



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®
User Interface

BASED ON
msdn™ library

Microsoft®

The essential reference to Win32®
technologies and APIs

David Iseminger
Series Editor

Microsoft®
Windows®
User Interface

BASED ON
msdn™ library

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

Copyright © 2000 by Microsoft Corporation; portions © 2000 by David Iseminger.

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

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.

BackOffice, FrontPage, Microsoft, Microsoft Press, MSDN, Visual Basic, Visual C++, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, Win32, Windows, and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

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. 097-0002307

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 | 1 |
| How the Win32 Library Is Structured..... | 2 |
| How the Win32 Library Is Designed | 3 |
| Chapter 2: What's in This Volume?..... | 5 |
| Controls..... | 5 |
| Resources | 6 |
| User Input | 6 |
| Windowing..... | 7 |
| Chapter 3: Using Microsoft Reference Resources | 9 |
| The Microsoft Developer Network (MSDN) | 10 |
| Comparing MSDN and MSDN Online..... | 10 |
| MSDN Subscriptions | 11 |
| MSDN Library Subscription..... | 13 |
| MSDN Professional Subscription..... | 13 |
| MSDN Universal Subscription | 14 |
| Purchasing an MSDN Subscription | 14 |
| Using MSDN | 15 |
| Exploring MSDN..... | 16 |
| Quick Tips | 19 |
| Using MSDN Online..... | 19 |
| Exploring MSDN Online..... | 21 |
| MSDN Online Features | 22 |
| MSDN Online Registered Users..... | 26 |
| The Windows Programming Reference Series..... | 27 |
| Chapter 4: Finding the Developer Resources You Need..... | 29 |
| Developer Support | 29 |
| Online Resources | 31 |
| Learning Products | 32 |
| Conferences..... | 34 |
| Other Resources | 35 |

| | |
|--|-----------|
| Chapter 5: Getting the Most Out of Win32 Technologies: Part 2 | 37 |
| Avoiding Invalid Validation..... | 37 |
| Working with Handle-Based Objects | 38 |
| Verify Correlated Parameters..... | 39 |
| Limits of Exception Handling..... | 40 |
| Kernel mode NULL dereference is unsafe, even when protected by try- except..... | 40 |
| Use __leave in the try block of a try-finally | 41 |
| Ramifications of returning from a finally block | 42 |
| Be wary of execution order | 43 |
| Avoid relying on exceptions instead of correct validation | 43 |
| Alternate Code Paths..... | 44 |
| Trusted Data Sources | 45 |
| Solutions Summary | 47 |
| Chapter 6: Controls | 49 |
| Controls..... | 49 |
| About Controls..... | 49 |
| Predefined Controls | 49 |
| Control Reference | 50 |
| Control Messages | 50 |
| Buttons | 52 |
| About Buttons | 52 |
| Button Reference | 53 |
| Button Functions | 53 |
| Button Messages..... | 56 |
| Button Styles | 69 |
| Combo Boxes | 72 |
| About Combo Boxes..... | 72 |
| Combo-Box Types and Styles | 72 |
| Combo-Box Reference..... | 73 |
| Combo-Box Functions..... | 73 |
| Combo-Box Structures | 77 |
| Combo-Box Messages..... | 84 |
| Combo-Box Styles..... | 130 |
| Edit Controls | 131 |
| About Edit Controls | 132 |
| Getting Information About Edit Control Programming Elements | 132 |
| Rich-Edit Controls | 132 |

| | |
|--|------------|
| About Rich-Edit Controls | 133 |
| Getting More Information About Rich-Edit Controls..... | 133 |
| Scroll Bars..... | 134 |
| About Scroll Bars..... | 134 |
| Scroll-Bar Reference..... | 134 |
| Scroll-Bar Functions..... | 134 |
| Scroll-Bar Structures | 154 |
| Scroll-Bar Messages | 157 |
| Scroll-Bar Control Styles | 170 |
| Static Controls..... | 171 |
| About Static Controls..... | 171 |
| Static-Control Types..... | 171 |
| Static-Control Reference..... | 173 |
| Static-Control Messages | 173 |
| Static-Control Styles..... | 181 |
| Chapter 7: Resources..... | 185 |
| Resources | 185 |
| About Resources | 185 |
| Finding and Loading Resources | 186 |
| Adding, Deleting, and Replacing Resources | 187 |
| Enumerating Resources | 187 |
| Resource File Formats | 187 |
| Getting More Information About Resources..... | 190 |
| Carets | 190 |
| About Carets | 190 |
| Caret Visibility | 191 |
| Caret Blink Time | 191 |
| Caret Position | 191 |
| Removing a Caret..... | 191 |
| Caret Reference..... | 192 |
| Caret Functions..... | 192 |
| Cursors | 199 |
| About Cursors | 199 |
| Cursor Reference | 200 |
| Cursor Functions..... | 200 |
| Cursor Structures | 216 |
| Cursor Messages..... | 216 |
| Icons | 218 |
| About Icons..... | 218 |

| | |
|--|------------|
| Icon Reference | 218 |
| Icon Functions | 218 |
| Icon Structures | 239 |
| Menus | 244 |
| About Menus | 244 |
| Menu Bars and Menus | 244 |
| Menu Reference | 246 |
| Menu Functions | 246 |
| Menu Structures | 297 |
| Menu Messages | 311 |
| Strings | 321 |
| About Strings | 321 |
| Win32 String Functions | 321 |
| String Resources | 322 |
| String Reference | 323 |
| String Functions | 323 |
| Chapter 8: User Input..... | 369 |
| Common Dialog-Box Library | 369 |
| About Common Dialog Boxes | 369 |
| Dialog-Box Types | 369 |
| Getting More Information About Common Dialog Boxes | 371 |
| Mouse Input | 371 |
| About Mouse Input | 371 |
| Mouse-Input Reference..... | 372 |
| Mouse-Input Functions..... | 372 |
| Mouse-Input Structures | 385 |
| Mouse-Input Messages | 387 |
| Mouse-Input Macros..... | 437 |
| Keyboard Accelerators | 442 |
| About Keyboard Accelerators..... | 442 |
| Accelerator Tables..... | 442 |
| Accelerator Table Creation..... | 443 |
| Accelerator Keystroke Assignments..... | 444 |
| Accelerators and Menus | 445 |
| UI State..... | 445 |
| Keyboard Accelerator Reference | 446 |
| Keyboard Accelerator Functions | