it sends a **WM_PAINT** message to the window procedure.

An application can use the update region to generate its **WM_PAINT** messages. For example, an application that loads data from open files typically sets the update region while loading, so that new data is drawn during processing of the next **WM_PAINT** message. In general, an application should not draw at the time its data changes, but route all drawing operations through the **WM_PAINT** message.

Invalidating and Validating the Update Region

An application invalidates a portion of a window and sets the update region by using the **InvalidateRect** or **InvalidateRgn** function. These functions add the specified rectangle or region (in client coordinates) to the update region, combining the rectangle or region with anything the system or the application might have previously added to the update region.

The **InvalidateRect** and **InvalidateRgn** functions do not generate **WM_PAINT** messages. Instead, the system accumulates the changes made by these functions and its own changes while a window processes other messages in its message queue. By accumulating changes, a window processes all changes at once instead of updating bits and pieces one step at a time.

The **ValidateRect** and **ValidateRgn** functions validate a portion of the window by removing a specified rectangle or region from the update region. These functions are typically used when the window has updated a specific part of the screen in the update region before receiving the **WM_PAINT** message.

Retrieving the Update Region

The **GetUpdateRect** and **GetUpdateRgn** functions retrieve the current update region for the window. **GetUpdateRect** retrieves the smallest rectangle (in client coordinates) that encloses the entire update region. **GetUpdateRgn** retrieves the update region itself. These functions can be used to calculate the current size of the update region to determine where to carry out a drawing operation.

BeginPaint also retrieves the dimensions of the smallest rectangle enclosing the current update region, copying the dimensions to the **rcPaint** member in the **PAINTSTRUCT** structure. Because **BeginPaint** validates the update region, any call to **GetUpdateRect** and **GetUpdateRgn** immediately after a call to **BeginPaint** returns an empty update region.

Excluding the Update Region

The **ExcludeUpdateRgn** function excludes the update region from the clipping region for the display device context. This function is useful when drawing in a window other than when a **WM_PAINT** message is processing. It prevents drawing in the areas that will be updated during the next **WM_PAINT** message.

Synchronous and Asynchronous Drawing

Most drawing carried out during processing of the **WM_PAINT** message is asynchronous; that is, there is a delay between the time a portion of the window is invalidated and the time **WM_PAINT** is sent. During the delay, the application typically retrieves messages from the queue and carries out other tasks. The reason for the delay is that the system generally treats drawing in a window as a low-priority operation, and works as though user-input messages and messages that can affect the position or size of a window will be processed before **WM_PAINT**.

In some cases, it is necessary for an application to draw synchronously; that is, draw in the window immediately after invalidating a portion of the window. A typical application draws its main window immediately after creating the window to signal the user that the application has started successfully. The system draws some control windows synchronously, such as buttons, because such windows serve as the focus for user input. Although any window with a simple drawing routine can be drawn synchronously, all such drawing should be done quickly and not interfere with the application's ability to respond to user input.

The **UpdateWindow** and **RedrawWindow** functions allow for synchronous drawing. **UpdateWindow** sends a **WM_PAINT** message directly to the window if the update region is not empty. **RedrawWindow** also sends a **WM_PAINT** message, but gives the application greater control over how to draw the window, such as whether to draw the nonclient area and window background or whether to send the message regardless of whether the update region is empty. These functions send the **WM_PAINT** message directly to the window, regardless of the number of other messages in the application message queue.

Any window requiring time-consuming drawing operations should be drawn asynchronously to prevent pending messages from being blocked as the window is drawn. Also, any application that frequently invalidates small portions of a window should allow these invalid portions to consolidate into a single asynchronous **WM_PAINT** message, instead of a series of synchronous **WM_PAINT** messages.

Drawing Without the WM_PAINT Message

Although applications carry out most drawing operations while the **WM_PAINT** message is processing, it is sometimes more efficient for an application to draw directly in a window without relying on the **WM_PAINT** message. This can be useful when the user needs immediate feedback, such as when selecting text and dragging or sizing an object. In such cases, the application usually draws while processing keyboard or mouse messages.

To draw in a window without using a **WM_PAINT** message, the application uses the **GetDC** or **GetDCEx** function to retrieve a display device context for the window. With the display device context, the application can draw in the window and avoid intruding into other windows. When the application has finished drawing, it calls the **ReleaseDC** function to release the display device context for use by other applications.

When drawing without using a **WM_PAINT** message, the application usually does not invalidate the window. Instead, it draws in such a fashion that it can easily restore the window and remove the drawing. For example, when the user selects text or an object, the application typically draws the selection by inverting whatever is already in the window. The application can remove the selection and restore the original contents of the window by inverting again.

The application is responsible for carefully managing any changes it makes to the window. In particular, if an application draws a selection and an intervening **WM_PAINT** message occurs, the application must ensure that any drawing done during the message does not corrupt the selection. To avoid this, many applications remove the selection, carry out usual drawing operations, and then restore the selection when drawing is complete.

Window Coordinate System

The coordinate system for a window is based on the coordinate system of the display device. The basic unit of measure is the device unit (typically, the pixel). Points on the screen are described by x-coordinate and y-coordinate pairs. The x-coordinates increase to the right; the y-coordinates increase from top to bottom. The origin (0,0) for the system depends on the type of coordinates being used.

The system and applications specify the position of a window on the screen in *screen coordinates*. For screen coordinates, the origin is the upper-left corner of the screen. The full position of a window is described often by a **RECT** structure containing the screen coordinates of two points that define the upper-left and lower-right corners of the window.

The system and applications specify the position of points in a window by using *client coordinates*. The origin in this case is the upper-left corner of the window or client area. Client coordinates ensure that an application can use consistent coordinate values while drawing in the window, regardless of the position of the window on the screen.

The dimensions of the client area also are described by a **RECT** structure that contains client coordinates for the area. In all cases, the upper-left coordinate of the rectangle is included in the window or client area, while the lower-right coordinate is excluded. Graphics operations in a window or client area are excluded from the right and lower edges of the enclosing rectangle.

Occasionally, applications might be required to map coordinates in one window to those of another window. An application can map coordinates by using the **MapWindowPoints** function. If one of the windows is the desktop window, the function effectively converts screen coordinates to client coordinates, and vice versa; the desktop window is specified always in screen coordinates.

Window Regions

In addition to the update region, every window has a *visible region* that defines the window portion visible to the user. The system changes the visible region for the window whenever the window changes size or whenever another window is moved such that it obscures or exposes a portion of the window. Applications cannot change the visible region directly, but the system automatically uses the visible region to create the clipping region for any display device context retrieved for the window.

The *clipping region* determines where the system permits drawing. When the application retrieves a display device context using the **BeginPaint**, **GetDC**, or **GetDCEx** function, the system sets the clipping region for the device context to the intersection of the visible region and the update region. Applications can change the clipping region by using functions such as **SetWindowRgn**, **SelectClipPath** and **SelectClipRgn**, to further limit drawing to a particular portion of the update area.

The WS_CLIPCHILDREN and WS_CLIPSIBLINGS styles further specify how the system calculates the visible region for a window. If a window has one or both of these styles, the visible region excludes any child window or sibling windows (windows having the same parent window). Therefore, any drawing that would intrude otherwise in these windows will always be clipped.

Window Background

The window background is the color or pattern used to fill the client area before a window begins drawing. The window background covers whatever was on the screen before the window was moved there, erasing existing images and preventing the application's new output from being mixed with unrelated information.

The system paints the background for a window or gives the window the opportunity to do so by sending it a **WM_ERASEBKGND** message when the application calls **BeginPaint**. If an application does not process the message, but passes it to **DefWindowProc**, the system erases the background by filling it with the pattern in the background brush specified by the window's class. If the brush is not valid or the class has no background brush, the system sets the **fErase** member in the **PAINTSTRUCT** structure that **BeginPaint** returns, but carries out no other action. The application, then, has a second chance to draw the window background, if necessary.

If it processes **WM_ERASEBKGND**, the application should use the message's *wParam* parameter to draw the background. This parameter contains a handle to the display device context for the window. After drawing the background, the application should return a nonzero value. This ensures that **BeginPaint** does not erroneously set the **fErase** member of the **PAINTSTRUCT** structure to a nonzero value (indicating the background should be erased) when the application processes the subsequent **WM_PAINT** message.

An application can define a class background brush by assigning a brush handle or a system color value to the **hbrBackground** member of the **WNDCLASS** structure when registering the class with the **RegisterClass** function. The **GetStockObject** or **CreateSolidBrush** function can be used to create a brush handle. A system color value can be one of those defined for the **SetSysColors** function. (The value must be increased by one before it is assigned to the member.)

An application can process the **WM_ERASEBKGND** message even though a class background brush is defined. This is typical in applications that enable the user to change the window background color or pattern for a specified window without affecting other windows in the class. In such cases, the application must not pass the message to **DefWindowProc**.

It is not necessary for an application to align brushes, because the system draws the brush using the window origin as the point of reference. Given this, the user can move the window without affecting the alignment of pattern brushes.

Minimized Windows

The system reduces an application's main window (overlapping style) to a minimized window when the user clicks **Minimize** from the window menu or, the application calls the **ShowWindow** function and specifies a value such as SW_MINIMIZE. Minimizing a window speeds up system performance by reducing the amount of work an application must do when updating its main window.

For a typical application, the system draws an icon, called the class icon, when the window is minimized, labeling the icon with the name of the window. The class icon, a static image that represents the application, is specified by the application when it registers the window class. The application assigns a handle to the class icon to the **hIcon** member of **WNDCLASS** before calling **RegisterClass**. The application can use the **LoadIcon** function to retrieve the icon handle.

Before drawing the class icon, the system sends a **WM_ICONERASEBKGND** message to the window procedure, enabling the application to prepare the background for drawing the icon by setting the best background colors possible for the icon. This is useful for applications that combine the icon with the current background colors. If the application processes the message, it should use the display device context provided with the message to draw the background (the *wParam* parameter contains a handle to the display DC). If the application does not process the **WM_ICONERASEBKGND** message, it should pass the message to **DefWindowProc**; the function fills the icon area with the current desktop color and pattern. After sending the **WM_ICONERASEBKGND** message, the system sends the **WM_PAINTICON** message to the window procedure. The application should forward immediately this internal message to **DefWindowProc**.

The system does not require that a window class have a class icon. If an application sets the **hIcon** member of **WNDCLASS** to NULL, a class icon is not defined. In this case, the system sends the **WM_ERASEBKGND** message (instead of **WM_ICONERASEBKGND**) to a window of the class whenever the window must paint the icon background. The system then sends a **WM_PAINT** message and the application draws an icon or another image representing the minimized window. In such cases, the application must determine when the window is minimized and draw accordingly. It can do so by calling the **IsIconic** function. If the function returns TRUE, the window is minimized. If an application has no class icon and fails to process **WM_ERASEBKGND** and **WM_PAINT**, the area that the system reserves for the application's icon will contain whatever was on the screen previously.

Resized Windows

The system changes the size of a window when the user chooses window menu commands, such as **Size** and **Maximize**, or when the application calls functions, such as the **SetWindowPos** function. When a window changes size, the system assumes that the contents of the previously exposed portion of the window are not affected and do not need to be redrawn. The system invalidates only the newly exposed portion of the window, which saves time when the eventual **WM_PAINT** message is processed by the application. In this case, **WM_PAINT** is not generated when the size of the window is reduced.

For some windows, any change to the size of the window invalidates the contents. For example, a clock application that adapts the face of the clock to fit neatly within its window must redraw the clock whenever the window changes size. To force the system to invalidate the entire client area of the window when a vertical, horizontal, or both vertical and horizontal change is made, an application must specify the CS_VREDRAW or CS_HREDRAW style, or both, when registering the window class. Any window belonging to a window class having these styles is invalidated each time the user or the application changes the size of the window.

Nonclient Area

The system sends a **WM_NCPAINT** message to the window whenever a part of the nonclient area of the window, such as the title bar, menu bar, or window frame, must be updated. The system can send also other messages to direct a window to update a portion of its client area; for example, when a window becomes active or inactive, it sends the **WM_NCACTIVATE** message to update its title bar. In general, processing these messages for standard windows is not recommended, because the application must be able to draw all the required parts of the nonclient area for the window. For this reason, most applications pass these messages to **DefWindowProc** for default processing.

An application that creates custom nonclient areas for its windows must process these messages. When doing so, the application must use a window device context to carry out drawing in the window. The *window device context* enables the application to draw in all portions of the window, including the nonclient area. An application retrieves a window device context by using the **GetWindowDC** or **GetDCEx** function and, when drawing is complete, must release the window device context by using the **ReleaseDC** function.

The system maintains an update region for the nonclient area. When an application receives a **WM_NCPAINT** message, the *wParam* parameter contains a handle to a region defining the dimensions of the update region. The application can use the handle to combine the update region with the clipping region for the window device context. The system does not automatically combine the update region when retrieving the window device context unless the application uses **GetDCEx** and specifies both the region handle and the DCX_INTERSECTRGN flag. If the application does not combine the update region, only drawing operations that would otherwise extend outside the window are clipped. The application is not responsible for clearing the update region, regardless of whether it uses the region.

If an application processes the **WM_NCACTIVATE** message, after processing it must return TRUE to direct the system to complete the change of active window. If the window is minimized when the application receives the **WM_NCACTIVATE** message, it should pass the message to **DefWindowProc**. In such cases, the default function redraws the label for the icon.

Child Windows

A child window is a window with the WS_CHILD or WS_CHILDWINDOW style. Like other window styles, child windows receive **WM_PAINT** messages to prompt updating. Each child window has an update region, which either the system or the application can set to generate eventual **WM_PAINT** messages.

A child window's update and visible regions are affected by the child's parent window; this is not true for windows of other styles. The system often sets the child window's update region when it sets the parent window's update region, causing the child window to receive **WM_PAINT** messages when the parent window receives them. The system limits the location of the child window's visible region to within the client area of the parent window, and clips any portion of the child window moved outside the parent window.

The system sets the update region for a child window whenever part of the parent window's update region includes a portion of the child window. In such cases, the system first sends a **WM_PAINT** message to the parent window and then sends a message to the child window, allowing the child to restore any portions of the window that the parent might have drawn over.

The system does not set the parent's update region when the child's is set. An application cannot generate a **WM_PAINT** message for the parent window by invalidating the child window. Similarly, an application cannot generate a **WM_PAINT** message for the child by invalidating a portion of the parent's client area that lies entirely under the child window. In such cases, neither window receives a **WM_PAINT** message.

An application can prevent a child window's update region from being set when the parent window's is set by specifying the WS_CLIPCHILDREN style when creating the parent window. When this style is set, the system excludes the child windows from the parent's visible region and therefore ignores any portion of the update region that may contain the child windows. When the application paints in the parent window, any drawing that would cover the child window is clipped, making a subsequent **WM_PAINT** message to the child window unnecessary.

The update and visible regions of a child window are also affected by the child window's siblings. Sibling windows are any windows that have a common parent window. If sibling windows overlap, then setting the update region for one affects the update region of another, causing **WM_PAINT** messages to be sent to both windows. Sibling windows receive **WM_PAINT** messages in the reverse order of their position in the Z order. Given this, the window highest in the Z order (on the top) receives its **WM_PAINT** message last, and vice versa.

Sibling windows are not automatically clipped. One sibling can draw over another overlapping sibling even if the window that is drawing has a lower position in the Z order. An application can prevent this by specifying the WS_CLIPSIBLINGS style when creating the windows. When this style is set, the system excludes all portions of an overlapping sibling window from a window's visible region, if the overlapping sibling window has a higher position in the Z order.

Note The update and visible regions for windows that have the WS_POPUP or WS_POPUPWINDOW style are not affected by their parent windows.

About Display Device Contexts

A display device context is a device context (DC), created by the system, that an application uses to paint and draw a window. The system prepares each display device context for output to a window, setting the drawing objects, colors, and modes for the window instead of for the display device. When the application supplies the display device context through calls to GDI functions, GDI uses the information in the context to generate output in the specified window without intruding on other windows or other parts of the screen.

The system provides five kinds of display device contexts:

Type	Meaning
Common	Permits drawing in the client area of a specified window.
Class	Permits drawing in the client area of a specified window.
Parent	Permits drawing anywhere in the window. Although the parent device context also permits drawing in the parent window, it is not intended to be used in this way.
Private	Permits drawing in the client area of a specified window.
Window	Permits drawing anywhere in the window.

The system supplies a common, class, parent, or private device context to a window based on the type of display device context specified in that window's class style. The system supplies a window device context only when the application explicitly requests one—for example, by calling the **GetWindowDC** or **GetDCEx** function. In all cases, an application can use the **WindowFromDC** function to determine which window a display DC currently represents.

Display Device Context Cache

The system maintains a cache of display device contexts that it uses for common, parent, and window device contexts. The system retrieves a device context from the cache whenever an application calls the **GetDC** or **BeginPaint** function; the system returns the DC to the cache when the application subsequently calls the **ReleaseDC** or **EndPaint** function.

In 16-bit Windows, the cache contains five display device contexts, but only five device contexts from the cache can be active at a time. To ensure that other applications have access to these device contexts, an application must release a device context immediately after using it. Failure to do so eventually causes the application to fail.

There is no predetermined limit on the amount of device contexts that a cache can hold; the system creates a new display device context for the cache if none is available. Given this, a Win32-based application can have more than five active device contexts from the cache at a time. However, the application must continue to release these device contexts after use. Because new display device contexts for the cache are allocated in the application's heap space, failing to release the device contexts eventually consumes all available heap space. The system indicates this failure by returning an error when it cannot allocate space for the new device context. Other functions unrelated to the cache may also return errors.

Display Device Context Defaults

Upon first creating a display device context, the system assigns default values for the attributes (that is, drawing objects, colors, and modes) that make up the device context. The following table shows the default values for the attributes of a display device context:

Attribute	Default value
Background color	Background color setting from Control Panel (typically, white).
Background mode	OPAQUE
Bitmap	None
Brush	WHITE_BRUSH
Brush origin	(0,0)
Clipping region	Entire window or client area with the update region clipped, as appropriate. Child and pop-up windows in the client area may also be clipped.
Current pen position	(0,0)
Device origin	Upper-left corner of the window or client area.
Drawing mode	R2_COPYPEN
Font	SYSTEM_FONT
Intercharacter spacing	0

Mapping mode	MM_TEXT
Palette	DEFAULT_PALETTE
Pen	BLACK_PEN
Polygon-fill mode	ALTERNATE
Stretch mode	BLACKONWHITE
Text color	Text color setting from Control Panel (typically, black).
Viewport extent	(1,1)
Viewport origin	(0,0)
Window extent	(1,1)
Window origin	(0,0)

An application can modify the values of the display device context attributes by using selection and attribute functions, such as **SelectObject**, **SetMapMode**, and **SetTextColor**. For example, an application can modify the default units of measure in the coordinate system by using **SetMapMode** to change the mapping mode.

Changes to the attribute values of a common, parent, or window device context are not permanent. When an application releases these device contexts, the current selections, such as mapping mode and clipping region, are lost as the context is returned to the cache. Changes to a class or private device context persist indefinitely. To restore them to their original defaults, an application must set explicitly each attribute.

Common Display Device Contexts

A *common device context* is used for drawing in the client area of the window. The system provides a common device context by default for any window whose window class does not explicitly specify a display device context style. Common device contexts are typically used with windows that can be drawn without extensive changes to the device context attributes. Common device contexts are convenient because they do not require additional memory or system resources, but they can be inconvenient if the application must set up many attributes before using them.

The system retrieves all common device contexts from the display device context cache. An application can retrieve a common device context immediately after the window is created. Because the common device context is from the cache, the application must always release the device context as soon as possible after drawing. After the common device context is released, it is no longer valid and the application must not attempt to draw with it. To draw again, the application must retrieve a new common device context, and continue to retrieve and release a common device context each time it draws in the window. If the application retrieves the device context handle by using the **GetDC** function, it must use the **ReleaseDC** function to release the handle. Similarly, for each **BeginPaint** function, the application must use a corresponding **EndPaint** function.

When the application retrieves the device context, the system adjusts the origin so that it aligns with the upper-left corner of the client area. It also sets the clipping region so that output to the device context is clipped to the client area. Any output that would otherwise appear outside the client area is clipped. If the application retrieves the common device context by using **BeginPaint**, the system also includes the update region in the clipping region to further restrict the output.

When an application releases a common device context, the system restores the default values for the attributes of the device context. An application that modifies attribute values must do so each time it retrieves a common device context. Releasing the device context releases any drawing objects the application might have selected into it, so the application does not need to release these objects before releasing the device context. In all cases, an application must never assume that the common device context retains nondefault selections after being released.

Private Display Device Contexts

A *private device context* enables an application to avoid retrieving and initializing a display device context each time the application must draw in a window. Private device contexts are useful for windows that require many changes to the values of the attributes of the device context to prepare it for drawing. Private device contexts reduce the time required to prepare the device context and, therefore, the time needed to carry out drawing in the window.

An application directs the system to create a private device context for a window by specifying the CS_OWNDC style in the window class. The system creates a unique private device context each time it creates a new window belonging to the class. Initially, the private device context has the same default values for attributes as a common device context, but the application can modify these at any time. The system preserves changes to the device context for the life of the window or until the application makes additional changes.

An application can retrieve a handle to the private device context by using the **GetDC** function any time after the window is created. The application must retrieve the handle only once. Thereafter, it can keep and use the handle any number of times. Because a private device context is not part of the display device context cache, an application need never release the device context by using the **ReleaseDC** function.

The system automatically adjusts the device context to reflect changes to the window, such as moving or sizing. This ensures that any overlapping windows are always properly clipped; that is, no action is required by the application to ensure clipping. However, the system does not revise the device context to include the update region. Therefore, when processing a **WM_PAINT** message, the application must incorporate the update region either by calling **BeginPaint** or retrieving the update region and intersecting it with the current clipping region. If the application does not call **BeginPaint**, it must explicitly validate the update region by using the **ValidateRect** or **ValidateRgn** function. If the application does not validate the update region, the window receives an endless series of **WM_PAINT** messages.

Because **BeginPaint** hides the caret if a window is showing it, an application that calls **BeginPaint** should also call the **EndPaint** function to restore the caret. **EndPaint** has no other effect on a private device context.

Although a private device context is convenient to use, it is expensive in terms of system resources, requiring 800 or more bytes to store. Private device contexts are recommended when performance considerations outweigh storage costs.

The system includes the private device context when sending the **WM_ERASEBKGND** message to the application. The current selections of the private device context, including mapping mode, are in effect when the application or the system processes these messages. To avoid undesirable effects, the system uses logical coordinates when erasing the background; for example, it uses the **GetClipBox** function to retrieve the logical coordinates of the area to erase and passes these coordinates to the **FillRect** function. Applications that process these messages can use similar techniques. The system supplies a window device context with the **WM_ICONERASEBKGND** message regardless of whether the corresponding window has a private device context.

An application can use the **GetDCEx** function to force the system to return a common device context for the window that has a private device context. This is useful for carrying out quick touch-ups to a window without changing the current values of the attributes of the private device context.

Class Display Device Contexts

By using a *class device context*, an application can use a single display device context for every window belonging to a specified class. Class device contexts are often used with control windows that are drawn using the same attribute values. Like private device contexts, class device contexts minimize the time required to prepare a device context for drawing.

The system supplies a class device context for a window if it belongs to a window class having the CS_CLASSDC style. The system creates the device context when creating the first window belonging to the class and then uses the same device context for all subsequently created windows in the class. Initially, the class device context has the same default values for attributes as a common device context, but the application can modify these at any time. The system preserves all changes, except for the clipping region and device origin, until the last window in the class has been destroyed. A change made for one window applies to all windows in that class.

An application can retrieve the handle for the class device context by using the **GetDC** function any time after the first window has been created. The application can keep and use the handle without releasing it because the class device context is not part of the display device context cache. If the application creates another window in the same window class, the application must retrieve the class device context again. Retrieving the device context sets the correct device origin and clipping region for the new window. After the application retrieves the class device context for a new window in the class, the device context can no longer be used to draw in the original window without again retrieving it for that window. In general, each time it must draw in a window, an application must explicitly retrieve the class device context for the window.

Applications that use class device contexts should always call **BeginPaint** when processing a **WM_PAINT** message. The function sets the correct device origin and clipping region for the window, and incorporates the update region. The application should also call **EndPaint** to restore the caret if **BeginPaint** hid it. **EndPaint** has no other effect on a class device context.

The system passes the class device context when sending the **WM_ERASEBKGND** message to the application, permitting the current attribute values to affect any drawing carried out by the application or the system when processing this message. The system supplies a window device context with the **WM_ICONERASEBKGND** message regardless of whether the corresponding window has a class device context. As it could with a window having a private device context, an application can use **GetDCEx** to force the system to return a common device context for the window that has a class device context.

Using class device contexts is not recommended.

Window Display Device Contexts

A *window device context* enables an application to draw anywhere in a window, including the nonclient area. Window device contexts are typically used by applications that process the **WM_NCPAINT** and **WM_NCACTIVATE** messages for windows with custom nonclient areas. Using a window device context is not recommended for any other purpose.

An application can retrieve a window device context by using the **GetWindowDC** or **GetDCEx** function with the DCX_WINDOW option specified. The function retrieves a window device context from the display device context cache. A window that uses a window device context must release it after drawing by using the **ReleaseDC** function as soon as possible. Window device contexts are always from the cache; the CS_OWNDC and CS_CLASSDC class styles do not affect the device context.

When an application retrieves a window device context, the system sets the device origin to the upper-left corner of the window instead of the upper-left corner of the client area. It also sets the clipping region to include the entire window, not just the client area. The system sets the current attribute values of a window device context to the same default values as a common device context. An application can change the attribute values, but the system does not preserve any changes when the device context is released.

Parent Display Device Contexts

A *parent device context* enables an application to minimize the time necessary to set up the clipping region for a window. An application typically uses parent device contexts to speed up drawing for control windows without requiring a private or class device context.

For example, the system uses parent device contexts for push-button and edit controls. Parent device contexts are intended for use with child windows only, never with top-level or pop-up windows.

An application can specify the CS_PARENTDC style to set the clipping region of the child window to that of the parent window, so that the child can draw in the parent. Specifying CS_PARENTDC enhances an application's performance, because the system does not need to keep recalculating the visible region for each child window.

Attribute values set by the parent window are not preserved for the child window; for example, the parent window cannot set the brush for its child windows. The only property preserved is the clipping region. The window must clip its own output to the limits of the window. Because the clipping region for the parent device context is identical to the parent window, the child window can potentially draw over the entire parent window, but the parent device context must not be used in this way.

The system ignores the CS_PARENTDC style if the parent window uses a private or class device context, if the parent window clips its child windows, or if the child window clips its child windows or sibling windows.

Window Update Lock

A *window update lock* is a temporary suspension of drawing in a window. The system uses the lock to prevent other windows from drawing over the tracking rectangle whenever the user moves or sizes a window. Applications can use the lock to prevent drawing if they carry out similar moving or sizing operations with their own windows.

An application uses the **LockWindowUpdate** function to set or clear a window update lock, specifying the window to lock. The lock applies to the specified window and all of its child windows. When the lock is set, the **GetDC** and **BeginPaint** functions return a display device context with a visible region that is empty. Given this, the application can continue to draw in the window, but all output is clipped. The lock persists until the application clears it by calling **LockWindowUpdate**, specifying NULL for the window. Although **LockWindowUpdate** forces a window's visible region to be empty, the function does not make the specified window invisible and does not clear the WS_VISIBLE style bit.

After the lock is set, the application can use the **GetDCEx** function, with the DCX_LOCKWINDOWUPDATE value, to retrieve a display device context to draw over the locked window. This allows the application to draw a tracking rectangle when processing keyboard or mouse messages. The system uses this method when the user moves and sizes windows. **GetDCEx** retrieves the display device context from the display device context cache, so the application must release the device context as soon as possible after drawing.

While a window update lock is set, the system creates an accumulated bounding rectangle for each locked window. When the lock is cleared, the system uses this bounding rectangle to set the update region for the window and its child windows, forcing an eventual **WM_PAINT** message. If the accumulated bounding rectangle is empty (that is, if no drawing has occurred while the lock was set), the update region is not set.

Accumulated Bounding Rectangle

The *accumulated bounding rectangle* is the smallest rectangle enclosing the portion of a window or client area affected by recent drawing operations. An application can use this rectangle to determine conveniently the extent of changes caused by drawing operations. It is sometimes used in conjunction with **LockWindowUpdate** to determine which portion of the client area must be redrawn after the update lock is cleared.

An application uses the **SetBoundsRect** function (specifying DCB_ENABLE) to begin accumulating the bounding rectangle. The system subsequently accumulates points for the bounding rectangle as the application uses the specified display device context. The application can retrieve the current bounding rectangle at any time by using the **GetBoundsRect** function. The application stops the accumulation by calling **SetBoundsRect** again, specifying the DCB_DISABLE value.

Painting and Drawing Reference

Painting and Drawing Functions

BeginPaint

The **BeginPaint** function prepares the specified window for painting and fills a **PAINTSTRUCT** structure with information about the painting.

```
HDC BeginPaint(
  HWND hwnd,            // handle to window
  LPPAINTSTRUCT lpPaint // paint information
);
```

Parameters

hwnd

[in] Handle to the window to be repainted.

lpPaint

[out] Pointer to the **PAINTSTRUCT** structure that will receive painting information.

Return Values

If the function succeeds, the return value is the handle to a display device context for the specified window.

If the function fails, the return value is NULL, indicating that no display device context is available.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **BeginPaint** function automatically sets the clipping region of the device context to exclude any area outside the update region. The update region is set by the **InvalidateRect** or **InvalidateRgn** function and by the system after sizing, moving, creating, scrolling, or any other operation that affects the client area. If the update region is marked for erasing, **BeginPaint** sends a **WM_ERASEBKGND** message to the window.

An application should not call **BeginPaint** except in response to a **WM_PAINT** message. Each call to **BeginPaint** must have a corresponding call to the **EndPaint** function.

If the caret is in the area to be painted, **BeginPaint** automatically hides the caret to prevent it from being erased.

If the window's class has a background brush, **BeginPaint** uses that brush to erase the background of the update region before returning.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **EndPaint**, **InvalidateRect**, **InvalidateRgn**, **PAINTSTRUCT**, **ValidateRect**, **ValidateRgn**

DrawAnimatedRects

The **DrawAnimatedRects** function draws a wire-frame rectangle and animates it to indicate the opening of an icon or the minimizing or maximizing of a window.

```
BOOL WINAPI DrawAnimatedRects(
  HWND hwnd,             // handle to clipping window
  int idAni,             // type of animation
  CONST RECT *lprcFrom,  // rectangle coordinates (minimized)
  CONST RECT *lprcTo     // rectangle coordinates (restored)
);
```

Parameters

hwnd

[in] Handle to the window to which the rectangle is clipped. If this parameter is NULL, the working area of the screen is used.

idAni

[in] Specifies the type of animation. If you specify IDANI_CAPTION, the window caption will animate from the position specified by *lprcFrom* to the position specified by *lprcTo*. The effect is similar to minimizing or maximizing a window.

lprcFrom

[in] Pointer to a **RECT** structure specifying the location and size of the icon or minimized window. Coordinates are relative to the rectangle specified by the *lprcClip* parameter.

lprcTo

[in] Pointer to a **RECT** structure specifying the location and size of the restored window. Coordinates are relative to the rectangle specified by the *lprcClip* parameter.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **RECT**

DrawCaption

The **DrawCaption** function draws a window caption.

```
BOOL WINAPI DrawCaption(
  HWND hwnd,      // handle to window
  HDC hdc,        // handle to device context
  LPCRECT lprc,   // rectangle to draw into
  UINT uFlags     // drawing options
);
```

Parameters

hwnd

[in] Handle to a window that supplies text and an icon for the window caption.

hdc

[in] Handle to a device context. The function draws the window caption into this device context.

lprc

[in] Pointer to a **RECT** structure that specifies the bounding rectangle for the window caption.

uFlags

[in] Specifies drawing options. This parameter can be zero or more of the following values:

Value	Meaning
DC_ACTIVE	The function uses the colors that denote an active caption.
DC_GRADIENT	**Windows 98, Windows 2000:** When this flag is set, the function uses COLOR_GRADIENTACTIVECAPTION (if the DC_ACTIVE flag was set) or COLOR_GRADIENTINACTIVECAPTION for the title-bar color. If this flag is not set, the function uses COLOR_ACTIVECAPTION or COLOR_INACTIVECAPTION for both colors.
DC_ICON	The function draws the icon when drawing the caption text.
DC_INBUTTON	The function draws the caption as a button.
DC_SMALLCAP	The function draws a small caption, using the current small caption font.
DC_TEXT	The function draws the caption text when drawing the caption.

If DC_SMALLCAP is specified, the function draws a normal window caption.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **RECT**

DrawEdge

The **DrawEdge** function draws one or more edges of rectangle.

```
BOOL DrawEdge(
  HDC hdc,         // handle to device context
  LPRECT qrc,      // rectangle coordinates
  UINT edge,       // type of edge
  UINT grfFlags    // type of border
);
```

Parameters

hdc

[in] Handle to the device context.

qrc

[in/out] Pointer to a **RECT** structure that contains the logical coordinates of the rectangle.

edge

[in] Specifies the type of inner and outer edges to draw. This parameter must be a combination of one inner-border flag and one outer-border flag. The inner-border flags are as follows:

Value	Meaning
BDR_RAISEDINNER	Raised inner edge
BDR_SUNKENINNER	Sunken inner edge

The outer-border flags are as follows:

Value	Meaning
BDR_RAISEDOUTER	Raised outer edge
BDR_SUNKENOUTER	Sunken outer edge

Alternatively, the *edge* parameter can specify one of the following flags:

Value	Meaning
EDGE_BUMP	Combination of BDR_RAISEDOUTER and BDR_SUNKENINNER
EDGE_ETCHED	Combination of BDR_SUNKENOUTER and BDR_RAISEDINNER
EDGE_RAISED	Combination of BDR_RAISEDOUTER and BDR_RAISEDINNER
EDGE_SUNKEN	Combination of BDR_SUNKENOUTER and BDR_SUNKENINNER

grfFlags

[in] Specifies the type of border. This parameter can be a combination of the following values:

Value	Meaning
BF_ADJUST	Rectangle to be adjusted to leave space for client area.
BF_BOTTOM	Bottom of border rectangle.
BF_BOTTOMLEFT	Bottom and left side of border rectangle.
BF_BOTTOMRIGHT	Bottom and right side of border rectangle.
BF_DIAGONAL	Diagonal border.
BF_DIAGONAL_ENDBOTTOMLEFT	Diagonal border. The end point is the bottom-left corner of the rectangle; the origin is the top-right corner.
BF_DIAGONAL_ENDBOTTOMRIGHT	Diagonal border. The end point is the bottom-right corner of the rectangle; the origin is the top-left corner.
BF_DIAGONAL_ENDTOPLEFT	Diagonal border. The end point is the top-left corner of the rectangle; the origin is the bottom-right corner.
BF_DIAGONAL_ENDTOPRIGHT	Diagonal border. The end point is the top-right corner of the rectangle; the origin is the bottom-left corner.
BF_FLAT	Flat border.
BF_LEFT	Left side of border rectangle.
BF_MIDDLE	Interior of rectangle to be filled.
BF_MONO	One-dimensional border.
BF_RECT	Entire border rectangle.
BF_RIGHT	Right side of border rectangle.

(continued)

(continued)

Value	Meaning
BF_SOFT	Soft buttons instead of tiles.
BF_TOP	Top of border rectangle.
BF_TOPLEFT	Top and left side of border rectangle.
BF_TOPRIGHT	Top and right side of border rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **RECT**

DrawFocusRect

The **DrawFocusRect** function draws a rectangle in the style used to indicate that the rectangle has the focus.

```
BOOL DrawFocusRect(
  HDC hDC,          // handle to device context
  CONST RECT *lprc  // logical coordinates
);
```

Parameters

hDC
  [in] Handle to the device context.

lprc
  [in] Pointer to a **RECT** structure that specifies the logical coordinates of the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Because **DrawFocusRect** is an XOR function, calling it a second time with the same rectangle removes the rectangle from the screen.

This function draws a rectangle that cannot be scrolled. To scroll an area containing a rectangle drawn by this function, call **DrawFocusRect** to remove the rectangle from the screen, scroll the area, and then call **DrawFocusRect** again to draw the rectangle in the new position.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **FrameRect**, **RECT**

DrawFrameControl

The **DrawFrameControl** function draws a frame control of the specified type and style.

```
BOOL DrawFrameControl(
  HDC hdc,       // handle to device context
  LPRECT lprc,   // bounding rectangle
  UINT uType,    // frame-control type
  UINT uState    // frame-control state
);
```

Parameters

hdc

[in] Handle to the device context of the window in which to draw the control.

lprc

[in] Pointer to a **RECT** structure that contains the logical coordinates of the bounding rectangle for frame control.

uType

[in] Specifies the type of frame control to draw. This parameter can be one of the following values:

Value	Meaning
DFC_BUTTON	Standard button
DFC_CAPTION	Title bar
DCF_MENU	Menu bar
DFC_POPUPMENU	**Windows 98, Windows 2000:** Pop-up menu item
DFC_SCROLL	Scroll bar

uState

[in] Specifies the initial state of the frame control. If *uType* is DFC_BUTTON, *uState* can be one of the following values:

Value	Meaning
DFCS_BUTTON3STATE	Three-state button
DFCS_BUTTONCHECK	Check box
DFCS_BUTTONPUSH	Push button
DFCS_BUTTONRADIO	Radio button
DFCS_BUTTONRADIOIMAGE	Image for radio button (nonsquare needs image)
DFCS_BUTTONRADIOMASK	Mask for radio button (nonsquare needs mask)

If *uType* is DFC_CAPTION, *uState* can be one of the following values:

Value	Meaning
DFCS_CAPTIONCLOSE	**Close** button
DFCS_CAPTIONHELP	**Help** button
DFCS_CAPTIONMAX	**Maximize** button
DFCS_CAPTIONMIN	**Minimize** button
DFCS_CAPTIONRESTORE	**Restore** button

If *uType* is DFC_MENU, *uState* can be one of the following values:

Value	Meaning
DFCS_MENUARROW	Submenu arrow
DFCS_MENUARROWRIGHT	Submenu arrow pointing left. This is used for the right-to-left cascading menus used with right-to-left languages, such as Arabic or Hebrew
DFCS_MENUBULLET	Bullet
DFCS_MENUCHECK	Check mark

If *uType* is DFC_SCROLL, *uState* can be one of the following values:

Value	Meaning
DFCS_SCROLLCOMBOBOX	Combo-box scroll bar
DFCS_SCROLLDOWN	Down arrow of scroll bar
DFCS_SCROLLLEFT	Left arrow of scroll bar
DFCS_SCROLLRIGHT	Right arrow of scroll bar
DFCS_SCROLLSIZEGRIP	Size grip in bottom-right corner of window
DFCS_SCROLLSIZEGRIPRIGHT	Size grip in bottom-left corner of window. This is used with right-to-left languages such as Arabic or Hebrew
DFCS_SCROLLUP	Up arrow of scroll bar

The following style can be used to adjust the bounding rectangle of the push button:

Value	Meaning
DFCS_ADJUSTRECT	Bounding rectangle is adjusted to exclude the surrounding edge of the push button

One or more of the following values can be used to set the state of the control to be drawn:

Value	Meaning
DFCS_CHECKED	Button is checked
DFCS_FLAT	Button has a flat border
DFCS_HOT	**Windows 98, Windows 2000:** Button is hot-tracked
DFCS_INACTIVE	Button is inactive (appears dimmed)
DFCS_MONO	Button has a monochrome border
DFCS_PUSHED	Button is pushed
DFCS_TRANSPARENT	**Windows 98, Windows 2000:** The background remains untouched

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If *uType* is either DFC_MENU or DFC_BUTTON, and *uState* is not DFCS_BUTTONPUSH, the frame control is a black-on-white mask (that is, a black frame control on a white background). In such cases, the application must pass a handle to a bitmap memory device control. The application can then use the associated bitmap as the *hbmMask* parameter to the **MaskBlt** function, or it can use the device context as a parameter to the **BitBlt** function using ROPs, such as SRCAND and SRCINVERT.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **RECT**

DrawState

The **DrawState** function displays an image and applies a visual effect to indicate a state, such as a disabled or default state.

```
BOOL WINAPI DrawState(
  HDC hdc,                       // handle to device context
  HBRUSH hbr,                    // handle to brush
  DRAWSTATEPROC lpOutputFunc,    // callback function
  LPARAM lData,                  // image information
  WPARAM wData,                  // more image information
  int x,                         // horizontal location
  int y,                         // vertical location
  int cx,                        // image width
  int cy,                        // image height
  UINT fuFlags                   // image type and state
);
```

Parameters

hdc
[in] Handle to the device context in which to draw.

hbr
[in] Handle to the brush used to draw the image, if the state specified by the *fuFlags* parameter is DSS_MONO. This parameter is ignored for other states.

lpOutputFunc
[in] Pointer to an application-defined callback function used to render the image. This parameter is required if the image type in *fuFlags* is DST_COMPLEX. It is optional and can be NULL if the image type is DST_TEXT. For all other image types, this

parameter is ignored. For more information about the callback function, see the ***DrawStateProc*** function.

lData

[in] Specifies information about the image. The meaning of this parameter depends on the image type.

wData

[in] Specifies information about the image. The meaning of this parameter depends on the image type. It is, however, zero extended for use with the ***DrawStateProc*** function.

x

[in] Specifies the horizontal location at which to draw the image.

y

[in] Specifies the vertical location at which to draw the image.

cx

[in] Specifies the width of the image, in device units. This parameter is required if the image type is DST_COMPLEX. Otherwise, it can be zero to calculate the width of the image.

cy

[in] Specifies the height of the image, in device units. This parameter is required if the image type is DST_COMPLEX. Otherwise, it can be zero to calculate the height of the image.

fuFlags

[in] Specifies the image type and state. This parameter can be one of the following type values:

Value (type)	Meaning
DST_BITMAP	The image is a bitmap. The low-order word of the *lData* parameter is the bitmap handle.
DST_COMPLEX	The image is application defined. To render the image, **DrawState** calls the callback function specified by the *lpOutputFunc* parameter.
DST_ICON	The image is an icon. The low-order word of *lData* is the icon handle.
DST_PREFIXTEXT	The image is text that can contain an accelerator mnemonic. **DrawState** interprets the ampersand (&) prefix character as a directive to underscore the character that follows. The *lData* parameter is a pointer to the string, and the *wData* parameter specifies the length. If *wData* is zero, the string must be null-terminated.
DST_TEXT	The image is text. The *lData* parameter is a pointer to the string, and the *wData* parameter specifies the length. If *wData* is zero, the string must be null-terminated.

This parameter can be also one of the following state values:

Value (state)	Meaning
DSS_DISABLED	Embosses the image.
DSS_HIDEPREFIX	**Windows 2000:** Ignores the ampersand (&) prefix character in the text; thus, the letter that follows will not be underlined. This must be used with DST_PREFIXTEXT.
DSS_MONO	Draws the image using the brush specified by the *hbr* parameter.
DSS_NORMAL	Draws the image without any modification.
DSS_PREFIXONLY	**Windows 2000:** Draws only the underline at the position of the letter after the ampersand (&) prefix character. No text in the string is drawn. This must be used with DST_PREFIXTEXT.
DSS_RIGHT	Aligns the text to the right.
DSS_UNION	Dithers the image.

For all states, except DSS_NORMAL, the image is converted to monochrome before the visual effect is applied.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **DrawStateProc**

DrawStateProc

The **DrawStateProc** function is an application-defined callback function that renders a complex image for the **DrawState** function. The **DRAWSTATEPROC** type defines a pointer to this callback function. **DrawStateProc** is a placeholder for the application-defined function name.

```
BOOL CALLBACK DrawStateProc(
  HDC hdc,        // handle to device context
  LPARAM lData,   // image information
  WPARAM wData,   // more image information
  int cx,         // image width
  int cy          // image height
);
```

Parameters

hdc

[in] Handle to the device context to draw in. The device context is a memory device context with a bitmap selected, the dimensions of which are at least as great as those specified by the *cx* and *cy* parameters.

lData

[in] Specifies information about the image, which the application passed to **DrawState**.

wData

[in] Specifies information about the image, which the application passed to **DrawState**.

cx

[in] Specifies the image width, in device units, as specified by the call to **DrawState**.

cy

[in] Specifies the image height, in device units, as specified by the call to **DrawState**.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **DrawState**

EndPaint

The **EndPaint** function marks the end of painting in the specified window. This function is required for each call to the **BeginPaint** function, but only after painting is complete.

```
BOOL EndPaint(
  HWND hWnd,                 // handle to window
  CONST PAINTSTRUCT *lpPaint  // paint data
);
```

Parameters

hWnd

[in] Handle to the window that has been repainted.

lpPaint

[in] Pointer to a **PAINTSTRUCT** structure that contains the painting information retrieved by **BeginPaint**.

Return Values

The return value is always nonzero.

Remarks

If the caret was hidden by **BeginPaint**, **EndPaint** restores the caret to the screen.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **PAINTSTRUCT**

ExcludeUpdateRgn

The **ExcludeUpdateRgn** function prevents drawing within invalid areas of a window by excluding an updated region in the window from a clipping region.

```
int ExcludeUpdateRgn(
  HDC hDC,    // handle to device context
  HWND hWnd   // handle to window
);
```

Parameters

hDC

[in] Handle to the device context associated with the clipping region.

hWnd

[in] Handle to the window to update.

Return Values

The return value specifies the complexity of the excluded region; it can be any one of the following values:

Value	Meaning
COMPLEXREGION	Region consists of more than one rectangle.
ERROR	An error occurred.
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **GetUpdateRect**, **GetUpdateRgn**, **UpdateWindow**

GdiFlush

The **GdiFlush** function flushes the calling thread's current batch.

```
BOOL GdiFlush(VOID);
```

Parameters

This function has no parameters.

Return Values

If all functions in the current batch succeed, the return value is nonzero.

If not all functions in the current batch succeed, the return value is zero, indicating that at least one function returned an error.

Remarks

Batching enhances drawing performance by minimizing the amount of time needed to call GDI drawing functions that return Boolean values. The system accumulates the parameters for calls to these functions in the current batch, and then calls the functions when the batch is flushed by any of the following means:

- Calling the **GdiFlush** function.
- Reaching or exceeding the batch limit set by the **GdiSetBatchLimit** function.
- Filling the batching buffers.
- Calling any GDI function that does not return a Boolean value.

The return value for **GdiFlush** applies only to the functions in the batch at the time **GdiFlush** is called. Errors that occur when the batch is flushed by any other means are never reported.

The **GdiGetBatchLimit** function returns the batch limit.

Note The batch limit is maintained for each thread separately. In order to completely disable batching, call **GdiSetBatchLimit**(1) during the initialization of each thread.

An application should call **GdiFlush** before a thread goes away if there is a possibility that there are pending function calls in the graphics batch queue. The system does not execute such batched functions when a thread goes away.

A multithreaded application that serializes access to GDI objects with a mutex must ensure flushing the GDI batch queue by calling **GdiFlush** as each thread releases ownership of the GDI object. This prevents collisions of the GDI objects (device contexts, metafiles, and so on).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GdiGetBatchLimit**, **GdiSetBatchLimit**

GdiGetBatchLimit

The **GdiGetBatchLimit** function returns the maximum number of function calls that can be accumulated in the calling thread's current batch. The system flushes the current batch whenever this limit is exceeded.

```
DWORD GdiGetBatchLimit(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the batch limit.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The batch limit is set by using the **GdiSetBatchLimit** function. Setting the limit to 1 effectively disables batching.

Only GDI drawing functions that return Boolean values can be batched; calls to any other GDI functions immediately flush the current batch. Exceeding the batch limit or calling the **GdiFlush** function also flushes the current batch.

When the system batches a function call, the function returns TRUE. The actual return value for the function is reported only if **GdiFlush** is used to flush the batch.

Note The batch limit is maintained for each thread separately. In order to completely disable batching, call **GdiSetBatchLimit**(1) during the initialization of each thread.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GdiFlush**, **GdiSetBatchLimit**

GdiSetBatchLimit

The **GdiSetBatchLimit** function sets the maximum number of functions that can be accumulated in the calling thread's current batch. The system flushes the current batch whenever this limit is exceeded.

```
DWORD GdiSetBatchLimit(
  DWORD dwLimit   // batch limit
);
```

Parameters

dwLimit

[in] Specifies the batch limit to be set. A value of 0 sets the default limit. A value of 1 disables batching.

Return Values

If the function succeeds, the return value is the previous batch limit.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Only GDI drawing functions that return Boolean values can be accumulated in the current batch; calls to any other GDI functions immediately flush the current batch. Exceeding the batch limit or calling the **GdiFlush** function also flushes the current batch.

When the system accumulates a function, the function returns TRUE to indicate it is in the batch. When the system flushes the current batch and executes the function for the second time, the return value is either TRUE or FALSE, depending on whether the function succeeds. This second return value is reported only if **GdiFlush** is used to flush the batch.

Note The batch limit is maintained for each thread separately. In order to completely disable batching, call **GdiSetBatchLimit**(1) during the initialization of each thread.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GdiFlush**, **GdiGetBatchLimit**

GetBkColor

The **GetBkColor** function returns the current background color for the specified device context.

```
COLORREF GetBkColor(
  HDC hdc   // handle to device context
);
```

Parameters

hdc
  [in] Handle to the device context whose background color is to be returned.

Return Values

If the function succeeds, the return value is a **COLORREF** value for the current background color.

If the function fails, the return value is CLR_INVALID.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **COLORREF**, **GetBkMode**, **SetBkColor**

GetBkMode

The **GetBkMode** function returns the current background mix mode for a specified device context. The background mix mode of a device context affects text, hatched brushes, and pen styles that are not solid lines.

```
int GetBkMode(
  HDC hdc   // handle to device context
);
```

Parameters

hdc

[in] Handle to the device context whose background mode is to be returned.

Return Values

If the function succeeds, the return value specifies the current background mix mode, either OPAQUE or TRANSPARENT.

If the function fails, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GetBkColor**, **SetBkMode**

GetBoundsRect

The **GetBoundsRect** function obtains the current accumulated bounding rectangle for a specified device context.

The system maintains an accumulated bounding rectangle for each application. An application can retrieve and set this rectangle.

```
UINT GetBoundsRect(
  HDC hdc,              // handle to device context
  LPRECT lprcBounds,    // bounding rectangle
  UINT flags            // function options
);
```

Parameters

hdc

[in] Handle to the device context whose bounding rectangle the function will return.

lprcBounds

[out] Pointer to the **RECT** structure that will receive the current bounding rectangle. The application's rectangle is returned in logical coordinates, and the bounding rectangle is returned in screen coordinates.

flags

[in] Specifies how the **GetBoundsRect** function will behave. This parameter can be the following value:

Value	Meaning
DCB_RESET	Clears the bounding rectangle after returning it. If this flag is not set, the bounding rectangle will not be cleared.

Return Values

The return value specifies the state of the accumulated bounding rectangle; it can be one of the following values:

Value	Meaning
0	An error occurred. The specified device context handle is invalid.
DCB_DISABLE	Boundary accumulation is off.
DCB_ENABLE	Boundary accumulation is on.
DCB_RESET	The bounding rectangle is empty.
DCB_SET	The bounding rectangle is not empty.

Remarks

The DCB_SET value is a combination of the bit values DCB_ACCUMULATE and DCB_RESET. Applications that check the DCB_RESET bit to determine whether the bounding rectangle is empty must also check the DCB_ACCUMULATE bit. The bounding rectangle is empty only if the DCB_RESET bit is 1 and the DCB_ACCUMULATE bit is 0.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **SetBoundsRect**

GetROP2

The **GetROP2** function retrieves the foreground mix mode of the specified device context. The mix mode specifies how the pen or interior color and the color already on the screen are combined to yield a new color.

```
int GetROP2(
  HDC hdc   // handle to device context
);
```

Parameters

hdc
: [in] Handle to the device context.

Return Values

If the function succeeds, the return value specifies the foreground mix mode.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Following are the foreground mix modes:

Mix mode	Description
R2_BLACK	Pixel is always 0.
R2_COPYPEN	Pixel is the pen color.
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the screen and the inverse of the pen.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the screen.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the screen.
R2_MERGENOTPEN	Pixel is a combination of the screen color and the inverse of the pen color.
R2_MERGEPEN	Pixel is a combination of the pen color and the screen color.
R2_MERGEPENNOT	Pixel is a combination of the pen color and the inverse of the screen color.
R2_NOP	Pixel remains unchanged.
R2_NOT	Pixel is the inverse of the screen color.
R2_NOTCOPYPEN	Pixel is the inverse of the pen color.
R2_NOTMASKPEN	Pixel is the inverse of the R2_MASKPEN color.
R2_NOTMERGEPEN	Pixel is the inverse of the R2_MERGEPEN color.
R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color.
R2_WHITE	Pixel is always 1.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the screen, but not in both.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **SetROP2**

GetUpdateRect

The **GetUpdateRect** function retrieves the coordinates of the smallest rectangle that completely encloses the update region of the specified window. If the window was created with the CS_OWNDC style and the mapping mode is not MM_TEXT, **GetUpdateRect** retrieves the rectangle in logical coordinates. Otherwise, it retrieves the rectangle in client coordinates. If there is no update region, **GetUpdateRect** retrieves an empty rectangle (sets all coordinates to zero).

```
BOOL GetUpdateRect(
  HWND hWnd,      // handle to window
  LPRECT lpRect,  // update rectangle coordinates
  BOOL bErase     // erase state
);
```

Parameters

hWnd

[in] Handle to the window with an update region that is to be retrieved.

lpRect

[out] Pointer to the **RECT** structure that receives the coordinates of the enclosing rectangle.

An application can set this parameter to NULL to determine whether an update region exists for the window. If this parameter is NULL, **GetUpdateRect** returns nonzero if an update region exists, and zero if one does not. This provides a simple and efficient means of determining whether a **WM_PAINT** message resulted from an invalid area.

bErase

[in] Specifies whether the background in the update region is to be erased. If this parameter is TRUE and the update region is not empty, **GetUpdateRect** sends a **WM_ERASEBKGND** message to the specified window to erase the background.

Return Values

If the update region is not empty, the return value is nonzero.

If there is no update region, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The update rectangle retrieved by the **BeginPaint** function is identical to that retrieved by **GetUpdateRect**.

BeginPaint automatically validates the update region, so any call to **GetUpdateRect** made immediately after the call to **BeginPaint** retrieves an empty update region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **GetUpdateRgn**, **InvalidateRect**, **RECT**, **UpdateWindow**, **ValidateRect**

GetUpdateRgn

The **GetUpdateRgn** function retrieves the update region of a window by copying it into the specified region. The coordinates of the update region are relative to the upper-left corner of the window (that is, they are client coordinates).

```
int GetUpdateRgn(
  HWND hWnd,     // handle to window
  HRGN hRgn,     // handle to region
  BOOL bErase    // erase state
);
```

Parameters

hWnd
[in] Handle to the window with an update region that is to be retrieved.

hRgn
[in] Handle to the region to receive the update region.

bErase
[in] Specifies whether the window background should be erased and whether nonclient areas of child windows should be drawn. If this parameter is FALSE, no drawing is done.

Return Values

The return value indicates the complexity of the resulting region; it can be one of the following values:

Value	Meaning
COMPLEXREGION	Region consists of more than one rectangle.
ERROR	An error occurred.
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.

Remarks

The **BeginPaint** function automatically validates the update region, so any call to **GetUpdateRgn** made immediately after the call to **BeginPaint** retrieves an empty update region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions

GetWindowDC

The **GetWindowDC** function retrieves the device context (DC) for the entire window, including title bar, menus, and scroll bars. A window device context permits painting anywhere in a window, because the origin of the device context is the upper-left corner of the window instead of the client area.

GetWindowDC assigns default attributes to the window device context each time it retrieves the device context. Previous attributes are lost.

```
HDC GetWindowDC(
  HWND hWnd   // handle to window
);
```

Parameters

hWnd

[in] Handle to the window with a device context that is to be retrieved. If this value is NULL, **GetWindowDC** retrieves the device context for the entire screen.

Windows 98, Windows 2000: If this parameter is NULL, **GetWindowDC** retrieves the device context for the primary display monitor. To get the device context for other display monitors, use the **EnumDisplayMonitors** and CreateDC functions.

Return Values

If the function succeeds, the return value is a handle to a device context for the specified window.

If the function fails, the return value is NULL, indicating an error or an invalid *hWnd* parameter.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

GetWindowDC is intended for special painting effects within a window's nonclient area. Painting in nonclient areas of any window is not recommended.

The **GetSystemMetrics** function can be used to retrieve the dimensions of various parts of the nonclient area, such as the title bar, menu, and scroll bars.

The **GetDC** function can be used to retrieve a device context for the entire screen.

After painting is complete, the **ReleaseDC** function must be called to release the device context. Not releasing the window device context has serious effects on painting requested by applications.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **GetDC**, **GetSystemMetrics**, **ReleaseDC**

GetWindowRgn

The **GetWindowRgn** function obtains a copy of the window region of a window. The window region of a window is set by calling the **SetWindowRgn** function. The window region determines the area within the window where the system permits drawing. The system does not display any portion of a window that lies outside of the window region

```
int GetWindowRgn(
  HWND hWnd, // handle to window
  HRGN hRgn  // handle to window region
);
```

Parameters

hWnd

[in] Handle to the window whose window region is to be obtained.

hRgn

[in] Receives a handle to the window region.

Return Values

The return value specifies the type of the region that the function obtains. It can be one of the following values:

Value	Meaning
NULLREGION	The region is empty.
SIMPLEREGION	The region is a single rectangle.
COMPLEXREGION	The region is more than one rectangle.
ERROR	An error occurred; the region is unaffected.

Remarks

The coordinates of a window's window region are relative to the upper-left corner of the window, not the client area of the window.

To set the window region of a window, call the **SetWindowRgn** function.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **SetWindowRgn**

GrayString

The **GrayString** function draws shaded text at the specified location. The function draws the text by copying it into a memory bitmap, shading the bitmap, and then copying the bitmap to the screen. The function dims the text regardless of the selected brush and background. **GrayString** uses the font currently selected for the specified device context.

If the *lpOutputFunc* parameter is NULL, GDI uses the **TextOut** function, and the *lpData* parameter is assumed to be a pointer to the character string to be output. If the characters to be output cannot be handled by **TextOut** (for example, the string is stored as a bitmap), the application must supply its own output function.

```
BOOL GrayString(
  HDC hDC,                        // handle to DC
  HBRUSH hBrush,                  // handle to the brush
  GRAYSTRINGPROC lpOutputFunc,    // callback function
  LPARAM lpData,                  // application-defined data
  int nCount,                     // number of characters
  int X,                          // horizontal position
  int Y,                          // vertical position
  int nWidth,                     // width
  int nHeight                     // height
);
```

Parameters

hDC
[in] Handle to the device context.

hBrush
[in] Handle to the brush to be used for dimming. If this parameter is NULL, the text is dimmed with the same brush that was used to draw window text.

lpOutputFunc
[in] Pointer to the application-defined function that will draw the string, or, if **TextOut** is to be used to draw the string, it is a NULL pointer. For details, see the ***OutputProc*** callback function.

lpData
[in] Specifies a pointer to data to be passed to the output function. If the *lpOutputFunc* parameter is NULL, *lpData* must be a pointer to the string to be output.

nCount

[in] Specifies the number of characters to be output. If the *nCount* parameter is zero, **GrayString** calculates the length of the string (assuming *lpData* is a pointer to the string). If *nCount* is –1 and the function pointed to by *lpOutputFunc* returns FALSE, the image is shown but does not appear dimmed.

X

[in] Specifies the device x-coordinate of the starting position of the rectangle that encloses the string.

Y

[in] Specifies the device y-coordinate of the starting position of the rectangle that encloses the string.

nWidth

[in] Specifies the width, in device units, of the rectangle that encloses the string. If this parameter is zero, **GrayString** calculates the width of the area, assuming *lpData* is a pointer to the string.

nHeight

[in] Specifies the height, in device units, of the rectangle that encloses the string. If this parameter is zero, **GrayString** calculates the height of the area, assuming *lpData* is a pointer to the string.

Return Values

If the string is drawn, the return value is nonzero.

If either the **TextOut** function or the application-defined output function returned zero, or there was insufficient memory to create a memory bitmap for dimming, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Without calling **GrayString**, an application can draw dimmed strings on devices that support a solid gray color. The system color COLOR_GRAYTEXT is the solid-gray system color used to draw disabled text. The application can call the **GetSysColor** function to retrieve the color value of COLOR_GRAYTEXT. If the color is other than zero (black), the application can call the **SetTextColor** function to set the text color to the color value and, then, draw the string directly. If the retrieved color is black, the application must call **GrayString** to shade the text.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **DrawText**, **GetSysColor**, **OutputProc**, **SetTextColor**, **TabbedTextOut**, **TextOut**

InvalidateRect

The **InvalidateRect** function adds a rectangle to the specified window's update region. The update region represents the portion of the window's client area that must be redrawn.

```
BOOL InvalidateRect(
  HWND hWnd,           // handle to window
  CONST RECT *lpRect,  // rectangle coordinates
  BOOL bErase          // erase state
);
```

Parameters

hWnd

[in] Handle to the window whose update region has changed. If this parameter is NULL, the system invalidates and redraws all windows, and sends the **WM_ERASEBKGND** and **WM_NCPAINT** messages to the window procedure before the function returns.

lpRect

[in] Pointer to a **RECT** structure that contains the client coordinates of the rectangle to be added to the update region. If this parameter is NULL, the entire client area is added to the update region.

bErase

[in] Specifies whether the background within the update region is to be erased when the update region is processed. If this parameter is TRUE, the background is erased when the **BeginPaint** function is called. If this parameter is FALSE, the background remains unchanged.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The invalidated areas accumulate in the update region until the region is processed when the next **WM_PAINT** message occurs or until the region is validated by using the **ValidateRect** or **ValidateRgn** function.

The system sends a **WM_PAINT** message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

If the *bErase* parameter is TRUE for any part of the update region, the background is erased in the entire region, not just in the specified part.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **InvalidateRgn**, **RECT**, **ValidateRect**, **ValidateRgn**, **WM_ERASEBKGND**, **WM_NCPAINT**, **WM_PAINT**

InvalidateRgn

The **InvalidateRgn** function invalidates the client area within the specified region by adding it to the current update region of a window. The invalidated region, along with all other areas in the update region, is marked for painting when the next **WM_PAINT** message occurs.

```
BOOL InvalidateRgn(
  HWND hWnd,      // handle to window
  HRGN hRgn,      // handle to region
  BOOL bErase     // erase state
);
```

Parameters

hWnd

[in] Handle to the window with an update region that is to be modified.

hRgn

[in] Handle to the region to be added to the update region. The region is assumed to have client coordinates. If this parameter is NULL, the entire client area is added to the update region.

bErase

[in] Specifies whether the background within the update region should be erased when the update region is processed. If this parameter is TRUE, the background is erased when the **BeginPaint** function is called. If the parameter is FALSE, the background remains unchanged.

Return Values

The return value is always nonzero.

Remarks

Invalidated areas accumulate in the update region until the next **WM_PAINT** message is processed, or until the region is validated by using the **ValidateRect** or **ValidateRgn** function.

The system sends a **WM_PAINT** message to a window whenever its update region is not empty and there are no other messages in the application queue for that window.

The specified region must have been created by using one of the region functions.

If the *bErase* parameter is TRUE for any part of the update region, the background in the entire region is erased, not just in the specified part.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **InvalidateRect**, **ValidateRect**, **ValidateRgn**, **WM_PAINT**

LockWindowUpdate

The **LockWindowUpdate** function disables or re-enables drawing in the specified window. Only one window can be locked at a time.

```
BOOL LockWindowUpdate(
  HWND hWndLock   // handle to window
);
```

Parameters

hWndLock

[in] Specifies the window in which drawing will be disabled. If this parameter is NULL, drawing in the locked window is enabled.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero, indicating that an error occurred or another window was already locked.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If an application with a locked window (or any locked child windows) calls the **GetDC**, **GetDCEx**, or **BeginPaint** function, the called function returns a device context with a visible region that is empty. This will occur until the application unlocks the window by calling **LockWindowUpdate**, specifying a value of NULL for *hWndLock*.

If an application attempts to draw within a locked window, the system records the extent of the attempted operation in a bounding rectangle. When the window is unlocked, the system invalidates the area within this bounding rectangle, forcing an eventual **WM_PAINT** message to be sent to the previously locked window and its child windows. If no drawing has occurred while the window updates were locked, no area is invalidated.

LockWindowUpdate does not make the specified window invisible or clear the WS_VISIBLE style bit.

A locked window cannot be moved.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **GetDC**, **GetDCEx**, **WM_PAINT**

OutputProc

The **OutputProc** function is an application-defined callback function used with the **GrayString** function. It is used to draw a string. The **GRAYSTRINGPROC** type defines a pointer to this callback function. **OutputProc** is a placeholder for the application-defined or library-defined function name.

```
BOOL CALLBACK OutputProc(
  HDC hdc,          // handle to DC
  LPARAM lpData,    // string
  int cchData       // length of string
);
```

Parameters

hdc

[in] Handle to a device context with a bitmap of at least the width and height specified by the *nWidth* and *nHeight* parameters passed to **GrayString**.

lpData

[in] Pointer to the string to be drawn.

cchData

[in] Specifies the length, in characters, of the string.

Return Values

If it succeeds, the callback function should return TRUE.

If the function fails, the return value is FALSE.

Remarks

The callback function must draw an image relative to the coordinates (0,0).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GrayString**

PaintDesktop

The **PaintDesktop** function fills the clipping region in the specified device context with the desktop pattern or wallpaper. The function is provided primarily for shell desktops.

```
BOOL WINAPI PaintDesktop(
  HDC hdc // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions

RedrawWindow

The **RedrawWindow** function updates the specified rectangle or region in a window's client area.

```
BOOL RedrawWindow(
  HWND hWnd,                // handle to window
  CONST RECT *lprcUpdate,   // update rectangle
  HRGN hrgnUpdate,          // handle to update region
  UINT flags                // array of redraw flags
);
```

Parameters

hWnd

[in] Handle to the window to be redrawn. If this parameter is NULL, the desktop window is updated.

lprcUpdate

[in] Pointer to a **RECT** structure containing the coordinates of the update rectangle. This parameter is ignored if the *hrgnUpdate* parameter identifies a region.

hrgnUpdate

[in] Handle to the update region. If both the *hrgnUpdate* and *lprcUpdate* parameters are NULL, the entire client area is added to the update region.

flags

[in] Specifies one or more redraw flags. This parameter can be used to invalidate or validate a window, control repainting, and control which windows are affected by **RedrawWindow**.

The following flags are used to invalidate the window:

Flag (invalidation)	Description
RDW_ERASE	Causes the window to receive a **WM_ERASEBKGND** message when the window is repainted. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_ERASE has no effect.
RDW_FRAME	Causes any part of the nonclient area of the window that intersects the update region to receive a **WM_NCPAINT** message. The RDW_INVALIDATE flag must also be specified; otherwise, RDW_FRAME has no effect. The **WM_NCPAINT** message typically is not sent during the execution of **RedrawWindow** unless either RDW_UPDATENOW or RDW_ERASENOW is specified.
RDW_INTERNALPAINT	Causes a **WM_PAINT** message to be posted to the window regardless of whether any portion of the window is invalid.
RDW_INVALIDATE	Invalidates *lprcUpdate* or *hrgnUpdate* (only one may be non-NULL). If both are NULL, the entire window is invalidated.

The following flags are used to validate the window:

Flag (validation)	Description
RDW_NOERASE	Suppresses any pending **WM_ERASEBKGND** messages.

RDW_NOFRAME	Suppresses any pending **WM_NCPAINT** messages. This flag must be used with RDW_VALIDATE and is typically used with RDW_NOCHILDREN. RDW_NOFRAME should be used with care, as it could cause parts of a window to be painted improperly.
RDW_NOINTERNALPAINT	Suppresses any pending internal **WM_PAINT** messages. This flag does not affect **WM_PAINT** messages resulting from a non-NULL update area.
RDW_VALIDATE	Validates *lprcUpdate* or *hrgnUpdate* (only one may be non-NULL). If both are NULL, the entire window is validated. This flag does not affect internal **WM_PAINT** messages.

The following flags control when repainting occurs. **RedrawWindow** will not repaint unless one of these flags is specified:

Flag	**Description**
RDW_ERASENOW	Causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive **WM_NCPAINT** and **WM_ERASEBKGND** messages, if necessary, before the function returns. **WM_PAINT** messages are received at the ordinary time.
RDW_UPDATENOW	Causes the affected windows (as specified by the RDW_ALLCHILDREN and RDW_NOCHILDREN flags) to receive **WM_NCPAINT**, **WM_ERASEBKGND**, and **WM_PAINT** messages, if necessary, before the function returns.

By default, the windows affected by **RedrawWindow** depend on whether the specified window has the WS_CLIPCHILDREN style. Child windows that are not the WS_CLIPCHILDREN style are unaffected; non-WS_CLIPCHILDREN windows are validated or invalidated recursively until a WS_CLIPCHILDREN window is encountered. The following flags control which windows are affected by the **RedrawWindow** function:

Flag	**Description**
RDW_ALLCHILDREN	Includes child windows, if any, in the repainting operation.
RDW_NOCHILDREN	Excludes child windows, if any, from the repainting operation.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When **RedrawWindow** is used to invalidate part of the desktop window, the desktop window does not receive a **WM_PAINT** message. To repaint the desktop, an application uses the RDW_ERASE flag to generate a **WM_ERASEBKGND** message.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GetUpdateRect**, **GetUpdateRgn**, **InvalidateRect**, **InvalidateRgn**, **RECT**, **UpdateWindow**

SetBkColor

The **SetBkColor** function sets the current background color to the specified color value, or to the nearest physical color if the device cannot represent the specified color value.

```
COLORREF SetBkColor(
  HDC hdc,           // handle to DC
  COLORREF crColor   // background-color value
);
```

Parameters

hdc

[in] Handle to the device context.

crColor

[in] Specifies the new background color. To make a **COLORREF** value, use the RGB macro.

Return Values

If the function succeeds, the return value specifies the previous background color as a **COLORREF** value.

If the function fails, the return value is CLR_INVALID.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

This function fills the gaps between styled lines drawn using a pen created by the **CreatePen** function; it does not fill the gaps between styled lines drawn using a pen created by the **ExtCreatePen** function. The **SetBKColor** function also sets the background colors for **TextOut** and **ExtTextOut**.

If the background mode is OPAQUE, the background color is used to fill gaps between styled lines, gaps between hatched lines in brushes, and character cells. The background color is used also when converting bitmaps from color to monochrome, and vice versa.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **COLORREF**, **CreatePen**, **ExtCreatePen**, **GetBKColor**, **GetBkMode**, **SetBkMode**

SetBkMode

The **SetBkMode** function sets the background mix mode of the specified device context. The background mix mode is used with text, hatched brushes, and pen styles that are not solid lines.

```
int SetBkMode(
  HDC hdc,        // handle to DC
  int iBkMode     // background mode
);
```

Parameters

hdc
[in] Handle to the device context.

iBkMode
[in] Specifies the background mode. This parameter can be one of the following values:

Value	Description
OPAQUE	Background is filled with the current background color before the text, hatched brush, or pen is drawn.
TRANSPARENT	Background remains untouched.

Return Values

If the function succeeds, the return value specifies the previous background mode.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **SetBkMode** function affects the line styles for lines drawn using a pen created by the **CreatePen** function. **SetBkMode** does not affect lines drawn using a pen created by the **ExtCreatePen** function.

The *iBkMode* parameter can also be set to driver-specific values. GDI passes such values to the device driver and, otherwise, ignores them.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **CreatePen**, **ExtCreatePen**, **GetBkMode**

SetBoundsRect

The **SetBoundsRect** function controls the accumulation of bounding rectangle information for the specified device context. The system can maintain a bounding rectangle for all drawing operations. An application can examine and set this rectangle. The drawing boundaries are useful for invalidating bitmap caches.

```
UINT SetBoundsRect(
  HDC hdc,                 // handle to DC
  CONST RECT *lprcBounds,  // bounding rectangle
  UINT flags               // rectangle combination option
);
```

Parameters

hdc

[in] Handle to the device context for which to accumulate bounding rectangles.

lprcBounds

[in] Pointer to a **RECT** structure used to set the bounding rectangle. Rectangle dimensions are in logical coordinates. This parameter can be NULL.

flags

[in] Specifies how the new rectangle will be combined with the accumulated rectangle. This parameter can be one of more of the following values:

Value	Description
DCB_ACCUMULATE	Adds the rectangle specified by the *lprcBounds* parameter to the bounding rectangle (using a rectangle union operation). Using both DCB_RESET and DCB_ACCUMULATE sets the bounding rectangle to the rectangle specified by the *lprcBounds* parameter.
DCB_DISABLE	Turns off boundary accumulation.
DCB_ENABLE	Turns on boundary accumulation, which is disabled by default.
DCB_RESET	Clears the bounding rectangle.

Return Values

If the function succeeds, the return value specifies the previous state of the bounding rectangle. This state can be a combination of the following values:

Value	Meaning
DCB_DISABLE	Boundary accumulation is off.
DCB_ENABLE	Boundary accumulation is on. DCB_ENABLE and DCB_DISABLE are mutually exclusive.
DCB_RESET	Bounding rectangle is empty.
DCB_SET	Bounding rectangle is not empty. DCB_SET and DCB_RESET are mutually exclusive.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The DCB_SET value is a combination of the bit values DCB_ACCUMULATE and DCB_RESET. Applications that check the DCB_RESET bit to determine whether the bounding rectangle is empty also must check the DCB_ACCUMULATE bit. The bounding rectangle is empty only if the DCB_RESET bit is 1 and the DCB_ACCUMULATE bit is 0.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GetBoundsRect**, **RECT**

SetROP2

The **SetROP2** function sets the current foreground mix mode. GDI uses the foreground mix mode to combine pens and interiors of filled objects with the colors already on the screen. The foreground mix mode defines how colors from the brush or pen and the colors in the existing image are to be combined.

```
int SetROP2(
  HDC hdc,         // handle to DC
  int fnDrawMode   // drawing mode
);
```

Parameters

hdc

[in] Handle to the device context.

fnDrawMode

[in] Specifies the mix mode. This parameter can be one of the following values:

Mix mode	Description
R2_BLACK	Pixel is always 0.
R2_COPYPEN	Pixel is the pen color.
R2_MASKNOTPEN	Pixel is a combination of the colors common to both the screen and the inverse of the pen.
R2_MASKPEN	Pixel is a combination of the colors common to both the pen and the screen.
R2_MASKPENNOT	Pixel is a combination of the colors common to both the pen and the inverse of the screen.
R2_MERGENOTPEN	Pixel is a combination of the screen color and the inverse of the pen color.

R2_MERGEPEN	Pixel is a combination of the pen color and the screen color.
R2_MERGEPENNOT	Pixel is a combination of the pen color and the inverse of the screen color.
R2_NOP	Pixel remains unchanged.
R2_NOT	Pixel is the inverse of the screen color.
R2_NOTCOPYPEN	Pixel is the inverse of the pen color.
R2_NOTMASKPEN	Pixel is the inverse of the R2_MASKPEN color.
R2_NOTMERGEPEN	Pixel is the inverse of the R2_MERGEPEN color.
R2_NOTXORPEN	Pixel is the inverse of the R2_XORPEN color.
R2_WHITE	Pixel is always 1.
R2_XORPEN	Pixel is a combination of the colors in the pen and in the screen, but not in both.

Return Values

If the function succeeds, the return value specifies the previous mix mode.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

Mix modes define how GDI combines source and destination colors when drawing with the current pen. The mix modes are binary raster operation codes, representing all possible Boolean functions of two variables, using the binary operations AND, OR, and XOR (exclusive OR), and the unary operation NOT. The mix mode is for raster devices only; it is not available for vector devices.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GetROP2**

SetWindowRgn

The **SetWindowRgn** function sets the window region of a window. The window region determines the area within the window where the system permits drawing. The system does not display any portion of a window that lies outside of the window region.

```
int SetWindowRgn(
  HWND hWnd,      // handle to window
  HRGN hRgn,      // handle to region
  BOOL bRedraw    // window redraw option
);
```

Parameters

hWnd

[in] Handle to the window whose window region is to be set.

hRgn

[in] Handle to a region. The function sets the window region of the window to this region.

If *hRgn* is NULL, the function sets the window region to NULL.

bRedraw

[in] Specifies whether the system redraws the window after setting the window region. If *bRedraw* is TRUE, the system does so; otherwise, it does not.

Typically, you set *bRedraw* to TRUE if the window is visible.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

When this function is called, the system sends the **WM_WINDOWPOSCHANGING** and **WM_WINDOWPOSCHANGED** messages to the window.

The coordinates of a window's window region are relative to the upper-left corner of the window, not the client area of the window.

After a successful call to **SetWindowRgn**, the system owns the region specified by the region handle *hRgn*. The system does not make a copy of the region. Thus, you should not make any further function calls with this region handle. In particular, do not delete this region handle. The system deletes the region handle when it no longer needed.

To obtain the window region of a window, call the **GetWindowRgn** function.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GetWindowRgn**, **WM_WINDOWPOSCHANGING**

UpdateWindow

The **UpdateWindow** function updates the client area of the specified window by sending a **WM_PAINT** message to the window if the window's update region is not empty. The function sends a **WM_PAINT** message directly to the window procedure of the specified window, bypassing the application queue. If the update region is empty, no message is sent.

```
BOOL UpdateWindow(
  HWND hWnd   // handle to window
);
```

Parameters

hWnd

[in] Handle to the window to be updated.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **ExcludeUpdateRgn**, **GetUpdateRect**, **GetUpdateRgn**, **InvalidateRect**, **InvalidateRgn**, **WM_PAINT**

ValidateRect

The **ValidateRect** function validates the client area within a rectangle by removing the rectangle from the update region of the specified window.

```
BOOL ValidateRect(
  HWND hWnd,          // handle to window
  CONST RECT *lpRect  // validation rectangle coordinates
);
```

Parameters

hWnd

[in] Handle to the window whose update region is to be modified. If this parameter is NULL, the system invalidates and redraws all windows and sends the **WM_ERASEBKGND** and **WM_NCPAINT** messages to the window procedure before the function returns.

lpRect

[in] Pointer to a **RECT** structure that contains the client coordinates of the rectangle to be removed from the update region. If this parameter is NULL, the entire client area is removed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/ 2000: To get extended error information, call **GetLastError**.

Remarks

The **BeginPaint** function automatically validates the entire client area. Neither the **ValidateRect** nor the **ValidateRgn** function should be called if a portion of the update region must be validated before the next **WM_PAINT** message is generated.

The system continues to generate **WM_PAINT** messages until the current update region is validated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **InvalidateRect**, **InvalidateRgn**, **RECT**, **ValidateRgn**, **WM_PAINT**

ValidateRgn

The **ValidateRgn** function validates the client area within a region by removing the region from the current update region of the specified window.

```
BOOL ValidateRgn(
  HWND hWnd,  // handle to window
  HRGN hRgn   // handle to region
);
```

Parameters

hWnd

[in] Handle to the window whose update region is to be modified.

hRgn

[in] Handle to a region that defines the area to be removed from the update region. If this parameter is NULL, the entire client area is removed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The specified region must have been created by a region function. The region coordinates are assumed to be client coordinates.

The **BeginPaint** function automatically validates the entire client area. Neither the **ValidateRect** nor the **ValidateRgn** function should be called if a portion of the update region must be validated before the next **WM_PAINT** message is generated.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **BeginPaint**, **ExcludeUpdateRgn**, **InvalidateRect**, **InvalidateRgn**, **ValidateRect**, **WM_PAINT**

WindowFromDC

The **WindowFromDC** function returns a handle to the window associated with the specified display device context (DC). Output functions that use the specified device context draw into this window.

```
HWND WindowFromDC(
  HDC hDC   // handle to window
);
```

Parameters

hDC

[in] Handle to the device context from which a handle for the associated window is to be retrieved.

Return Values

The return value is a handle to the window associated with the specified DC. If no window is associated with the specified DC, the return value is NULL.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Painting and Drawing Overview, Painting and Drawing Functions, **GetDC**, **GetDCEx**, **GetWindowDC**

Painting and Drawing Structures

PAINTSTRUCT

The **PAINTSTRUCT** structure contains information for an application. This information can be used to paint the client area of a window owned by that application.

```
typedef struct tagPAINTSTRUCT {
  HDC  hdc;
  BOOL fErase;
  RECT rcPaint;
  BOOL fRestore;
  BOOL fIncUpdate;
  BYTE rgbReserved[32];
} PAINTSTRUCT, *PPAINTSTRUCT;
```

Members

hdc
Handle to the display DC to be used for painting.

fErase
Specifies whether the background must be erased. This value is nonzero if the application should erase the background. The application is responsible for erasing the background if a window class is created without a background brush. For more information, see the description of the **hbrBackground** member of the ***WNDCLASS*** structure.

rcPaint
Specifies a **RECT** structure that specifies the upper-left and lower-right corners of the rectangle in which the painting is requested.

fRestore
Reserved; used internally by the system.

fIncUpdate
Reserved; used internally by the system.

rgbReserved
Reserved; used internally by the system.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Structures, **BeginPaint**, **RECT**, **WNDCLASS**

Painting and Drawing Messages

WM_DISPLAYCHANGE

The **WM_DISPLAYCHANGE** message is sent to all windows when the display resolution has changed.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_DISPLAYCHANGE
  WPARAM wParam,   // image depth
  LPARAM lParam    // screen resolution
);
```

Parameters

wParam

Specifies the new image depth of the display, in bits per pixel.

lParam

The low-order word specifies the horizontal resolution of the screen.

The high-order word specifies the vertical resolution of the screen.

Remarks

This message is sent only to top-level windows. For all other windows, it is posted.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Messages, **HIWORD**, **LOWORD**

WM_NCPAINT

The **WM_NCPAINT** message is sent to a window when its frame must be painted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_NCPAINT
  WPARAM wParam,   // handle to update region (HRGN)
  LPARAM lParam    // not used
);
```

Parameters

wParam

Handle to the update region of the window. The update region is clipped to the window frame. When *wParam* is 1, the entire window frame needs to be updated.

lParam

This parameter is not used.

Return Values

An application returns zero if it processes this message.

Remarks

The **DefWindowProc** function paints the window frame.

An application can intercept the **WM_NCPAINT** message and paint its own custom window frame. The clipping region for a window is always rectangular, even if the shape of the frame is altered.

The *wParam* value can be passed to **GetDCEx**, as in the following example:

```
case WM_NCPAINT:
{
    HDC hdc;
    hdc = GetDCEx(hwnd, (HRGN)wParam, DCX_WINDOW|DCX_INTERSECTRGN);
    // Paint into this DC
    ReleaseDC(hwnd, hdc);
}
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Messages, **DefWindowProc**, **GetDCEx, GetWindowDC**, **WM_PAINT**

WM_PAINT

The **WM_PAINT** message is sent when the system or another application makes a request to paint a portion of an application's window. The message is sent when the **UpdateWindow** or **RedrawWindow** function is called, or by the **DispatchMessage** function when the application obtains a **WM_PAINT** message by using the **GetMessage** or **PeekMessage** function.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_PAINT
  WPARAM wParam,   // handle to DC (HDC)
  LPARAM lParam    // not used
);
```

Parameters

wParam

Handle to the device context to draw in. If this parameter is NULL, use the default device context. This parameter is used by some common controls to enable drawing in a device context other than the default device context. Other windows can ignore this parameter safely.

lParam

This parameter is not used.

Return Values

An application returns zero if it processes this message.

Remarks

The **DefWindowProc** function validates the update region. The function may also send the **WM_NCPAINT** message to the window procedure if the window frame must be painted and send the **WM_ERASEBKGND** message if the window background must be erased.

The system sends this message when there are no other messages in the application's message queue. **DispatchMessage** determines where to send the message; **GetMessage** determines which message to dispatch. **GetMessage** returns the **WM_PAINT** message when there are no other messages in the application's message queue, and **DispatchMessage** sends the message to the appropriate window procedure.

A window may receive internal paint messages as a result of calling **RedrawWindow** with the RDW_INTERNALPAINT flag set. In this case, the window may not have an update region. An application should call the **GetUpdateRect** function to determine whether the window has an update region. If **GetUpdateRect** returns zero, the application should not call the **BeginPaint** and **EndPaint** functions.

An application must check for any necessary internal painting by looking at its internal data structures for each **WM_PAINT** message, because a **WM_PAINT** message might have been caused by both a non-NULL update region and a call to **RedrawWindow** with the RDW_INTERNALPAINT flag set.

The system sends an internal **WM_PAINT** message only once. After an internal **WM_PAINT** message is returned from **GetMessage** or **PeekMessage** or is sent to a window by **UpdateWindow**, the system does not post or send further **WM_PAINT** messages until the window is invalidated or until **RedrawWindow** is called again with the RDW_INTERNALPAINT flag set.

For some common controls, the default **WM_PAINT** message processing checks the *wParam* parameter. If *wParam* is non-NULL, the control assumes that the value is an HDC and paints using that device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.

+ See Also

Painting and Drawing Overview, Painting and Drawing Messages, **BeginPaint**, **DefWindowProc**, **DispatchMessage**, **EndPaint**, **GetMessage**, **GetUpdateRect**, **PeekMessage**, **RedrawWindow**, **UpdateWindow**, **WM_ERASEBKGND**, **WM_NCPAINT**

WM_PRINT

The **WM_PRINT** message is sent to a window to request that it draw itself in the specified device context, most commonly in a printer device context.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_PRINT
  WPARAM wParam,   // handle to DC (HDC)
  LPARAM lParam    // drawing options
);
```

Parameters

wParam

Handle to the device context in which to draw.

lParam

Specifies the drawing options. This parameter can be one or more of the following values:

Value	Meaning
PRF_CHECKVISIBLE	Draws the window only if it is visible.
PRF_CHILDREN	Draws all visible children windows.
PRF_CLIENT	Draws the client area of the window.
PRF_ERASEBKGND	Erases the background before drawing the window.
PRF_NONCLIENT	Draws the nonclient area of the window.
PRF_OWNED	Draws all owned windows.

Remarks

The **DefWindowProc** function processes this message based on which drawing option is specified: if PRF_CHECKVISIBLE is specified and the window is not visible, do nothing; if PRF_NONCLIENT is specified, draw the nonclient area in the specified device context; if PRF_ERASEBKGND is specified, send the window a **WM_ERASEBKGND** message; if PRF_PRINTCLIENT is specified, send the window a **WM_PRINTCLIENT** message; if PRF_PRINTCHILDREN is set, send each visible child window a **WM_PRINT** message; if PRF_OWNED is set, send each visible owned window a **WM_PRINT** message.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Messages, **DefWindowProc**, **WM_ERASEBKGND**, **WM_PRINTCLIENT**

WM_PRINTCLIENT

The **WM_PRINTCLIENT** message is sent to a window to request that it draw its client area in the specified device context, most commonly in a printer device context.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_PRINTCLIENT
  WPARAM wParam,    // handle to DC (HDC)
  LPARAM lParam     // drawing options
);
```

Parameters

wParam

Handle to the device context in which to draw.

lParam

Specifies drawing options. This parameter can be one or more of the following values:

Value	Meaning
PRF_CHECKVISIBLE	Draws the window only if it is visible.
PRF_CHILDREN	Draws all visible children windows.
PRF_CLIENT	Draws the client area of the window.
PRF_ERASEBKGND	Erases the background before drawing the window.
PRF_NONCLIENT	Draws the nonclient area of the window.
PRF_OWNED	Draws all owned windows.

Remarks

A window can process this message in much the same manner as **WM_PAINT**, except that **BeginPaint** and **EndPaint** do not have to be called (a device context is provided), and the window should draw its entire client area instead of only the invalid region.

Windows that can be used anywhere in the system, such as controls, should process this message. It is probably worthwhile for other windows to process this message also, because it is relatively easy to implement.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

+ See Also

Painting and Drawing Overview, Painting and Drawing Messages, **BeginPaint**, **EndPaint**, **WM_PAINT**

WM_SETREDRAW

An application sends the **WM_SETREDRAW** message to a window to allow changes in that window to be redrawn or to prevent changes in that window from being redrawn.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
  (HWND) hWnd,              // handle to destination window
  WM_SETREDRAW,             // message to send
  (WPARAM) wParam;          // redraw state
  (LPARAM) lParam;          // not used; must be zero
);
```

Parameters

wParam
Specifies the redraw state. If this parameter is TRUE, the content can be redrawn after a change. If this parameter is FALSE, the content cannot be redrawn after a change.

lParam
This parameter is not used.

Return Values

An application returns zero if it processes this message.

Remarks

This message can be useful if an application must add several items to a list box. The application can call this message with *wParam* set to FALSE, add the items, and then call the message again with *wParam* set to TRUE. Finally, the application can call the **InvalidateRect** function to cause the list box to be repainted.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Messages, **InvalidateRect**

WM_SYNCPAINT

The **WM_SYNCPAINT** message is used to synchronize painting while avoiding linking independent GUI threads.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_SYNCPAINT
  WPARAM wParam,   // not used
  LPARAM lParam    // not used
);
```

Parameters

This message has no parameters.

Return Values

An application returns zero if it processes this message.

Remarks

When a window has been hidden, shown, moved, or sized, the system can determine that it is necessary to send a **WM_SYNCPAINT** message to the top-level windows of other threads. Applications must pass **WM_SYNCPAINT** to **DefWindowProc** for processing. The **DefWindowProc** function will send a **WM_NCPAINT** message to the window procedure if the window frame must be painted and send a **WM_ERASEBKGND** message if the window background must be erased.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Painting and Drawing Overview, Painting and Drawing Messages, **DefWindowProc**, **GetDCEx, GetWindowDC**, **WM_PAINT**

Raster-Operation Codes

Raster-operation codes define how the graphical device interface (GDI) combines the bits from the selected pen with the bits in the destination bitmap.

This overview lists and describes the binary and ternary raster operations used by GDI. A binary raster operation involves two operands: a pen and a destination bitmap. A ternary raster operation involves three operands: a source bitmap, a brush, and a destination bitmap. Both binary and ternary raster operations use Boolean operators.

Binary Raster Operations

This section lists the binary raster-operation codes used by the **GetROP2** and **SetROP2** functions. Raster-operation codes define how GDI combines the bits from the selected pen with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the values of the pixels in the selected pen and the destination bitmap are combined. The following are the two operands used in these operations:

Operand	Meaning
P	Selected pen
D	Destination bitmap

The Boolean operators used in these operations follow:

Operator	Meaning
a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the values of the pixels in the destination bitmap with a combination of the pixel values of the pen and the selected brush:

```
DPo
```

Each raster-operation code is a 32-bit integer whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended, 8-bit value that represents all possible outcomes resulting from the Boolean operation on two parameters (in this case, the pen and destination values). For example, the operation indexes for the DPo and DPan operations are shown in the following list:

P	D	DPo	Dpan
0	0	0	1
0	1	1	1
1	0	1	1
1	1	1	0

The following list outlines the drawing modes and the Boolean operations that they represent:

Raster operation	Boolean operation
R2_BLACK	0
R2_COPYPEN	P
R2_MASKNOTPEN	DPna
R2_MASKPEN	DPa
R2_MASKPENNOT	PDna
R2_MERGENOTPEN	DPno
R2_MERGEPEN	DPo
R2_MERGEPENNOT	PDno
R2_NOP	D
R2_NOT	Dn
R2_NOTCOPYPEN	Pn
R2_NOTMASKPEN	DPan
R2_NOTMERGEPEN	DPon
R2_NOTXORPEN	DPxn
R2_WHITE	1
R2_XORPEN	DPx

For a monochrome device, GDI maps the value zero to black and the value 1 to white. If an application attempts to draw with a black pen on a white destination by using the available binary raster operations, the following results occur:

Raster operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible black line
R2_MASKNOTPEN	No visible line
R2_MASKPEN	Visible black line
R2_MASKPENNOT	Visible black line
R2_MERGENOTPEN	No visible line
R2_MERGEPEN	Visible black line
R2_MERGEPENNOT	Visible black line
R2_NOP	No visible line
R2_NOT	Visible black line
R2_NOTCOPYPEN	No visible line
R2_NOTMASKPEN	No visible line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Visible black line
R2_WHITE	No visible line
R2_XORPEN	No visible line

For a color device, GDI uses RGB values to represent the colors of the pen and the destination. An RGB color value is a long integer that contains a red, a green, and a blue color field, each specifying the intensity of the specified color. Intensities range from 0 through 255. The values are packed in the three low-order bytes of the long integer. The color of a pen is always a solid color, but the color of the destination can be a mixture of any two or three colors. If an application attempts to draw with a white pen on a blue destination by using the available binary raster operations, the following results occur:

Raster operation	Result
R2_BLACK	Visible black line
R2_COPYPEN	Visible white line
R2_MASKNOTPEN	Visible black line
R2_MASKPEN	Invisible blue line
R2_MASKPENNOT	Visible red/green line
R2_MERGENOTPEN	Invisible blue line
R2_MERGEPEN	Visible white line
R2_MERGEPENNOT	Visible white line
R2_NOP	Invisible blue line
R2_NOT	Visible red/green line
R2_NOTCOPYPEN	Visible black line

R2_NOTMASKPEN	Visible red/green line
R2_NOTMERGEPEN	Visible black line
R2_NOTXORPEN	Invisible blue line
R2_WHITE	Visible white line
R2_XORPEN	Visible red/green line

Ternary Raster Operations

This section lists the ternary raster-operation codes used by the **BitBlt**, **PatBlt**, and **StretchBlt** functions. Ternary raster-operation codes define how GDI combines the bits in a source bitmap with the bits in the destination bitmap.

Each raster-operation code represents a Boolean operation in which the values of the pixels in the source, the selected brush, and the destination are combined. The following are the three operands used in these operations:

Operand	Meaning
D	Destination bitmap
P	Selected brush (also called pattern)
S	Source bitmap

Boolean operators used in these operations follow:

Operator	Meaning
a	Bitwise AND
n	Bitwise NOT (inverse)
o	Bitwise OR
x	Bitwise exclusive OR (XOR)

All Boolean operations are presented in reverse Polish notation. For example, the following operation replaces the values of the pixels in the destination bitmap with a combination of the pixel values of the source and brush:

```
PSo
```

The following operation combines the values of the pixels in the source and brush with the pixel values of the destination bitmap (there are alternative spellings of the same function, so, although a particular spelling might not be in the list, an equivalent form would be):

```
DPSoo
```

Each raster-operation code is a 32-bit integer whose high-order word is a Boolean operation index and whose low-order word is the operation code. The 16-bit operation index is a zero-extended, 8-bit value that represents the result of the Boolean operation on predefined brush, source, and destination values. For example, the operation indexes for the PSo and DPSoo operations are shown in the following list:

P	S	D	PSo	DPSoo
0	0	0	0	0
0	0	1	0	1
0	1	0	1	1
0	1	1	1	1
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1
Operation index:			00FCh	00FEh

In this case, PSo has the operation index 00FC (read from the bottom up); DPSoo has the operation index 00FE. These values define the location of the corresponding raster-operation codes, as shown in Table A.1, "Raster-Operation Codes." The PSo operation is in line 252 (00FCh) of the table; DPSoo is in line 254 (00FEh).

The most commonly used raster operations have been given special names in the SDK header file, Windows.h. You should use these names in your applications whenever possible.

When the source and destination bitmaps are monochrome, a bit value of zero represents a black pixel and a bit value of 1 represents a white pixel. When the source and the destination bitmaps are color, those colors are represented with RGB values. For more information about RGB values, see ***RGB***.

Raster-Operation Codes

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
00	00000042	0	**BLACKNESS**
01	00010289	DPSoon	–
02	00020C89	DPSona	–
03	000300AA	PSon	–
04	00040C88	SDPona	–
05	000500A9	DPon	–
06	00060865	PDSxnon	–

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
07	000702C5	PDSaon	–
08	00080F08	SDPnaa	–
09	00090245	PDSxon	–
0A	000A0329	DPna	–
0B	000B0B2A	PSDnaon	–
0C	000C0324	SPna	–
0D	000D0B25	PDSnaon	–
0E	000E08A5	PDSonon	–
0F	000F0001	Pn	–
10	00100C85	PDSona	–
11	001100A6	DSon	**NOTSRCERASE**
12	00120868	SDPxnon	–
13	001302C8	SDPaon	–
14	00140869	DPSxnon	–
15	001502C9	DPSaon	–
16	00165CCA	PSDPSanaxx	–
17	00171D54	SSPxDSxaxn	–
18	00180D59	SPxPDxa	–
19	00191CC8	SDPSanaxn	–
1A	001A06C5	PDSPaox	–
1B	001B0768	SDPSxaxn	–
1C	001C06CA	PSDPaox	–
1D	001D0766	DSPDxaxn	–
1E	001E01A5	PDSox	–
1F	001F0385	PDSoan	–
20	00200F09	DPSnaa	–
21	00210248	SDPxon	–
22	00220326	DSna	–
23	00230B24	SPDnaon	–
24	00240D55	SPxDSxa	–
25	00251CC5	PDSPanaxn	–
26	002606C8	SDPSaox	–
27	00271868	SDPSxnox	–
28	00280369	DPSxa	–
29	002916CA	PSDPSaoxxn	–

(continued)

(continued)

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
2A	002A0CC9	DPSana	–
2B	002B1D58	SSPxPDxaxn	–
2C	002C0784	SPDSoax	–
2D	002D060A	PSDnox	–
2E	002E064A	PSDPxox	–
2F	002F0E2A	PSDnoan	–
30	0030032A	PSna	–
31	00310B28	SDPnaon	–
32	00320688	SDPSoox	–
33	00330008	Sn	**NOTSRCCOPY**
34	003406C4	SPDSaox	–
35	00351864	SPDSxnox	–
36	003601A8	SDPox	–
37	00370388	SDPoan	–
38	0038078A	PSDPoax	–
39	00390604	SPDnox	–
3A	003A0644	SPDSxox	–
3B	003B0E24	SPDnoan	–
3C	003C004A	PSx	–
3D	003D18A4	SPDSonox	–
3E	003E1B24	SPDSnaox	–
3F	003F00EA	PSan	–
40	00400F0A	PSDnaa	–
41	00410249	DPSxon	–
42	00420D5D	SDxPDxa	–
43	00431CC4	SPDSanaxn	–
44	00440328	SDna	**SRCERASE**
45	00450B29	DPSnaon	–
46	004606C6	DSPDaox	–
47	0047076A	PSDPxaxn	–
48	00480368	SDPxa	–

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
49	004916C5	PDSPDaoxxn	–
4A	004A0789	DPSDoax	–
4B	004B0605	PDSnox	–
4C	004C0CC8	SDPana	–
4D	004D1954	SSPxDSxoxn	–
4E	004E0645	PDSPxox	–
4F	004F0E25	PDSnoan	–
50	00500325	PDna	–
51	00510B26	DSPnaon	–
52	005206C9	DPSDaox	–
53	00530764	SPDSxaxn	–
54	005408A9	DPSonon	–
55	00550009	Dn	**DSTINVERT**
56	005601A9	DPSox	–
57	00570389	DPSoan	–
58	00580785	PDSPoax	–
59	00590609	DPSnox	–
5A	005A0049	DPx	**PATINVERT**
5B	005B18A9	DPSDonox	–
5C	005C0649	DPSDxox	–
5D	005D0E29	DPSnoan	–
5E	005E1B29	DPSDnaox	–
5F	005F00E9	DPan	–
60	00600365	PDSxa	–
61	006116C6	DSPDSaoxxn	–
62	00620786	DSPDoax	–
63	00630608	SDPnox	–
64	00640788	SDPSoax	–
65	00650606	DSPnox	–
66	00660046	DSx	**SRCINVERT**
67	006718A8	SDPSonox	–
68	006858A6	DSPDSonoxxn	–
69	00690145	PDSxxn	–
6A	006A01E9	DPSax	–
6B	006B178A	PSDPSoaxxn	–

(continued)

(continued)

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
6C	006C01E8	SDPax	–
6D	006D1785	PDSPDoaxxn	–
6E	006E1E28	SDPSnoax	–
6F	006F0C65	PDSxnan	–
70	00700CC5	PDSana	–
71	00711D5C	SSDxPDxaxn	–
72	00720648	SDPSxox	–
73	00730E28	SDPnoan	–
74	00740646	DSPDxox	–
75	00750E26	DSPnoan	–
76	00761B28	SDPSnaox	–
77	007700E6	DSan	–
78	007801E5	PDSax	–
79	00791786	DSPDSoaxxn	–
7A	007A1E29	DPSDnoax	–
7B	007B0C68	SDPxnan	–
7C	007C1E24	SPDSnoax	–
7D	007D0C69	DPSxnan	–
7E	007E0955	SPxDSxo	–
7F	007F03C9	DPSaan	–
80	008003E9	DPSaa	–
81	00810975	SPxDSxon	–
82	00820C49	DPSxna	–
83	00831E04	SPDSnoaxn	–
84	00840C48	SDPxna	–
85	00851E05	PDSPnoaxn	–
86	008617A6	DSPDSoaxx	–
87	008701C5	PDSaxn	–
88	008800C6	DSa	**SRCAND**
89	00891B08	SDPSnaoxn	–
8A	008A0E06	DSPnoa	–

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
8B	008B0666	DSPDxoxn	–
8C	008C0E08	SDPnoa	–
8D	008D0668	SDPSxoxn	–
8E	008E1D7C	SSDxPDxax	–
8F	008F0CE5	PDSanan	–
90	00900C45	PDSxna	–
91	00911E08	SDPSnoaxn	–
92	009217A9	DPSDPoaxx	–
93	009301C4	SPDaxn	–
94	009417AA	PSDPSoaxx	–
95	009501C9	DPSaxn	–
96	00960169	DPSxx	–
97	0097588A	PSDPSonoxx	–
98	00981888	SDPSonoxn	–
99	00990066	DSxn	–
9A	009A0709	DPSnax	–
9B	009B07A8	SDPSoaxn	–
9C	009C0704	SPDnax	–
9D	009D07A6	DSPDoaxn	–
9E	009E16E6	DSPDSaoxx	–
9F	009F0345	PDSxan	–
A0	00A000C9	DPa	–
A1	00A11B05	PDSPnaoxn	–
A2	00A20E09	DPSnoa	–
A3	00A30669	DPSDxoxn	–
A4	00A41885	PDSPonoxn	–
A5	00A50065	PDxn	–
A6	00A60706	DSPnax	–
A7	00A707A5	PDSPoaxn	–
A8	00A803A9	DPSoa	–
A9	00A90189	DPSoxn	–
AA	00AA0029	D	–
AB	00AB0889	DPSono	–
AC	00AC0744	SPDSxax	–
AD	00AD06E9	DPSDaoxn	–

(continued)

(continued)

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
AE	00AE0B06	DSPnao	–
AF	00AF0229	DPno	–
B0	00B00E05	PDSnoa	–
B1	00B10665	PDSPxoxn	–
B2	00B21974	SSPxDSxox	–
B3	00B30CE8	SDPanan	–
B4	00B4070A	PSDnax	–
B5	00B507A9	DPSDoaxn	–
B6	00B616E9	DPSDPaoxx	–
B7	00B70348	SDPxan	–
B8	00B8074A	PSDPxax	–
B9	00B906E6	DSPDaoxn	–
BA	00BA0B09	DPSnao	–
BB	00BB0226	DSno	**MERGEPAINT**
BC	00BC1CE4	SPDSanax	–
BD	00BD0D7D	SDxPDxan	–
BE	00BE0269	DPSxo	–
BF	00BF08C9	DPSano	–
C0	00C000CA	PSa	**MERGECOPY**
C1	00C11B04	SPDSnaoxn	–
C2	00C21884	SPDSonoxn	–
C3	00C3006A	PSxn	–
C4	00C40E04	SPDnoa	–
C5	00C50664	SPDSxoxn	–
C6	00C60708	SDPnax	–
C7	00C707AA	PSDPoaxn	–
C8	00C803A8	SDPoa	–
C9	00C90184	SPDoxn	–
CA	00CA0749	DPSDxax	–
CB	00CB06E4	SPDSaoxn	–
CC	00CC0020	S	**SRCCOPY**

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
CD	00CD0888	SDPono	–
CE	00CE0B08	SDPnao	–
CF	00CF0224	SPno	–
D0	00D00E0A	PSDnoa	–
D1	00D1066A	PSDPxoxn	–
D2	00D20705	PDSnax	–
D3	00D307A4	SPDSoaxn	–
D4	00D41D78	SSPxPDxax	–
D5	00D50CE9	DPSanan	–
D6	00D616EA	PSDPSaoxx	–
D7	00D70349	DPSxan	–
D8	00D80745	PDSPxax	–
D9	00D906E8	SDPSaoxn	–
DA	00DA1CE9	DPSDanax	–
DB	00DB0D75	SPxDSxan	–
DC	00DC0B04	SPDnao	–
DD	00DD0228	SDno	–
DE	00DE0268	SDPxo	–
DF	00DF08C8	SDPano	–
E0	00E003A5	PDSoa	–
E1	00E10185	PDSoxn	–
E2	00E20746	DSPDxax	–
E3	00E306EA	PSDPaoxn	–
E4	00E40748	SDPSxax	–
E5	00E506E5	PDSPaoxn	–
E6	00E61CE8	SDPSanax	–
E7	00E70D79	SPxPDxan	–
E8	00E81D74	SSPxDSxax	–
E9	00E95CE6	DSPDSanaxxn	–
EA	00EA02E9	DPSao	–
EB	00EB0849	DPSxno	–
EC	00EC02E8	SDPao	–
ED	00ED0848	SDPxno	–
EE	00EE0086	DSo	**SRCPAINT**
EF	00EF0A08	SDPnoo	–

(continued)

(continued)

Boolean function (hexadecimal)	Raster operation (hexadecimal)	Boolean function in reverse Polish	Common name
F0	00F00021	P	**PATCOPY**
F1	00F10885	PDSono	–
F2	00F20B05	PDSnao	–
F3	00F3022A	PSno	–
F4	00F40B0A	PSDnao	–
F5	00F50225	PDno	–
F6	00F60265	PDSxo	–
F7	00F708C5	PDSano	–
F8	00F802E5	PDSao	–
F9	00F90845	PDSxno	–
FA	00FA0089	DPo	–
FB	00FB0A09	DPSnoo	**PATPAINT**
FC	00FC008A	PSo	–
FD	00FD0A0A	PSDnoo	–
FE	00FE02A9	DPSoo	–
FF	00FF0062	1	**WHITENESS**
8000	80000000		**Windows 98, Windows 2000:** NOMIRRORBITMAP

CHAPTER 16

Paths

A *path* is one or more figures (or shapes) that are filled, outlined, or both filled and outlined. Win32-based applications use paths in many ways. Paths are used in drawing and painting applications. Computer-aided design (CAD) applications use paths to create unique clipping regions, to draw outlines of irregular shapes, and to fill the interiors of irregular shapes. An irregular shape is a shape composed of Bézier curves and straight lines. (A regular shape is an ellipse, a circle, a rectangle, or a polygon.)

About Paths

A path is one of the objects associated with a device context (DC). However, unlike the default objects (the pen, the brush, and the font) that are part of any new DC, there is no default path. For more information about DCs, see *Device Contexts*.

To create a path and select it into a DC, it is first necessary to define the points that describe it. This is done by calling the **BeginPath** function, specifying the appropriate drawing functions, and then by calling the **EndPath** function. This combination of functions (**BeginPath**, drawing functions, and **EndPath**) constitute a *path bracket*.

Windows NT/2000: The following functions can be used in a path bracket.

AngleArc
Arc
ArcTo
Chord
CloseFigure
Ellipse
ExtTextOut
LineTo
MoveToEx
Pie
PolyBezier
PolyBezierTo
PolyDraw
Polygon
Polyline
PolylineTo
PolyPolygon
PolyPolyline
Rectangle
RoundRect
TextOut

Windows 95/98: When constructing a path, only the **CloseFigure**, **ExtTextOut**, **LineTo**, **MoveToEx**, **PolyBezier**, **PolyBezierTo**, **Polygon**, **Polyline**, **PolylineTo**, **PolyPolygon**, **PolyPolyline**, and **TextOut** functions are recorded.

When an application calls **EndPath**, the system selects the associated path into the specified DC. (If another path had previously been selected into the DC, the system deletes that path without saving it.) After the system selects the path into the DC, an application can operate on the path in one of the following ways:

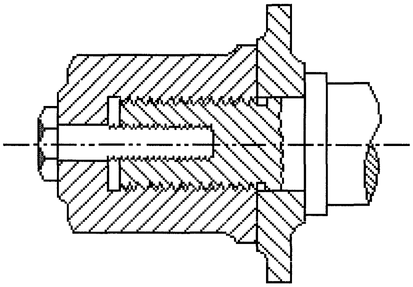
- Draw the outline of the path (using the current pen).
- Paint the interior of the path (using the current brush).
- Draw the outline and fill the interior of the path.
- Modify the path (converting curves to line segments).
- Convert the path into a clip path.
- Convert the path into a region.
- Flatten the path by converting each curve in the path into a series of line segments.
- Retrieve the coordinates of the lines and curves that compose a path.

Outlined and Filled Paths

An application can draw the outline of a path by calling the **StrokePath** function, it can fill the interior of a path by calling the **FillPath** function, and it can both outline and fill the path by calling the **StrokeAndFillPath** function.

Whenever an application fills a path, the system uses the DC's current fill mode. An application can retrieve this mode by calling the **GetPolyFillMode** function, and it can set a new fill mode by calling the **SetPolyFillMode** function. For a description of the two fill modes, see *Regions.*

The following illustration shows the cross-section of an object created by a computer-aided design (CAD) application using paths that were both outlined and filled.



Transformations of Paths

Paths are defined using logical units and current transformations. (If the **SetWorldTransform** function has been called, the logical units are world units; otherwise, the logical units are page units.) An application can use world transformations to scale, rotate, shear, translate, and reflect the lines and Bézier curves that define a path.

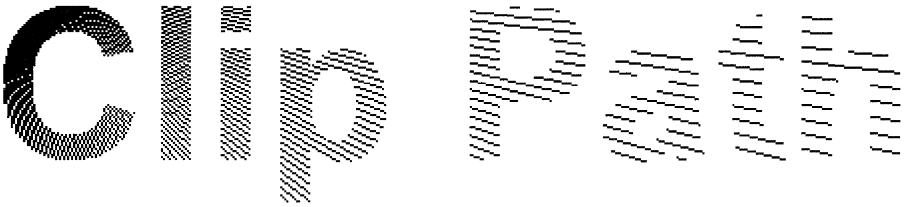
Note A world transformation within a path bracket only affects those lines and curves drawn *after* the transformation was set. It will have no affect on those lines and curves that were drawn before it was set. For a description of the world transformation, see *Coordinate Spaces and Transformations.*

An application can also use **SetWorldTransform** to outline the shape of the pen used to outline a path if the pen is a geometric pen. For a description of geometric pens, see *Pens.*

Clip Paths and Graphic Effects

An application can use clipping and paths to create special graphic effects. The following illustration shows a string of text drawn with a large Arial font.

The next illustration shows the result of selecting the text as a clip path and drawing radial lines for a circle whose center is located above and left of the string.



Note Before graphical device interface (GDI) adds text created with a bitmapped font to a path, it converts the font to an outline or vector font.

An application creates a clip path by generating a path bracket and then calling the **SelectClipPath** function. After a clip path is selected into a DC, output only appears within the borders of the path.

In addition to creating special graphics effects, clip paths are also useful in applications that save images as enhanced metafiles. By using a clip path, an application is able to ensure device independence because the units used to specify a path are logical units (as opposed to device units that are used to specify a region).

Conversion of Paths to Regions

An application can convert a path into a region by calling the **PathToRegion** function. Like **SelectClipPath**, **PathToRegion** is useful in the creation of special graphics effects. For example, there are no functions that allow an application to offset a path; however, there is a function that enables an application to offset a region (**OffsetRgn**). Using **PathToRegion**, an application can create the effect of animating a complex shape by creating a path that defines the shape, converting the path into a region (by calling **PathToRegion**), and then repeatedly painting, moving, and erasing the region (by calling functions in a sequence, such as **FillRgn**, **OffsetRgn**, and **FillRgn**).

Curved Paths

An application can flatten the curves in a path by calling the **FlattenPath** function. This function is especially useful for applications that fit text onto the contour of a path which contains curves. To fit the text, the application must perform the following steps:

1. Create the path where the text appears.
2. Call the **FlattenPath** function to convert the curves in a path into line segments.
3. Call the **GetPath** function to retrieve those line segments.
4. Calculate the length of each line and the width of each character in the string.
5. Use line-width and character-width data to position each character along the curve.

Path Reference

Path Functions

AbortPath

The **AbortPath** function closes and discards any paths in the specified device context.

```
BOOL AbortPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context from which a path will be discarded.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

If there is an open path bracket in the given device context, the path bracket is closed and the path is discarded. If there is a closed path in the device context, the path is discarded.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **BeginPath**, **EndPath**

BeginPath

The **BeginPath** function opens a path bracket in the specified device context.

```
BOOL BeginPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

After a path bracket is open, an application can begin calling GDI drawing functions to define the points that lie in the path. An application can close an open path bracket by calling the **EndPath** function.

When an application calls **BeginPath** for a device context, any previous paths are discarded from that device context.

Windows NT/2000: The following drawing functions define points in a path:

AngleArc
Arc
ArcTo
Chord
CloseFigure
Ellipse
ExtTextOut
LineTo
MoveToEx
Pie
PolyBezier
PolyBezierTo
PolyDraw
Polygon
Polyline
PolylineTo
PolyPolygon
PolyPolyline
Rectangle
RoundRect
TextOut

Windows 95/98: When constructing a path, only the **CloseFigure**, **ExtTextOut**, **LineTo**, **MoveToEx**, **PolyBezier**, **PolyBezierTo**, **Polygon**, **Polyline**, **PolylineTo**, **PolyPolygon**, **PolyPolyline**, and **TextOut** functions are recorded.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **EndPath**, **FillPath**, **PathToRegion**, **SelectClipPath**, **StrokeAndFillPath**, **StrokePath**, **WidenPath**

CloseFigure

The **CloseFigure** function closes an open figure in a path.

```
BOOL CloseFigure(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
[in] Handle to the device context in which the figure will be closed.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **CloseFigure** function closes the figure by drawing a line from the current position to the first point of the figure (usually, the point specified by the most recent call to the **MoveToEx** function) and then connects the lines by using the line join style. If a figure is closed by using the **LineTo** function instead of **CloseFigure**, end caps are used to create the corner instead of a join.

The **CloseFigure** function should only be called if there is an open path bracket in the specified device context.

A figure in a path is open unless it is explicitly closed by using this function. (A figure can be open even if the current point and the starting point of the figure are the same.)

After a call to **CloseFigure**, adding a line or curve to the path starts a new figure.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Paths Overview, Path Functions, **BeginPath**, **EndPath**, **ExtCreatePen**, **LineTo**, **MoveToEx**

EndPath

The **EndPath** function closes a path bracket and selects the path defined by the bracket into the specified device context.

```
BOOL EndPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to the device context into which the new path is selected.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **BeginPath**

FillPath

The **FillPath** function closes any open figures in the current path and fills the path's interior by using the current brush and polygon-filling mode.

```
BOOL FillPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to a device context that contains a valid path.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Remarks

After its interior is filled, the path is discarded from the DC identified by the *hdc* parameter.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Paths Overview, Path Functions, **BeginPath**, **SetPolyFillMode**, **StrokeAndFillPath**, **StrokePath**

FlattenPath

The **FlattenPath** function transforms any curves in the path that is selected into the current device context (DC), turning each curve into a sequence of lines.

```
BOOL FlattenPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to a DC that contains a valid path.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **WidenPath**

GetMiterLimit

The **GetMiterLimit** function returns the miter limit for the specified device context.

```
BOOL GetMiterLimit(
  HDC hdc,          // handle to DC
  PFLOAT peLimit    // miter limit
);
```

Parameters

hdc
  [in] Handle to the device context.

peLimit
  [out] Pointer to a floating-point value that receives the current miter limit.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The miter limit is used when drawing geometric lines that have miter joins.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **ExtCreatePen**, **SetMiterLimit**

GetPath

The **GetPath** function retrieves the coordinates defining the endpoints of lines and the control points of curves found in the path that is selected into the specified device context.

```
int GetPath(
  HDC hdc,              // handle to DC
  LPPOINT lpPoints,     // path vertices
  LPBYTE lpTypes,       // array of path vertex types
  int nSize             // count of points defining path
);
```

Parameters

hdc

[in] Handle to a device context that contains a closed path.

lpPoints

[out] Pointer to an array of **POINT** structures that receives the line endpoints and curve control points.

lpTypes

[out] Pointer to an array of bytes that receives the vertex types. This parameter can be one of the following values.

Type	Description
PT_MOVETO	Specifies that the corresponding point in the *lpPoints* parameter starts a disjoint figure.
PT_LINETO	Specifies that the previous point and the corresponding point in *lpPoints* are the endpoints of a line.
PT_BEZIERTO	Specifies that the corresponding point in *lpPoints* is a control point or ending point for a Bézier curve. PT_BEZIERTO values always occur in sets of three. The point in the path immediately preceding them defines the starting point for the Bézier curve. The first two PT_BEZIERTO points are the control points, and the third PT_BEZIERTO point is the ending (if hard-coded) point.

A PT_LINETO or PT_BEZIERTO value may be combined with the following value (by using the bitwise operator OR) to indicate that the corresponding point is the last point in a figure and the figure should be closed.

Flag	Description
PT_CLOSEFIGURE	Specifies that the figure is automatically closed after the corresponding line or curve is drawn. The figure is closed by drawing a line from the line or curve endpoint to the point corresponding to the last PT_MOVETO.

nSize

[in] Specifies the total number of **POINT** structures that can be stored in the array pointed to by *lpPoints*. This value must be the same as the number of bytes that can be placed in the array pointed to by *lpTypes*.

Return Values

If the *nSize* parameter is nonzero, the return value is the number of points enumerated. If *nSize* is 0, the return value is the total number of points in the path (and **GetPath** writes nothing to the buffers). If *nSize* is nonzero and is less than the number of points in the path, the return value is –1.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_BUFFER_OVERFLOW

Remarks

The device context identified by the *hdc* parameter must contain a closed path.

The points of the path are returned in logical coordinates. Points are stored in the path in device coordinates, so **GetPath** changes the points from device coordinates to logical coordinates by using the inverse of the current transformation.

The **FlattenPath** function may be called before **GetPath** to convert all curves in the path into line segments.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Paths Overview, Path Functions, **FlattenPath**, **POINT**, **PolyDraw**, **WidenPath**

PathToRegion

The **PathToRegion** function creates a region from the path that is selected into the specified device context. The resulting region uses device coordinates.

```
HRGN PathToRegion(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to a device context that contains a closed path.

Return Values

If the function succeeds, the return value identifies a valid region.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Remarks

The device context identified by the *hdc* parameter must contain a closed path.

After **PathToRegion** converts a path into a region, the system discards the closed path from the specified device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Paths Overview, Path Functions, **BeginPath**, **EndPath**, **SetPolyFillMode**

SetMiterLimit

The **SetMiterLimit** function sets the limit for the length of miter joins for the specified device context.

```
BOOL SetMiterLimit(
  HDC hdc,            // handle to DC
  FLOAT eNewLimit,    // new miter limit
  PFLOAT peOldLimit   // previous miter limit
);
```

Parameters

hdc
[in] Handle to the device context.

eNewLimit
[in] Specifies the new miter limit for the device context.

peOldLimit
[out] Pointer to a floating-point value that receives the previous miter limit. If this parameter is NULL, the previous miter limit is not returned.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The miter length is defined as the distance from the intersection of the line walls on the inside of the join to the intersection of the line walls on the outside of the join. The miter limit is the maximum allowed ratio of the miter length to the line width.

The default miter limit is 10.0.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **ExtCreatePen**, **GetMiterLimit**

StrokeAndFillPath

The **StrokeAndFillPath** function closes any open figures in a path, strokes the outline of the path by using the current pen, and fills its interior by using the current brush.

```
BOOL StrokeAndFillPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc
[in] Handle to the device context.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Remarks

The device context identified by the *hdc* parameter must contain a closed path.

The **StrokeAndFillPath** function has the same effect as closing all the open figures in the path, and stroking and filling the path separately, except that the filled region will not overlap the stroked region even if the pen is wide.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **BeginPath**, **FillPath**, **SetPolyFillMode**, **StrokePath**

StrokePath

The **StrokePath** function renders the specified path by using the current pen.

```
BOOL StrokePath(
  HDC hdc    // handle to DC
);
```

Parameters

hdc

[in] Handle to a device context that contains a closed path.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Remarks

The device context identified by the *hdc* parameter must contain a closed path.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Paths Overview, Path Functions, **BeginPath**, **EndPath**, **ExtCreatePen**

WidenPath

The **WidenPath** function redefines the current path as the area that would be painted if the path were stroked using the pen currently selected into the given device context.

```
BOOL WidenPath(
  HDC hdc   // handle to DC
);
```

Parameters

hdc

[in] Handle to a device context that contains a closed path.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_CAN_NOT_COMPLETE
ERROR_INVALID_PARAMETER
ERROR_NOT_ENOUGH_MEMORY

Remarks

The **WidenPath** function is successful only if the current pen is a geometric pen created by the **ExtCreatePen** function, or if the pen is created with the **CreatePen** function and has a width, in device units, of more than one.

The device context identified by the *hdc* parameter must contain a closed path.

Any Bézier curves in the path are converted to sequences of straight lines approximating the widened curves. As such, no Bézier curves remain in the path after **WidenPath** is called.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Paths Overview, Path Functions, **BeginPath**, **CreatePen**, **EndPath**, **ExtCreatePen**, **SetMiterLimit**

CHAPTER 17

Pens

A pen is a graphics tool that a Win32-based application uses to draw lines and curves. Drawing applications use pens to draw freehand lines, straight lines, and curves. Computer-aided design (CAD) applications use pens to draw visible lines, hidden lines, section lines, center lines, and so on. Word processing and desktop publishing applications use pens to draw borders and rules. Spreadsheet applications use pens to designate trends in graphs and to outline bar graphs and pie charts.

About Pens

There are two types of pens: cosmetic and geometric. A *cosmetic pen* is used with applications requiring lines of fixed width and lines that are quickly drawn. A CAD application, for example, uses a cosmetic pen to generate hidden, section, center, and dimension lines that are between .015 and .022 inches wide—regardless of the scale factor. A *geometric pen* is used with applications requiring scalable lines, lines with unique end or join styles, and lines that are wider than a single pixel. A spreadsheet application, for example, uses a geometric pen to define each of the bars in a bar graph as a wide line.

Cosmetic Pens

The dimensions of a cosmetic pen are specified in device units. Therefore, lines drawn with a cosmetic pen always have a fixed width. Lines drawn with a cosmetic pen are generally drawn 3 to 10 times faster than lines drawn with a geometric pen. Cosmetic pens have three attributes: width, style, and color. For more information about these attributes, see *Pen Attributes*.

To create a cosmetic pen, use the **CreatePen**, **CreatePenIndirect**, or **ExtCreatePen** function. To retrieve one of the three stock cosmetic pens managed by the system, use the **GetStockObject** function.

After you create a pen (or obtain a handle to one of the stock pens), select the pen into the application's device context (DC) using the **SelectObject** function. From this point on, the application uses this pen for any line-drawing operations in its client area.

Geometric Pens

The dimensions of a geometric pen are specified in logical units. Therefore, lines drawn with a geometric pen can be scaled—that is, they may appear wider or narrower, depending on the current world transformation. For more information about world transformation, see *Coordinate Spaces and Transformations*.

In addition to the three attributes shared with cosmetic pens (width, style, and color), geometric pens possess the following four attributes: pattern, optional hatch, end style, and join style. For more information about these attributes, see *Pen Attributes.*

To create a geometric pen, an application uses the **ExtCreatePen** function. As with cosmetic pens, the **SelectObject** function selects a geometric pen into the application's DC.

Pen Attributes

There are seven pen attributes that define the type of pen and its characteristics: width, style, color, pattern, hatch, end style, and join style. Both cosmetic and geometric pens have the width, style, and color attributes. Only geometric pens have the pattern, hatch, end style, and join style attributes. The pattern and optional hatch attribute are usually associated with a brush but can also be used with geometricpens.

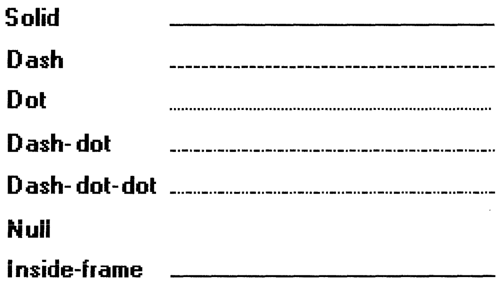
Pen Width

The width attribute specifies a cosmetic pen width in device units. When used with a geometric pen, however, it specifies the pen's width in logical units. For more information about device units, see *Coordinate Spaces and Transformations.*

Currently, the system limits the width of cosmetic pens to a single pixel; however, future versions may remove this limitation.

Pen Style

The style attribute specifies the line pattern that appears when a particular cosmetic or geometric pen is used. There are eight predefined pen styles. The following illustration shows the seven of these styles that are defined by the system.



The inside-frame style is identical to the solid style for cosmetic pens. However, it operates differently when used with a geometric pen. If the geometric pen is wider than a single pixel and a drawing function uses the pen to draw a border around a filled object, the system draws the border *inside* the object's frame. By using the inside-frame style, an application can ensure that an object appears entirely within the specified dimensions, regardless of the geometric pen width.

In addition to the seven styles defined by the system, there is an eighth style that is user (or application) defined. A user-defined style generates lines with a customized series of dashes and dots.

Use the **CreatePen**, **CreatePenIndirect**, or **ExtCreatePen** function to create a pen that has the system-defined styles. Use the **ExtCreatePen** function to create a pen that has a user-defined style.

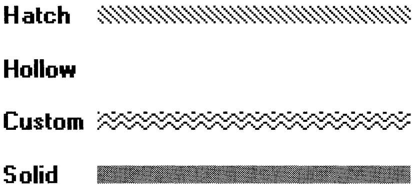
Pen Color

The color attribute specifies the pen's color. An application can create a cosmetic pen with a unique color by using the **RGB** macro to store the red, green, blue triplet that specifies the desired color in a **COLORREF** structure and passing this structure's address to the **CreatePen**, **CreatePenIndirect**, or **ExtCreatePen** function. (The stock pens are limited to black, white, and invisible.) For more information about RGB triplets and color, see *Colors*.

Pen Pattern

The pattern attribute specifies the pattern of a geometric pen.

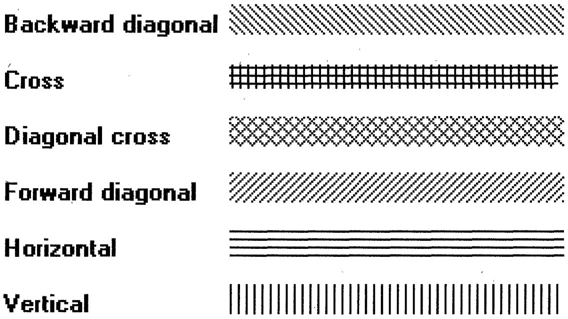
The following illustration shows lines drawn with different geometric pens. Each pen was created using a different pattern attribute.



The first line in the previous illustration is drawn using one of the six available hatch patterns; for more information about hatch patterns, see *Pen Hatch*. The next line is drawn using the hollow pattern, identical to the null pattern. The third line is drawn using a custom pattern created from an 8-pixel-by-8-pixel bitmap. (For more information about bitmaps and their creation, see *Bitmaps*.) The last line is drawn using a solid pattern. Creating a brush and passing its handle to the **ExtCreatePen** function creates a pattern.

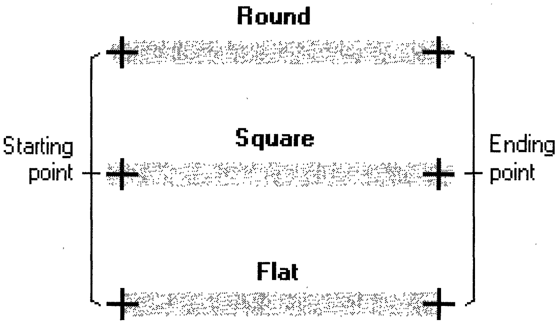
Pen Hatch

The hatch attribute specifies the hatch type of a geometric pen with the hatch pattern attribute. There are six patterns available. The following illustration shows lines drawn using different hatch patterns.



Pen End Cap

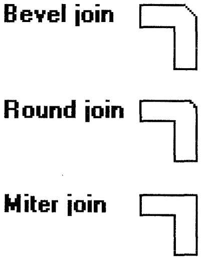
The end cap attribute specifies the shape of a geometric pen: round, square, or flat. The following illustration shows parallel lines drawn using each type of end cap.



The round and square end caps extend past the starting and ending points of a line drawn with a geometric pen; the flat end cap does not.

Pen Join

The join attribute specifies how the ends of two geometric lines are joined: beveled, mitered, or round. The following illustration shows pairs of connected lines drawn using each type of join.



ICM-Enabled Pen Functions

Microsoft Windows 98 and Microsoft Windows 2000 have been designed to work with Microsoft Image Color Management (ICM). ICM technology ensures that a color image, graphic, or text object is rendered as close as possible to its original intent on any device, despite differences in imaging technologies and color capabilities among devices. Whether you are scanning an image or other graphic on a color scanner, downloading it over the Internet, viewing or editing it on the screen, or outputting it to paper, film, or other media, ICM version 2.0 helps you keep its colors consistent and accurate. For more information about ICM, see *About Image Color Management Version 2.0.*

There are various functions in the graphical device interface (GDI) that use or operate on color data. The following pen functions are enabled for use with ICM:

- **CreatePen**
- **ExtCreatePen**

Pen Reference

Pen Functions

CreatePen

The **CreatePen** function creates a logical pen that has the specified style, width, and color. The pen can subsequently be selected into a device context and used to draw lines and curves.

```
HPEN CreatePen(
  int fnPenStyle,    // pen style
  int nWidth,        // pen width
  COLORREF crColor   // pen color
);
```

Parameters

fnPenStyle

[in] Specifies the pen style. It can be any one of the following values:

Value	Meaning
PS_DASH	The pen is dashed. This style is valid only when the pen width is one or less in device units.
PS_DOT	The pen is dotted. This style is valid only when the pen width is one or less in device units.

(continued)

(continued)

Value	Meaning
PS_DASHDOT	The pen has alternating dashes and dots. This style is valid only when the pen width is one or less in device units.
PS_DASHDOTDOT	The pen has alternating dashes and double dots. This style is valid only when the pen width is one or less in device units.
PS_INSIDEFRAME	The pen is solid. When this pen is used in any GDI drawing function that takes a bounding rectangle, the dimensions of the figure are shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen. This applies only to geometric pens.
PS_NULL	The pen is invisible.
PS_SOLID	The pen is solid.

nWidth

[in] Specifies the width of the pen, in logical units. If *nWidth* is zero, the pen is a single pixel wide, regardless of the current transformation.

CreatePen returns a pen with the specified width bit with the PS_SOLID style if you specify a width greater than one for the following styles: PS_DASH, PS_DASHDOT, PS_DASHDOTDOT, PS_DOT.

crColor

[in] Specifies a color reference for the pen color. To generate a **COLORREF** structure, use the **RGB** macro.

Return Values

If the function succeeds, the return value is a handle that identifies a logical pen.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

After an application creates a logical pen, it can select that pen into a device context by calling the **SelectObject** function. After a pen is selected into a device context, it can be used to draw lines and curves.

If the value specified by the *nWidth* parameter is zero, a line drawn with the created pen always is a single pixel wide regardless of the current transformation.

If the value specified by *nWidth* is greater than 1, the *fnPenStyle* parameter must be PS_NULL, PS_SOLID, or PS_INSIDEFRAME.

If the value specified by *nWidth* is greater than 1 and *fnPenStyle* is PS_INSIDEFRAME, the line associated with the pen is drawn inside the frame of all primitives except polygons and polylines.

If the value specified by *nWidth* is greater than 1, *fnPenStyle* is PS_INSIDEFRAME, and the color specified by the *crColor* parameter does not match one of the entries in the logical palette, the system draws lines by using a dithered color. Dithered colors are not available with solid pens.

When you no longer need the pen, call the **DeleteObject** function to delete it.

ICM: No color management is done at creation. However, color management is performed when the pen is selected into an ICM-enabled device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Pens Overview, Pen Functions, **COLORREF**, **CreatePenIndirect**, **DeleteObject**, **ExtCreatePen**, **GetObject**, **RGB**, **SelectObject**

CreatePenIndirect

The **CreatePenIndirect** function creates a logical cosmetic pen that has the style, width, and color specified in a structure.

```
HPEN CreatePenIndirect(
  CONST LOGPEN *lplgpn   // style, width, and color
);
```

Parameters

lplgpn
[in] Pointer to a **LOGPEN** structure that specifies the pen's style, width, and color.

Return Values

If the function succeeds, the return value is a handle that identifies a logical cosmetic pen.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

After an application creates a logical pen, it can select that pen into a device context by calling the **SelectObject** function. After a pen is selected into a device context, it can be used to draw lines and curves.

When you no longer need the pen, call the **DeleteObject** function to delete it.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Pens Overview, Pen Functions, **CreatePen**, **DeleteObject**, **ExtCreatePen**, **GetObject**, **LOGPEN**, **RGB**, **SelectObject**

ExtCreatePen

The **ExtCreatePen** function creates a logical cosmetic or geometric pen that has the specified style, width, and brush attributes.

```
HPEN ExtCreatePen(
  DWORD dwPenStyle,          // pen style
  DWORD dwWidth,             // pen width
  CONST LOGBRUSH *lplb,      // brush attributes
  DWORD dwStyleCount,        // length of custom style array
  CONST DWORD *lpStyle       // custom style array
);
```

Parameters

dwPenStyle

[in] Specifies a combination of type, style, end cap, and join attributes. The values from each category are combined by using the bitwise OR operator (|).

The pen type can be one of the following values:

Value	Meaning
PS_COSMETIC	The pen is cosmetic.
PS_GEOMETRIC	The pen is geometric.

The pen style can be one of the following values:

Value	Meaning
PS_ALTERNATE	**Windows NT/2000:** The pen sets every other pixel. (This style is applicable only for cosmetic pens.)
PS_DASH	The pen is dashed. **Windows 95:** This style is not supported for geometric lines. **Windows 98:** Not supported.
PS_DOT	The pen is dotted. **Windows 95/98:** This style is not supported for geometric lines.
PS_DASHDOT	The pen has alternating dashes and dots. **Windows 95:** This style is not supported for geometric lines. **Windows 98:** Not supported.
PS_DASHDOTDOT	The pen has alternating dashes and double dots. **Windows 95:** This style is not supported for geometric lines. **Windows 98:** Not supported.
PS_INSIDEFRAME	The pen is solid. When this pen is used in any GDI drawing function that takes a bounding rectangle, the dimensions of the figure are shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen. This applies only to geometric pens.
PS_NULL	The pen is invisible.
PS_SOLID	The pen is solid.
PS_USERSTYLE	**Windows NT/2000:** The pen uses a styling array supplied by the user.

The end cap is only specified for geometric pens. The end cap can be one of the following values:

Value	Meaning
PS_ENDCAP_FLAT	End caps are flat.
PS_ENDCAP_ROUND	End caps are round.
PS_ENDCAP_SQUARE	End caps are square.

The join is only specified for geometric pens. The join can be one of the following values:

Value	Meaning
PS_JOIN_BEVEL	Joins are beveled.
PS_JOIN_MITER	Joins are mitered when they are within the current limit set by the **SetMiterLimit** function. If it exceeds this limit, the join is beveled.
PS_JOIN_ROUND	Joins are round.

Windows 95/98: The PS_ENDCAP_ROUND, PS_ENDCAP_SQUARE, PS_ENDCAP_FLAT, PS_JOIN_BEVEL, PS_JOIN_MITER, and PS_JOIN_ROUND styles are supported only for geometric pens when used to draw paths.

dwWidth

[in] Specifies the width of the pen. If the *dwPenStyle* parameter is PS_GEOMETRIC, the width is given in logical units. If *dwPenStyle* is PS_COSMETIC, the width must be set to 1.

lplb

[in] Pointer to a **LOGBRUSH** structure. If *dwPenStyle* is PS_COSMETIC, the **lbColor** member specifies the color of the pen and the **lbStyle** member must be set to BS_SOLID. If *dwPenStyle* is PS_GEOMETRIC, all members must be used to specify the brush attributes of the pen.

dwStyleCount

[in] Specifies the length, in **DWORD** units, of the *lpStyle* array. This value must be zero if *dwPenStyle* is not PS_USERSTYLE.

lpStyle

[in] Pointer to an array. The first value specifies the length of the first dash in a user-defined style, the second value specifies the length of the first space, and so on. This pointer must be NULL if *dwPenStyle* is not PS_USERSTYLE.

Return Values

If the function succeeds, the return value is a handle that identifies a logical pen.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A geometric pen can have any width and can have any of the attributes of a brush, such as dithers and patterns. A cosmetic pen can only be a single pixel wide and must be a solid color, but cosmetic pens are generally faster than geometric pens.

The width of a geometric pen is always specified in world units. The width of a cosmetic pen is always 1.

End caps and joins are only specified for geometric pens.

After an application creates a logical pen, it can select that pen into a device context by calling the **SelectObject** function. After a pen is selected into a device context, it can be used to draw lines and curves.

If *dwPenStyle* is PS_COSMETIC and PS_USERSTYLE, the entries in the *lpStyle* array specify lengths of dashes and spaces in style units. A style unit is defined by the device where the pen is used to draw a line.

If *dwPenStyle* is PS_GEOMETRIC and PS_USERSTYLE, the entries in the *lpStyle* array specify lengths of dashes and spaces in logical units.

If *dwPenStyle* is PS_ALTERNATE, the style unit is ignored and every other pixel is set.

If the **lbStyle** member of the **LOGBRUSH** structure pointed to by *lplb* is BS_PATTERN, the bitmap pointed to by the **lbHatch** member of that structure cannot be a DIB section. A DIB section is a bitmap created by **CreateDIBSection**. If that bitmap is a DIB section, the **ExtCreatePen** function fails.

When an application no longer requires a specified pen, it should call the **DeleteObject** function to delete the pen.

ICM: No color management is done at pen creation. However, color management is performed when the pen is selected into an ICM-enabled device context.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Pens Overview, Pen Functions, **CreateDIBSection**, **CreatePen**, **CreatePenIndirect**, **DeleteObject**, **GetObject**, **LOGBRUSH**, **SelectObject**, **SetMiterLimit**

Pen Structures

EXTLOGPEN

The **EXTLOGPEN** structure defines the pen style, width, and brush attributes for an extended pen. This structure is used by the **GetObject** function when it retrieves a description of a pen that was created when an application called the **ExtCreatePen** function.

```
typedef struct tagEXTLOGPEN {
  DWORD     elpPenStyle;
  DWORD     elpWidth;
  UINT      elpBrushStyle;
  COLORREF  elpColor;
  ULONG_PTR elpHatch;
  DWORD     elpNumEntries;
  DWORD     elpStyleEntry[1];
} EXTLOGPEN, *PEXTLOGPEN;
```

Members

elpPenStyle

Specifies a combination of pen type, style, end cap style, and join style. The values from each category can be retrieved by using a bitwise AND operator with the appropriate mask.

The **elpPenStyle** member masked with PS_TYPE_MASK has one of the following pen type values:

Value	Meaning
PS_COSMETIC	The pen is cosmetic.
PS_GEOMETRIC	The pen is geometric.

The **elpPenStyle** member masked with PS_STYLE_MASK has one of the following pen styles values:

Value	Meaning
PS_DASH	The pen is dashed.
PS_DASHDOT	The pen has alternating dashes and dots.
PS_DASHDOTDOT	The pen has alternating dashes and double dots.
PS_DOT	The pen is dotted.
PS_INSIDEFRAME	The pen is solid. When this pen is used in any GDI drawing function that takes a bounding rectangle, the dimensions of the figure are shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen. This applies only to PS_GEOMETRIC pens.
PS_NULL	The pen is invisible.
PS_SOLID	The pen is solid.
PS_USERSTYLE	The pen uses a styling array supplied by the user.

The following category applies only to PS_GEOMETRIC pens. The **elpPenStyle** member masked with PS_ENDCAP_MASK has one of the following end cap values:

Value	Meaning
PS_ENDCAP_FLAT	Line end caps are flat.
PS_ENDCAP_ROUND	Line end caps are round.
PS_ENDCAP_SQUARE	Line end caps are square.

The following category applies only to PS_GEOMETRIC pens. The **elpPenStyle** member masked with PS_JOIN_MASK has one of the following join values:

Value	Meaning
PS_JOIN_BEVEL	Line joins are beveled.
PS_JOIN_MITER	Line joins are mitered when they are within the current limit set by the **SetMiterLimit** function. A join is beveled when it would exceed the limit.
PS_JOIN_ROUND	Line joins are round.

elpWidth

Specifies the width of the pen. If the **elpPenStyle** member specifies geometric lines, this value is the width, in logical units, of the line. Otherwise, the lines are cosmetic and this value is 1.

elpBrushStyle

Specifies the brush style of the pen. The **elpBrushStyle** member value can be one of the following:

Value	Meaning
BS_DIBPATTERN	Specifies a pattern brush defined by a DIB specification. If **elpBrushStyle** is BS_DIBPATTERN, the **elpHatch** member contains a handle to a packed DIB. For more information, see discussion in **elpHatch.**
BS_DIBPATTERNPT	Specifies a pattern brush defined by a DIB specification. If **elpBrushStyle** is BS_DIBPATTERNPT, the **elpHatch** member contains a pointer to a packed DIB. For more information, see discussion in **elpHatch**.
BS_HATCHED	Specifies a hatched brush.
BS_HOLLOW	Specifies a hollow or NULL brush.
BS_PATTERN	Specifies a pattern brush defined by a memory bitmap.
BS_SOLID	Specifies a solid brush.

elpColor

If **elpBrushStyle** is BS_SOLID or BS_HATCHED, **elpColor** specifies the color in which the pen is to be drawn. For BS_HATCHED, the **SetBkMode** and **SetBkColor** functions determine the background color.

If **elpBrushStyle** is BS_HOLLOW or BS_PATTERN, **elpColor** is ignored.

If **elpBrushStyle** is BS_DIBPATTERN or BS_DIBPATTERNPT, the low-order word of **elpColor** specifies whether the **bmiColors** members of the **BITMAPINFO** structure contain explicit **RGB** values or indices into the currently realized logical palette. The **elpColor** value must be one of the following:

Value	Meaning
DIB_PAL_COLORS	The color table consists of an array of 16-bit indices into the currently realized logical palette.
DIB_RGB_COLORS	The color table contains literal RGB values.

The RGB macro is used to generate a **COLORREF** structure.

elpHatch

If **elpBrushStyle** is BS_PATTERN, **elpHatch** is a handle to the bitmap that defines the pattern.

If **elpBrushStyle** is BS_SOLID or BS_HOLLOW, **elpHatch** is ignored.

If **elpBrushStyle** is BS_DIBPATTERN, the **elpHatch** member is a handle to a packed DIB. To obtain this handle, an application calls the **GlobalAlloc** function with GMEM_MOVEABLE (or **LocalAlloc** with LMEM_MOVEABLE) to allocate a block of memory and then fills the memory with the packed DIB. A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes that define the pixels of the bitmap.

If **elpBrushStyle** is BS_DIBPATTERNPT, the **elpHatch** member is a pointer to a packed DIB. The pointer derives from the memory block created by **LocalAlloc** with LMEM_FIXED set or by **GlobalAlloc** with GMEM_FIXED set, or it is the pointer returned by a call like **LocalLock** (handle_to_the_dib). A packed DIB consists of a **BITMAPINFO** structure immediately followed by the array of bytes that define the pixels of the bitmap.

If **elpBrushStyle** is BS_HATCHED, the **elpHatch** member specifies the orientation of the lines used to create the hatch. It can be one of the following values:

Value	Meaning
HS_BDIAGONAL	45-degree upward hatch (left to right)
HS_CROSS	Horizontal and vertical crosshatch
HS_DIAGCROSS	45-degree crosshatch
HS_FDIAGONAL	45-degree downward hatch (left to right)
HS_HORIZONTAL	Horizontal hatch
HS_VERTICAL	Vertical hatch

elpNumEntries

Specifies the number of entries in the style array in the **elpStyleEntry** member. This value is zero if **elpPenStyle** does not specify PS_USERSTYLE.

elpStyleEntry

Specifies a user-supplied style array. The array is specified with a finite length, but it is used as if it repeated indefinitely. The first entry in the array specifies the length of the first dash. The second entry specifies the length of the first gap. Thereafter, lengths of dashes and gaps alternate.

If **elpWidth** specifies geometric lines, the lengths are in logical units. Otherwise, the lines are cosmetic and lengths are in device units.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Pens Overview, Pen Structures, **BITMAPINFO**, **COLORREF**, **ExtCreatePen**, **GetObject**, **GlobalAlloc**, **RGB**, **SetBkColor**, **SetBkMode**

LOGPEN

The **LOGPEN** structure defines the style, width, and color of a pen. The **CreatePenIndirect** function uses the **LOGPEN** structure.

```
typedef struct tagLOGPEN {
    UINT      lopnStyle;
    POINT     lopnWidth;
    COLORREF  lopnColor;
} LOGPEN, *PLOGPEN;
```

Members

lopnStyle

Specifies the pen style, which can be one of the following values:

Value	Meaning
PS_DASH	The pen is dashed.
PS_DOT	The pen is dotted.
PS_DASHDOT	The pen has alternating dashes and dots.
PS_DASHDOTDOT	The pen has dashes and double dots.
PS_INSIDEFRAME	The pen is solid. When this pen is used in any GDI drawing function that takes a bounding rectangle, the dimensions of the figure are shrunk so that it fits entirely in the bounding rectangle, taking into account the width of the pen. This applies only to geometric pens.
PS_NULL	The pen is invisible.
PS_SOLID	The pen is solid.

lopnWidth

Specifies the **POINT** structure that contains the pen width, in logical units. If the **pointer** member is NULL, the pen is one pixel wide on raster devices. The **y** member in the **POINT** structure for **lopnWidth** is not used.

lopnColor

Specifies the pen color. To generate a **COLORREF** structure, use the **RGB** macro.

Remarks

If the width of the pen is greater than 1 and the pen style is PS_INSIDEFRAME, the line is drawn inside the frame of all GDI objects except polygons and polylines. If the pen color does not match an available RGB value, the pen is drawn with a logical (dithered) color. If the pen width is less than or equal to 1, the PS_INSIDEFRAME style is identical to the PS_SOLID style.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Pens Overview, Pen Structures, **COLORREF**, **CreatePenIndirect**, **POINT**, **RGB**

CHAPTER 18

Rectangles

Microsoft Win32 -based applications use *rectangles* to specify rectangular areas on the screen or in a window.

About Rectangles

In Win32-based applications, rectangles are used for the cursor clipping region, the invalid portion of the client area, an area for displaying formatted text, or the scroll area. Your applications can also use rectangles to fill, frame, or invert a portion of the client area with a given brush, and to retrieve the coordinates of a window or a window's client area.

Rectangle Coordinates

An application must use a **RECT** structure to define a rectangle. The structure specifies the coordinates of two points: the upper left and lower right corners of the rectangle. The sides of the rectangle extend from these two points and are parallel to the x- and y-axes.

The coordinate values for a rectangle are expressed as signed integers. The coordinate value of a rectangle's right side must be greater than that of its left side. Likewise, the coordinate value of the bottom must be greater than that of the top.

Because applications can use rectangles for many different purposes, the Win32 rectangle functions do not use an explicit unit of measure. Instead, all rectangle coordinates and dimensions are given in signed, logical values. The mapping mode and the function in which the rectangle is used determine the units of measure.

Rectangle Operations

The Microsoft Win32 application programming interface (API) provides several functions for working with rectangles.

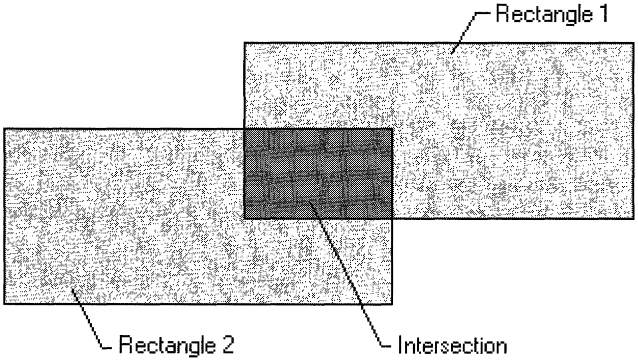
The **SetRect** function creates a rectangle, the **CopyRect** function makes a copy of a given rectangle, and the **SetRectEmpty** function creates an empty rectangle. An empty rectangle is any rectangle that has zero width, zero height, or both. The **IsRectEmpty** function determines whether a given rectangle is empty. The **EqualRect** function determines whether two rectangles are identical—that is, whether they have the same coordinates.

The **InflateRect** function increases or decreases the width or height of a rectangle, or both. It can add or remove width from both ends of the rectangle; it can add or remove height from both the top and bottom of the rectangle.

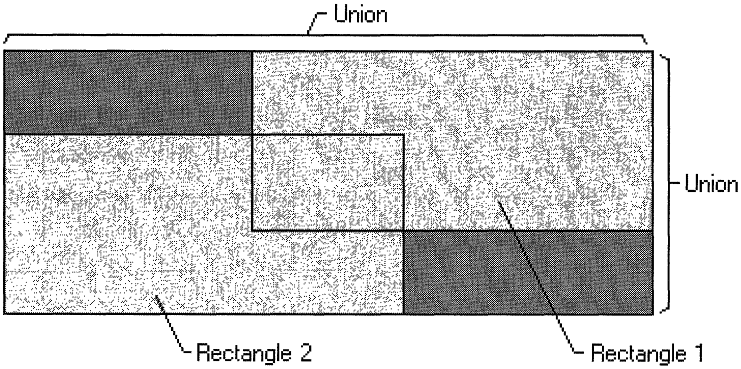
The **OffsetRect** function moves a rectangle by a given amount. It moves the rectangle by adding the given x-amount, y-amount, or x- and y-amounts to the corner coordinates.

The **PtInRect** function determines whether a given point lies within a given rectangle. The point is in the rectangle if it lies on the left or top side or is completely within the rectangle. The point is not in the rectangle if it lies on the right or bottom side.

The **IntersectRect** function creates a new rectangle that is the intersection of two existing rectangles, as shown in the following figure.



The **UnionRect** function creates a new rectangle that is the union of two existing rectangles, as shown in the following figure.



For information about functions that draw ellipses and polygons, see *Filled Shapes.*

Rectangle Reference

Rectangle Functions

CopyRect

The **CopyRect** function copies the coordinates of one rectangle to another.

```
BOOL CopyRect(
  LPRECT lprcDst,        // destination rectangle
  CONST RECT *lprcSrc    // source rectangle
);
```

Parameters

lprcDst

[out] Pointer to the **RECT** structure that receives the logical coordinates of the source rectangle.

lprcSrc

[in] Pointer to the **RECT** structure whose coordinates are to be copied.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Rectangles Overview, Rectangle Functions, **RECT**, **SetRect**, **SetRectEmpty**

EqualRect

The **EqualRect** function determines whether the two specified rectangles are equal by comparing the coordinates of their upper-left and lower-right corners.

```
BOOL EqualRect(
  CONST RECT *lprc1,  // first rectangle
  CONST RECT *lprc2   // second rectangle
);
```

Parameters

lprc1

[in] Pointer to a **RECT** structure that contains the logical coordinates of the first rectangle.

lprc2

[in] Pointer to a **RECT** structure that contains the logical coordinates of the second rectangle.

Return Values

If the two rectangles are identical, the return value is nonzero.

If the two rectangles are not identical, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Rectangles Overview, Rectangle Functions, **IsRectEmpty**, **PtInRect**, **RECT**

InflateRect

The **InflateRect** function increases or decreases the width and height of the specified rectangle. The **InflateRect** function adds *dx* units to the left and right ends of the rectangle and *dy* units to the top and bottom. The *dx* and *dy* parameters are signed values; positive values increase the width and height, and negative values decrease them.

```
BOOL InflateRect(
  LPRECT lprc,  // rectangle
  int dx,       // amount to adjust width
  int dy        // amount to adjust height
);
```

Parameters

lprc

[in/out] Pointer to the **RECT** structure that increases or decreases in size.

dx

[in] Specifies the amount to increase or decrease the rectangle width. This parameter must be negative to decrease the width.

dy

[in] Specifies the amount to increase or decrease the rectangle height. This parameter must be negative to decrease the height.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **IntersectRect**, **OffsetRect**, **RECT**, **UnionRect**

IntersectRect

The **IntersectRect** function calculates the intersection of two source rectangles and places the coordinates of the intersection rectangle into the destination rectangle. If the source rectangles do not intersect, an empty rectangle (in which all coordinates are set to zero) is placed into the destination rectangle.

```
BOOL IntersectRect(
  LPRECT lprcDst,          // intersection buffer
  CONST RECT *lprcSrc1,    // first rectangle
  CONST RECT *lprcSrc2     // second rectangle
);
```

Parameters

lprcDst

[out] Pointer to the **RECT** structure that is to receive the intersection of the rectangles pointed to by the *lprcSrc1* and *lprcSrc2* parameters. This parameter cannot be NULL.

lprcSrc1

[in] Pointer to the **RECT** structure that contains the first source rectangle.

lprcSrc2

[in] Pointer to the **RECT** structure that contains the second source rectangle.

Return Values

If the rectangles intersect, the return value is nonzero.

If the rectangles do not intersect, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **InflateRect**, **OffsetRect**, **RECT**, **UnionRect**

IsRectEmpty

The **IsRectEmpty** function determines whether the specified rectangle is empty. A empty rectangle is one that has no area; that is, the coordinate of the right side is less than or equal to the coordinate of the left side, or the coordinate of the bottom side is less than or equal to the coordinate of the top side.

```
BOOL IsRectEmpty(
  CONST RECT *lprc   // rectangle
);
```

Parameters

lprc

[in] Pointer to a **RECT** structure that contains the logical coordinates of the rectangle.

Return Values

If the rectangle is empty, the return value is nonzero.

If the rectangle is not empty, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **EqualRect**, **PtInRect**, **RECT**

OffsetRect

The **OffsetRect** function moves the specified rectangle by the specified offsets.

```
BOOL OffsetRect(
  LPRECT lprc,  // rectangle
  int dx,       // horizontal offset
  int dy        // vertical offset
);
```

Parameters

lprc

[in/out] Pointer to a **RECT** structure that contains the logical coordinates of the rectangle to be moved.

dx

[in] Specifies the amount to move the rectangle left or right. This parameter must be a negative value to move the rectangle to the left.

dy

[in] Specifies the amount to move the rectangle up or down. This parameter must be a negative value to move the rectangle up.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Rectangles Overview, Rectangle Functions, **InflateRect**, **IntersectRect**, **UnionRect**, **RECT**

PtInRect

The **PtInRect** function determines whether the specified point lies within the specified rectangle. A point is within a rectangle if it lies on the left or top side or is within all four sides. A point on the right or bottom side is considered outside the rectangle.

```
BOOL PtInRect(
  CONST RECT *lprc,  // rectangle
  POINT pt           // point
);
```

Parameters

lprc

[in] Pointer to a **RECT** structure that contains the specified rectangle.

pt

[in] Specifies a **POINT** structure that contains the specified point.

Return Values

If the specified point lies within the rectangle, the return value is nonzero.

If the specified point does not lie within the rectangle, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The rectangle must be normalized before **PtInRect** is called. That is, *lprc.right* must be greater than *lprc.left* and *lprc.bottom* must be greater than *lprc.top.* If the rectangle is not normalized, a point is never considered inside of the rectangle.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **EqualRect**, **IsRectEmpty**, **POINT**, **RECT**

SetRect

The **SetRect** function sets the coordinates of the specified rectangle. This is equivalent to assigning the left, top, right, and bottom arguments to the appropriate members of the **RECT** structure.

```
BOOL SetRect(
  LPRECT lprc,  // rectangle
  int xLeft,    // left side
  int yTop,     // top side
  int xRight,   // right side
  int yBottom   // bottom side
);
```

Parameters

lprc
[out] Pointer to the **RECT** structure that contains the rectangle to be set.

xLeft
[in] Specifies the x-coordinate of the rectangle's upper-left corner.

yTop
[in] Specifies the y-coordinate of the rectangle's upper-left corner.

xRight
[in] Specifies the x-coordinate of the rectangle's lower-right corner.

yBottom
[in] Specifies the y-coordinate of the rectangle's lower-right corner.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.

Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **CopyRect**, **SetRectEmpty**, **RECT**

SetRectEmpty

The **SetRectEmpty** function creates an empty rectangle in which all coordinates are set to zero.

```
BOOL SetRectEmpty(
  LPRECT lprc   // rectangle
);
```

Parameters

lprc

[out] Pointer to the **RECT** structure that contains the coordinates of the rectangle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **CopyRect**, **RECT**, **SetRect**

SubtractRect

The **SubtractRect** function obtains the coordinates of a rectangle determined by subtracting one rectangle from another.

```
BOOL SubtractRect(
  LPRECT lprcDst,          // destination rectangle
  CONST RECT *lprcSrc1,    // first rectangle
  CONST RECT *lprcSrc2     // second rectangle
);
```

Parameters

lprcDst

[out] Pointer to a **RECT** structure that receives the coordinates of the rectangle determined by subtracting the rectangle pointed to by *lprcSrc2* from the rectangle pointed to by *lprcSrc1*.

lprcSrc1

[in] Pointer to a **RECT** structure from which the function subtracts the rectangle pointed to by *lprcSrc2*.

lprcSrc2

[in] Pointer to a **RECT** structure that the function subtracts from the rectangle pointed to by *lprcSrc1*.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The function only subtracts the rectangle specified by *lprcSrc2* from the rectangle specified by *lprcSrc1* when the rectangles intersect completely in either the x- or y-direction. For example, if **lprcSrc1* has the coordinates (10,10,100,100) and **lprcSrc2* has the coordinates (50,50,150,150), the function sets the coordinates of the rectangle pointed to by *lprcDst* to (10,10,100,100). If **lprcSrc1* has the coordinates (10,10,100,100) and **lprcSrc2* has the coordinates (50,10,150,150), however, the function sets the coordinates of the rectangle pointed to by *lprcDst* to (10,10,50,100).

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

+ See Also

Rectangles Overview, Rectangle Functions, **IntersectRect**, **RECT**, **UnionRect**

UnionRect

The **UnionRect** function creates the union of two rectangles. The union is the smallest rectangle that contains both source rectangles.

```
BOOL UnionRect(
  LPRECT lprcDst,         // destination rectangle
  CONST RECT *lprcSrc1,   // first rectangle
  CONST RECT *lprcSrc2    // second rectangle
);
```

Parameters

lprcDst

[out] Pointer to the **RECT** structure that will receive a rectangle containing the rectangles pointed to by the *lprcSrc1* and *lprcSrc2* parameters.

lprcSrc1

[in] Pointer to the **RECT** structure that contains the first source rectangle.

lprcSrc2

[in] Pointer to the **RECT** structure that contains the second source rectangle.

Return Values

If the specified structure contains a nonempty rectangle, the return value is nonzero.

If the specified structure does not contain a nonempty rectangle, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The system ignores the dimensions of an empty rectangle—that is, a rectangle in which all coordinates are set to zero, so that it has no height or no width.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Rectangles Overview, Rectangle Functions, **InflateRect**, **IntersectRect**, **OffsetRect**, **RECT**

Rectangle Structures

POINT

The **POINT** structure defines the x- and y- coordinates of a point.

```
typedef struct tagPOINT {
  LONG x;
  LONG y;
} POINT, *PPOINT;
```

Members

x

Specifies the x-coordinate of the point.

y

Specifies the y-coordinate of the point.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in windef.h; include windows.h.

See Also

Rectangles Overview, Rectangle Structures, **ChildWindowFromPoint**, **GetBrushOrgEx**, **PtInRect**, **SetBrushOrgEx**, **WindowFromPoint**

POINTS

The **POINTS** structure defines the coordinates of a point.

```
typedef struct tagPOINTS {
  SHORT x;
  SHORT y;
} POINTS, *PPOINTS;
```

Members

x

Specifies the x-coordinate of the point.

y

Specifies the y-coordinate of the point.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in windef.h; include windows.h.

See Also

Rectangles Overview, Rectangle Structures, **ChildWindowFromPoint**, **PtInRect**, **WindowFromPoint**, **POINT**

RECT

The **RECT** structure defines the coordinates of the upper-left and lower-right corners of a rectangle.

```
typedef struct _RECT {
  LONG left;
  LONG top;
  LONG right;
  LONG bottom;
} RECT, *PRECT;
```

Members

left

Specifies the x-coordinate of the upper-left corner of the rectangle.

top

Specifies the y-coordinate of the upper-left corner of the rectangle.

right

Specifies the x-coordinate of the lower-right corner of the rectangle.

bottom

Specifies the y-coordinate of the lower-right corner of the rectangle.

Remarks

When **RECT** is passed to the **FillRect** function, the rectangle is filled up to, but not including, the right column and bottom row of pixels. This structure is identical to the **RECTL** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in windef.h; include windows.h.

See Also

Rectangles Overview, Rectangle Structures, **FillRect**, **RECTL**, **SMALL_RECT**

Rectangle Macros

MAKEPOINTS

The **MAKEPOINTS** macro converts a value that contains the x- and y-coordinates of a point into a **POINTS** structure.

```
POINTS MAKEPOINTS(
  DWORD dwValue  // coordinates of a point
);
```

Parameters

dwValue

Specifies the coordinates of a point. The x-coordinate is in the low-order word, and the y-coordinate is in the high-order word.

Return Values

The return value is a pointer to a **POINTS** structure.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Rectangles Overview, Rectangle Macros, **GetMessagePos**

POINTSTOPOINT

The **POINTSTOPOINT** macro copies the contents of a **POINTS** structure into a **POINT** structure.

```
POINTSTOPOINT(
  POINT pt,   // POINT structure
  POINTS pts // POINTS structure
);
```

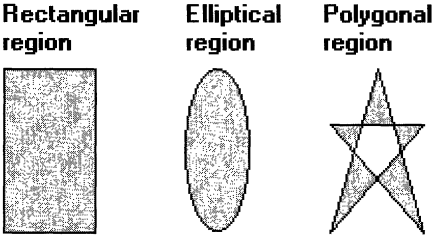
CHAPTER 19

Regions

A *region* is a rectangle, polygon, or ellipse (or a combination of two or more of these shapes) that can be filled, painted, inverted, framed, and used to perform hit testing (testing for the cursor location).

About Regions

Following are three types of regions that have been filled and framed.



Region Creation and Selection

An application creates a region by calling a function associated with a specific shape. The following table shows the function(s) associated with each of the standard shapes:

Shape	Function
Rectangular region	**CreateRectRgn**, **CreateRectRgnIndirect**, **SetRectRgn**
Rectangular region with rounded corners	**CreateRoundRectRgn**
Elliptical region	**CreateEllipticRgn**, **CreateEllipticRgnIndirect**
Polygonal region	**CreatePolygonRgn**, **CreatePolyPolygonRgn**

Each region creation function returns a handle that identifies the new region. An application can use this handle to select the region into a device context by calling the **SelectObject** function and supplying this handle as the second argument. After a region is selected into a device context, the application can perform various operations on it.

Region Operations

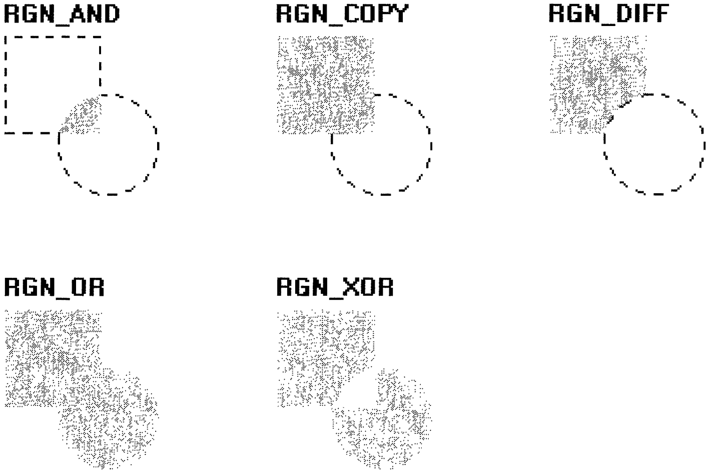
Applications can combine regions, compare them, paint or invert their interiors, draw a frame around them, retrieve their dimensions, and test whether the cursor lies within their boundaries.

Combining Regions

An application combines two regions by calling the **CombineRgn** function. Using this function, an application can combine the intersecting parts of two regions, all but the intersecting parts of two regions, the two original regions in their entirety, and so on. Following are five values that define the region combinations:

Value	Meaning
RGN_AND	The intersecting parts of two original regions define a new region.
RGN_COPY	A copy of the first (of the two original regions) defines a new region.
RGN_DIFF	The part of the first region that does not intersect the second defines a new region.
RGN_OR	The two original regions define a new region.
RGN_XOR	Those parts of the two original regions that do not overlap define a new region.

The following illustration shows the five possible combinations of a square and a circular region resulting from a call to **CombineRgn**.



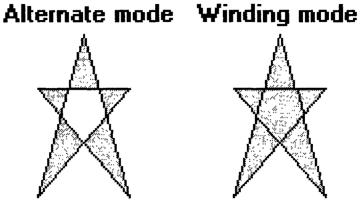
Comparing Regions

An application compares two regions to determine whether or not they are identical by calling the **EqualRgn** function. **EqualRgn** considers two regions identical if they are equal in size and shape.

Filling Regions

An application fills the interior of a region by calling the **FillRgn** function and supplying a handle that identifies a specific brush. When an application calls **FillRgn**, the system fills the region with the brush by using the current fill mode for the specified device context. There are two fill modes: alternate and winding. The application can set the fill mode for a device context by calling the **SetPolyFillMode** function. The application can retrieve the current fill mode for a device context by calling the **GetPolyFillMode** function.

The following illustration shows two identical regions: one filled using alternate mode and the other filled using winding mode.



Alternate mode Winding mode

Alternate Mode

To determine which pixels the system highlights when alternate mode is specified, perform the following test:

1. Select a pixel within the region's interior.
2. Draw an imaginary ray, in the positive x-direction, from that pixel towards infinity.
3. Each time the ray intersects a boundary line, increment a count value.

The system highlights the pixel if the count value is an odd number.

Winding Mode

To determine which pixels the system highlights when winding mode is specified, perform the following test:

1. Determine the direction in which each boundary line is drawn.
2. Select a pixel within the region's interior.
3. Draw an imaginary ray, in the positive x-direction, from the pixel toward infinity.
4. Each time the ray intersects a boundary line with a positive y-component, increment a count value. Each time the ray intersects a boundary line with a negative y-component, decrement the count value.

The system highlights the pixel if the count value is nonzero.

Painting Regions

An application fills the interior of a region by using the brush currently selected into a device context by the **PaintRgn** function. This function uses the current polygon fill modes (alternate and winding).

Inverting Regions

An application inverts the colors that appear within a region by calling the **InvertRgn** function. On monochrome displays, **InvertRgn** makes white pixels black and black pixels white. On color screens, this inversion is dependent on the type of technology used to generate the colors for the screen.

Framing Regions

An application draws a border around a region by calling the **FrameRgn** function and specifying the border width and brush pattern that the system uses when drawing the frame.

Retrieving a Bounding Rectangle

An application retrieves the dimensions of a region's bounding rectangle by calling the **GetRgnBox** function. If the region is rectangular, **GetRgnBox** returns the dimensions of the region. If the region is elliptical, the function returns the dimensions of the smallest rectangle that can be drawn around the ellipse. The long sides of the rectangle are the same length as the ellipse's major axis, and the short sides of the rectangle are the same length as the ellipse's minor axis. If the region is polygonal, **GetRgnBox** returns the dimensions of the smallest rectangle that can be drawn around the entire polygon.

Moving Regions

An application moves a region by calling the **OffsetRgn** function. The given offsets along the x-axis and y-axis determine the number of logical units to move left or right and up or down.

Hit Testing Regions

An application performs hit testing on regions to determine the coordinates of the current cursor position. Then it passes these coordinates—as well as a handle identifying the region—to the **PtInRegion** function. The cursor coordinates can be retrieved by processing the various mouse messages, such as **WM_LBUTTONDOWN, WM_LBUTTONUP, WM_RBUTTONDOWN, and WM_RBUTTONUP.** The return value for **PtInRegion** indicates whether the cursor position is within the given region.

Region Reference

Region Functions

CombineRgn

The **CombineRgn** function combines two regions and stores the result in a third region. The two regions are combined according to the specified mode.

```
int CombineRgn(
  HRGN hrgnDest,      // handle to destination region
  HRGN hrgnSrc1,      // handle to source region
  HRGN hrgnSrc2,      // handle to source region
  int fnCombineMode   // region combining mode
);
```

Parameters

hrgnDest
[in] Handle to a new region with dimensions defined by combining two other regions. (This region must exist before **CombineRgn** is called.)

hrgnSrc1
[in] Handle to the first of two regions to be combined.

hrgnSrc2
[in] Handle to the second of two regions to be combined.

fnCombineMode
[in] Specifies a mode indicating how the two regions will be combined. This parameter can be one of the following values:

Value	Description
RGN_AND	Creates the intersection of the two combined regions.
RGN_COPY	Creates a copy of the region identified by *hrgnSrc1*.
RGN_DIFF	Combines the parts of *hrgnSrc1* that are not part of *hrgnSrc2*.
RGN_OR	Creates the union of two combined regions.
RGN_XOR	Creates the union of two combined regions *except* for any overlapping areas.

Return Values

The return value specifies the type of the resulting region. It can be one of the following values:

Value	Meaning
COMPLEXREGION	The region is more than a single rectangle.
ERROR	No region is created.
NULLREGION	The region is empty.
SIMPLEREGION	The region is a single rectangle.

Remarks

The three regions need not be distinct. For example, the *hrgnSrc1* parameter can equal the *hrgnDest* parameter.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreatePolygonRgn**, **CreatePolyPolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**, **CreateRoundRectRgn**

CreateEllipticRgn

The **CreateEllipticRgn** function creates an elliptical region.

```
HRGN CreateEllipticRgn(
  int nLeftRect,    // x-coord of upper-left corner of rectangle
  int nTopRect,     // y-coord of upper-left corner of rectangle
  int nRightRect,   // x-coord of lower-right corner of rectangle
  int nBottomRect   // y-coord of lower-right corner of rectangle
);
```

Parameters

nLeftRect

[in] Specifies the x-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

nTopRect

[in] Specifies the y-coordinate of the upper-left corner of the bounding rectangle of the ellipse.

nRightRect
[in] Specifies the x-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

nBottomRect
[in] Specifies the y-coordinate of the lower-right corner of the bounding rectangle of the ellipse.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A bounding rectangle defines the size, shape, and orientation of the region: The long sides of the rectangle define the length of the ellipse's major axis; the short sides define the length of the ellipse's minor axis; and the center of the rectangle defines the intersection of the major and minor axes.

The coordinates of the bounding rectangle are specified in logical units.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **CreateEllipticRgnIndirect**, **DeleteObject**, **SelectObject**

CreateEllipticRgnIndirect

The **CreateEllipticRgnIndirect** function creates an elliptical region.

```
HRGN CreateEllipticRgnIndirect(
  CONST RECT *lprc   // bounding rectangle
);
```

Parameters

lprc

[in] Pointer to a **RECT** structure that contains the coordinates of the upper-left and lower-right corners of the bounding rectangle of the ellipse.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

A bounding rectangle defines the size, shape, and orientation of the region: The long sides of the rectangle define the length of the ellipse's major axis; the short sides define the length of the ellipse's minor axis; and the center of the rectangle defines the intersection of the major and minor axes.

The coordinates of the bounding rectangle are specified in logical units.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreateEllipticRgn**, **DeleteObject**, **RECT**, **SelectObject**

CreatePolygonRgn

The **CreatePolygonRgn** function creates a polygonal region.

```
HRGN CreatePolygonRgn(
  CONST POINT *lppt,    // array of points
  int cPoints,          // number of points in array
  int fnPolyFillMode    // polygon-filling mode
);
```

Parameters

lppt

[in] Pointer to an array of **POINT** structures that define the vertices of the polygon. The polygon is presumed closed. Each vertex can be specified only once.

cPoints

[in] Specifies the number of points in the array.

fnPolyFillMode

[in] Specifies the fill mode used to determine which pixels are in the region. This parameter can be one of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode (fills area between odd-numbered and even-numbered polygon sides on each scan line).
WINDING	Selects winding mode (fills any region with a nonzero winding value).

For more information about these modes, see the ***SetPolyFillMode*** function.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **CreatePolyPolygonRgn**, **DeleteObject**, **POINT**, **SelectObject**, **SetPolyFillMode**

CreatePolyPolygonRgn

The **CreatePolyPolygonRgn** function creates a region consisting of a series of polygons. The polygons can overlap.

```
HRGN CreatePolyPolygonRgn(
  CONST POINT *lppt,          // pointer to array of points
  CONST INT *lpPolyCounts,    // pointer to count of vertices
  int nCount,                 // number of integers in array
  int fnPolyFillMode          // polygon fill mode
);
```

Parameters

lppt

[in] Pointer to an array of **POINT** structures that define the vertices of the polygons. The polygons are specified consecutively. Each polygon is presumed closed and each vertex is specified only once.

lpPolyCounts

[in] Pointer to an array of integers, each of which specifies the number of points in one of the polygons in the array pointed to by *lppt*.

nCount

[in] Specifies the total number of integers in the array pointed to by *lpPolyCounts*.

fnPolyFillMode

[in] Specifies the fill mode used to determine which pixels are in the region. This parameter can be one of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode (fills area between odd-numbered and even-numbered polygon sides on each scan line).
WINDING	Selects winding mode (fills any region with a nonzero winding value).

For more information about these modes, see the **SetPolyFillMode** function.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreatePolygonRgn**, **DeleteObject**, **POINT**, **SelectObject**, **SetPolyFillMode**

CreateRectRgn

The **CreateRectRgn** function creates a rectangular region.

```
HRGN CreateRectRgn(
  int nLeftRect,   // x-coordinate of upper-left corner
  int nTopRect,    // y-coordinate of upper-left corner
  int nRightRect,  // x-coordinate of lower-right corner
  int nBottomRect  // y-coordinate of lower-right corner
);
```

Parameters

nLeftRect
[in] Specifies the x-coordinate of the upper-left corner of the region.

nTopRect
[in] Specifies the y-coordinate of the upper-left corner of the region.

nRightRect
[in] Specifies the x-coordinate of the lower-right corner of the region.

nBottomRect
[in] Specifies the y-coordinate of the lower-right corner of the region.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The region will be exclusive of the bottom and right edges.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreateRectRgnIndirect**, **CreateRoundRectRgn**, **DeleteObject**, **SelectObject**

CreateRectRgnIndirect

The **CreateRectRgnIndirect** function creates a rectangular region.

```
HRGN CreateRectRgnIndirect(
  CONST RECT *lprc   // rectangle
);
```

Parameters

lprc

[in] Pointer to a **RECT** structure that contains the coordinates of the upper-left and lower-right corners of the rectangle that defines the region.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The region will be exclusive of the bottom and right edges.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreateRectRgn**, **CreateRoundRectRgn**, **DeleteObject**, **RECT**, **SelectObject**

CreateRoundRectRgn

The **CreateRoundRectRgn** function creates a rectangular region with rounded corners.

```
HRGN CreateRoundRectRgn(
  int nLeftRect,      // x-coordinate of upper-left corner
  int nTopRect,       // y-coordinate of upper-left corner
  int nRightRect,     // x-coordinate of lower-right corner
  int nBottomRect,    // y-coordinate of lower-right corner
  int nWidthEllipse,  // height of ellipse
  int nHeightEllipse  // width of ellipse
);
```

Parameters

nLeftRect

[in] Specifies the x-coordinate of the upper-left corner of the region.

nTopRect

[in] Specifies the y-coordinate of the upper-left corner of the region.

nRightRect

[in] Specifies the x-coordinate of the lower-right corner of the region.

nBottomRect

[in] Specifies the y-coordinate of the lower-right corner of the region.

nWidthEllipse

[in] Specifies the width of the ellipse used to create the rounded corners.

nHeightEllipse

[in] Specifies the height of the ellipse used to create the rounded corners.

Return Values

If the function succeeds, the return value is the handle to the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreateRectRgn**, **CreateRectRgnIndirect**, **DeleteObject**, **SelectObject**

EqualRgn

The **EqualRgn** function checks the two specified regions to determine whether they are identical. The function considers two regions identical if they are equal in size and shape.

```
BOOL EqualRgn(
  HRGN hSrcRgn1,  // handle to first region
  HRGN hSrcRgn2   // handle to second region
);
```

Parameters

hSrcRgn1
[in] Handle to a region.

hSrcRgn2
[in] Handle to a region.

Return Values

If the two regions are equal, the return value is nonzero.

If the two regions are not equal, the return value is zero. A return value of ERROR means at least one of the region handles is invalid.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreateRectRgn**, **CreateRectRgnIndirect**

ExtCreateRegion

The **ExtCreateRegion** function creates a region from the specified region and transformation data.

```
HRGN ExtCreateRegion(
  CONST XFORM *lpXform,       // transformation data
  DWORD nCount,               // size of region data
  CONST RGNDATA *lpRgnData    // region data buffer
);
```

Parameters

lpXform

[in] Pointer to an **XFORM** structure that defines the transformation to be performed on the region. If this pointer is NULL, the identity transformation is used.

nCount

[in] Specifies the number of bytes pointed to by *lpRgnData*.

lpRgnData

[in] Pointer to a **RGNDATA** structure that contains the region data.

Return Values

If the function succeeds, the return value is the value of the region.

If the function fails, the return value is NULL.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

An application can retrieve data for a region by calling the **GetRegionData** function.

Windows 95/98: Regions are no longer limited to the 64-KB heap.

Windows 95/98: World transforms that involve either shearing or rotations are not supported. **ExtCreateRegion** fails if the transformation matrix is anything other than a scaling or translation of the region.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **GetRegionData**, **RGNDATA**, **XFORM**

FillRgn

The **FillRgn** function fills a region by using the specified brush.

```
BOOL FillRgn(
  HDC hdc,      // handle to device context
  HRGN hrgn,    // handle to region to be filled
  HBRUSH hbr    // handle to brush used to fill the region
);
```

Parameters

hdc

[in] Handle to the device context.

hrgn

[in] Handle to the region to be filled. The region's coordinates are presumed to be in logical units.

hbr

[in] Handle to the brush to be used to fill the region.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **CreateBrushIndirect**, **CreateDIBPatternBrush**, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**, **PaintRgn**

FrameRgn

The **FrameRgn** function draws a border around the specified region by using the specified brush.

```
BOOL FrameRgn(
  HDC hdc,        // handle to device context
  HRGN hrgn,      // handle to region to be framed
  HBRUSH hbr,     // handle to brush used to draw border
  int nWidth,     // width of region frame
  int nHeight     // height of region frame
);
```

Parameters

hdc
[in] Handle to the device context.

hrgn
[in] Handle to the region to be enclosed in a border. The region's coordinates are presumed to be in logical units.

hbr
[in] Handle to the brush to be used to draw the border.

nWidth
[in] Specifies the width, in logical units, of vertical brush strokes.

nHeight
[in] Specifies the height, in logical units, of horizontal brush strokes.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **FillRgn**, **PaintRgn**

GetPolyFillMode

The **GetPolyFillMode** function retrieves the current polygon fill mode.

```
int GetPolyFillMode(
  HDC hdc   // handle to device context
);
```

Parameters

hdc
[in] Handle to the device context.

Return Values

If the function succeeds, the return value specifies the polygon fill mode, which can be one of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode (fills area between odd-numbered and even-numbered polygon sides on each scan line).
WINDING	Selects winding mode (fills any region with a nonzero winding value).

If an error occurs, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **SetPolyFillMode**

GetRegionData

The **GetRegionData** function fills the specified buffer with data describing a region. This data includes the dimensions of the rectangles that make up the region.

```
DWORD GetRegionData(
  HRGN hRgn,            // handle to region
  DWORD dwCount,        // size of region data buffer
  LPRGNDATA lpRgnData   // region data buffer
);
```

Parameters

hRgn

[in] Handle to the region.

dwCount

[in] Specifies the size, in bytes, of the *lpRgnData* buffer.

lpRgnData

[out] Pointer to a **RGNDATA** structure that receives the information. If this parameter is NULL, the return value contains the number of bytes needed for the region data.

Return Values

If the function succeeds and *dwCount* specifies an adequate number of bytes, the return value is always *dwCount*. If *dwCount* is too small or the function fails, the return value is 0. If *lpRgnData* is NULL, the return value is the required number of bytes.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The **GetRegionData** function is used in conjunction with the **ExtCreateRegion** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **ExtCreateRegion**, **RGNDATA**

GetRgnBox

The **GetRgnBox** function retrieves the bounding rectangle of the specified region.

```
int GetRgnBox(
  HRGN hrgn,    // handle to a region
  LPRECT lprc   // bounding rectangle
);
```

Parameters

hrgn

[in] Handle to the region.

lprc

[out] Pointer to a **RECT** structure that receives the bounding rectangle.

Return Values

The return value specifies the region's complexity. It can be one of the following values:

Value	Meaning
COMPLEXREGION	Region is more than a single rectangle.
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.

If the *hrgn* parameter does not identify a valid region, the return value is zero.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **RECT**

InvertRgn

The **InvertRgn** function inverts the colors in the specified region.

```
BOOL InvertRgn(
  HDC hdc,      // handle to device context
  HRGN hrgn     // handle to region to be inverted
);
```

Parameters

hdc

[in] Handle to the device context.

hrgn

[in] Handle to the region for which colors are inverted. The region's coordinates are presumed to be logical coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

On monochrome screens, the **InvertRgn** function makes white pixels black and black pixels white. On color screens, this inversion is dependent on the type of technology used to generate the colors for the screen.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **FillRgn**, **PaintRgn**

OffsetRgn

The **OffsetRgn** function moves a region by the specified offsets.

```
int OffsetRgn(
  HRGN hrgn,        // handle to region
  int nXOffset,     // offset along x-axis
  int nYOffset      // offset along y-axis
);
```

Parameters

hrgn
  [in] Handle to the region to be moved.

nXOffset
  [in] Specifies the number of logical units to move left or right.

nYOffset
  [in] Specifies the number of logical units to move up or down.

Return Values

The return value specifies the new region's complexity. It can be one of the following values:

Value	Meaning
COMPLEXREGION	Region is more than one rectangle.
ERROR	An error occurred; region is unaffected.
NULLREGION	Region is empty.
SIMPLEREGION	Region is a single rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions

PaintRgn

The **PaintRgn** function paints the specified region by using the brush currently selected into the device context.

```
BOOL PaintRgn(
  HDC hdc,     // handle to device context
  HRGN hrgn    // handle to region to be painted
);
```

Parameters

hdc

[in] Handle to the device context.

hrgn

[in] Handle to the region to be filled. The region's coordinates are presumed to be logical coordinates.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **FillRgn**

PtInRegion

The **PtInRegion** function determines whether the specified point is inside the specified region.

```
BOOL PtInRegion(
  HRGN hrgn,  // handle to region
  int X,      // x-coordinate of point
  int Y       // y-coordinate of point
);
```

Parameters

hrgn

[in] Handle to the region to be examined.

X

[in] Specifies the x-coordinate of the point.

Y

[in] Specifies the y-coordinate of the point.

Return Values

If the specified point is in the region, the return value is nonzero.

If the specified point is not in the region, the return value is zero.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **RectInRegion**

RectInRegion

The **RectInRegion** function determines whether any part of the specified rectangle is within the boundaries of a region.

```
BOOL RectInRegion(
  HRGN hrgn,          // handle to region
  CONST RECT *lprc    // pointer to rectangle
);
```

Parameters

hrgn

[in] Handle to the region.

lprc

[in] Pointer to a **RECT** structure containing the coordinates of the rectangle. The lower and right edges of the rectangle are not included.

Return Values

If any part of the specified rectangle lies within the boundaries of the region, the return value is nonzero.

If no part of the specified rectangle lies within the boundaries of the region, the return value is zero.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **PtInRegion**, **RECT**

SetPolyFillMode

The **SetPolyFillMode** function sets the polygon fill mode for functions that fill polygons.

```
int SetPolyFillMode(
  HDC hdc,              // handle to device context
  int iPolyFillMode     // polygon fill mode
);
```

Parameters

hdc

[in] Handle to the device context.

iPolyFillMode

[in] Specifies the new fill mode. This parameter can be one of the following values:

Value	Meaning
ALTERNATE	Selects alternate mode (fills the area between odd-numbered and even-numbered polygon sides on each scan line).
WINDING	Selects winding mode (fills any region with a nonzero winding value).

Return Values

The return value specifies the previous filling mode. If an error occurs, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

In general, the modes differ only in cases where a complex, overlapping polygon must be filled (for example, a five-sided polygon that forms a five-pointed star with a pentagon in the center). In such cases, ALTERNATE mode fills every other enclosed region within the polygon (that is, the points of the star), but WINDING mode fills all regions (that is, the points and the pentagon).

When the fill mode is ALTERNATE, GDI fills the area between odd-numbered and even-numbered polygon sides on each scan line. That is, GDI fills the area between the first and second side, between the third and fourth side, and so on.

When the fill mode is WINDING, GDI fills any region that has a nonzero winding value. This value is defined as the number of times a pen used to draw the polygon would go around the region. The direction of each edge of the polygon is important.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

+ See Also

Regions Overview, Region Functions, **GetPolyFillMode**

SetRectRgn

The **SetRectRgn** function changes a region into a rectangular region with the specified coordinates.

```
BOOL SetRectRgn(
  HRGN hrgn,         // handle to region
  int nLeftRect,     // x-coordinate of upper-left corner of rectangle
  int nTopRect,      // y-coordinate of upper-left corner of rectangle
  int nRightRect,    // x-coordinate of lower-right corner of rectangle
  int nBottomRect    // y-coordinate of lower-right corner of rectangle
);
```

Parameters

hrgn

[in] Handle to the region.

nLeftRect

[in] Specifies the x-coordinate of the upper-left corner of the rectangular region.

nTopRect

[in] Specifies the y-coordinate of the upper-left corner of the rectangular region.

nRightRect

[in] Specifies the x-coordinate of the lower-right corner of the rectangular region.

nBottomRect

[in] Specifies the y-coordinate of the lower-right corner of the rectangular region.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Windows NT/2000: To get extended error information, call **GetLastError**.

Remarks

The region does not include the lower and right boundaries of the rectangle.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 2.0 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Regions Overview, Region Functions, **CreateRectRgn**

Region Structures

RGNDATA

The **RGNDATA** structure contains a header and an array of rectangles that compose a region. The rectangles are sorted top to bottom, left to right. They do not overlap.

```
typedef struct _RGNDATA {
    RGNDATAHEADER rdh;
    char          Buffer[1];
} RGNDATA, *PRGNDATA;
```

Members

rdh

Specifies a **RGNDATAHEADER** structure. The members of this structure specify the type of region (whether it is rectangular or trapezoidal), the number of rectangles that make up the region, the size of the buffer that contains the rectangle structures, and so on.

Buffer

Specifies an arbitrary-size buffer that contains the **RECT** structures that make up the region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

See Also

Regions Overview, Region Structures, **RECT**, **RGNDATAHEADER**

RGNDATAHEADER

The **RGNDATAHEADER** structure describes the data returned by the **GetRegionData** function.

```
typedef struct _RGNDATAHEADER {
    DWORD dwSize;
    DWORD iType;
    DWORD nCount;
    DWORD nRgnSize;
    RECT  rcBound;
} RGNDATAHEADER, *PRGNDATAHEADER;
```

Members

dwSize
Specifies the size, in bytes, of the header.

iType
Specifies the type of region. This value must be RDH_RECTANGLES.

nCount
Specifies the number of rectangles that make up the region.

nRgnSize
Specifies the size of the buffer required to receive the **RECT** structure that specifies the coordinates of the rectangles that make up the region. If the size is not known, this member can be zero.

rcBound
Specifies a bounding rectangle for the region.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Regions Overview, Region Structures, **GetRegionData**, **RECT**, **RGNDATA**

APPENDIX A

Index A: Elements Grouped by Technology

The indexes in Part 3 make finding information in the Win32 Library volumes as easy as possible. Rather than cluttering the Table of Contents with information about individual programmatic elements (and thereby making the TOC uselessly jumbled), I've created these indexes and put them in the back of each volume. With these indexes, you'll be able to locate the programmatic element you're interested in—and see where it fits within the volumes and their technologies—quickly and easily.

Also, to keep you informed and up-to-date about Microsoft technologies, I've created a live Web-based document that maps Microsoft technologies to the locations where you can get more information about them. This link gets you to the live index of technologies: *www.iseminger.com/winprs/technologies*

As always, send me feedback if you can think of ways to improve this section. I can't guarantee a reply, but I'll read it, and if others can benefit, I'll incorporate the idea into future volumes.

APPENDIX B

Index B: Volume 1, Elements Listed Alphabetically

APPENDIX B

Index B: Volume 2, Elements Listed Alphabetically

APPENDIX B

Index B: Volume 3, Elements Listed Alphabetically

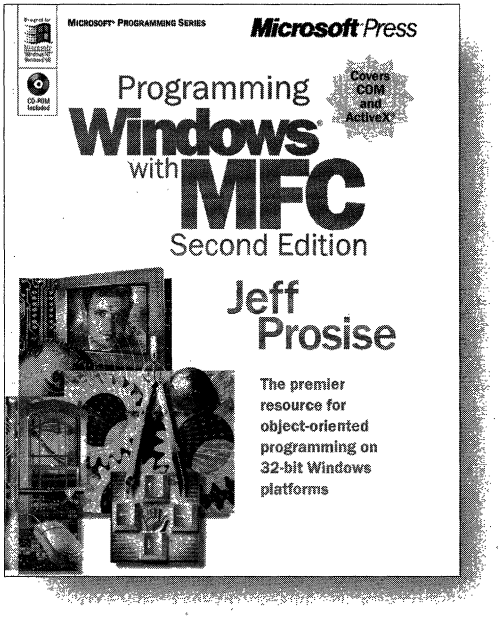
APPENDIX B

Index B: Volume 4, Elements Listed Alphabetically

APPENDIX B

Index B: Volume 5, Elements Listed Alphabetically

Petzold for the **MFC programmer!**


MICROSOFT® PROGRAMMING SERIES
Microsoft Press
CD-ROM Included
Programming
Windows®
with
MFC
Second Edition
Covers COM and ActiveX®
Jeff Prosise
The premier resource for object-oriented programming on 32-bit Windows platforms

U.S.A.	**$59.99**
U.K.	£56.99 [V.A.T. included]
Canada	$89.99

ISBN 1-57231-695-0

Expanding what's widely considered the definitive exposition of Microsoft's powerful C++ class library for the Windows API, PROGRAMMING WINDOWS® WITH MFC, Second Edition, fully updates the classic original with all-new coverage of COM, OLE, and ActiveX.® Author Jeff Prosise deftly builds your comprehension of underlying concepts and essential techniques for MFC programming with unparalleled expertise—once again delivering the consummate resource for rapid, object-oriented development on 32-bit Windows platforms.

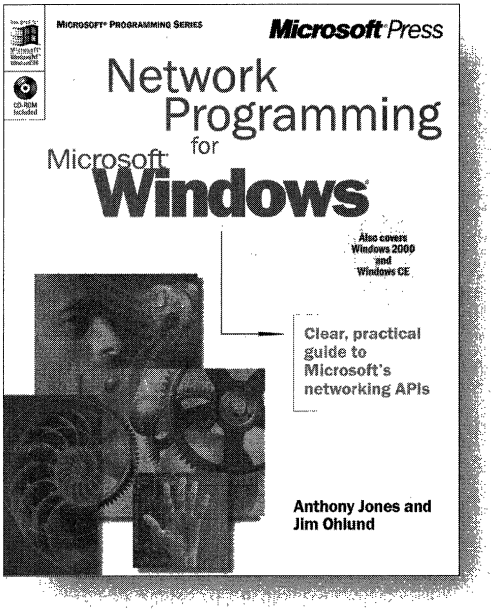
Microsoft Press® products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft® Sales Office, or visit our Web site at mspress.microsoft.com. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the U.S. (in Canada, call 1-800-268-2222).

Prices and availability dates are subject to change.

Microsoft®

mspress.microsoft.com

Here they are in one place— *practical, detailed explanations* of the Microsoft **networking APIs!**


MICROSOFT® PROGRAMMING SERIES
Microsoft Press
Network Programming for Microsoft Windows
Also covers Windows 2000 and Windows CE
Clear, practical guide to Microsoft's networking APIs
Anthony Jones and Jim Ohlund
CD-ROM Included

U.S.A.	**$49.99**
U.K.	£46.99 [V.A.T. included]
Canada	$74.99

ISBN 0-7356-0560-2

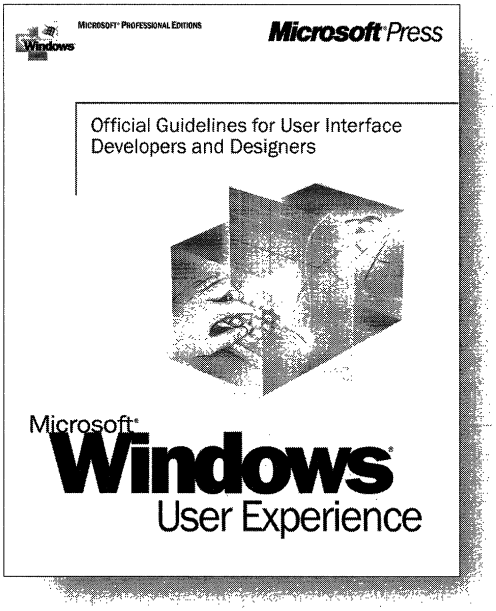
Microsoft has developed many exciting networking technologies, but until now no single source has described how to use them with older, and even some newer, application programming interfaces (APIs). NETWORK PROGRAMMING FOR MICROSOFT® WINDOWS® is the only book that provides definitive, hands-on coverage of how to use legacy networking APIs, such as NetBIOS, on 32-bit platforms, plus recent networking APIs such as Winsock 2 and Remote Access Service (RAS).

Microsoft Press® products are available worldwide wherever quality computer books are sold. For more information, contact your book or computer retailer, software reseller, or local Microsoft Sales Office, or visit our Web site at mspress.microsoft.com. To locate your nearest source for Microsoft Press products, or to order directly, call 1-800-MSPRESS in the U.S. (in Canada, call 1-800-268-2222).

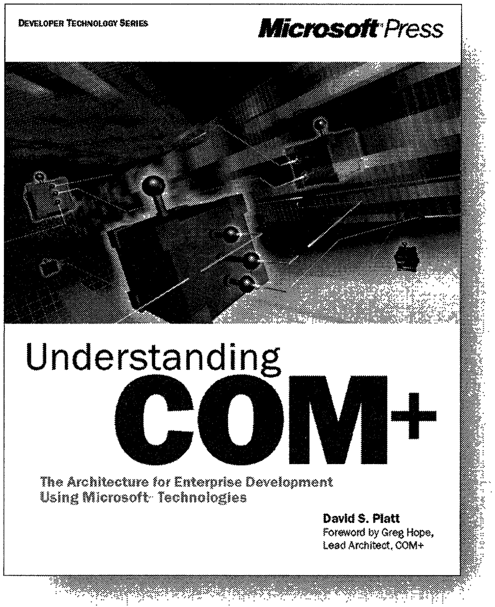
Prices and availability dates are subject to change.

Microsoft®

mspress.microsoft.com



MICROSOFT® PROFESSIONAL EDITIONS
Windows
Microsoft Press
Official Guidelines for User Interface Developers and Designers
Microsoft®
Windows®
User Experience


DEVELOPER TECHNOLOGY SERIES
Microsoft Press
Understanding
COM+
The Architecture for Enterprise Development Using Microsoft Technologies
David S. Platt
Foreword by Greg Hope, Lead Architect, COM+

Microsoft® Windows® GDI

This essential Windows 2000 and Windows 98/ Windows 95 reference volume is part of the five-volume Microsoft Win32® Developer's Reference Library. *In its printed form, this material is portable, easy to use, and easy to browse—a highly condensed, completely indexed, intelligently organized complement to the information available on line and through the Microsoft Developer Network (MSDN). Each volume includes an overview of the five-volume library, two appendixes of programming elements, and tips on how and where to find other Microsoft developer reference resources you may need.*

Microsoft Windows GDI

This volume provides complete reference materials about the services provided by the Windows GDI (Graphical Device Interface), including bitmaps, brushes, clipping, colors, coordinate spaces and transformations, device contexts, filled shapes, lines and curves, metafiles, painting and drawing, paths, pens, rectangles, and regions.

Part No. 097-0002308

Microsoft®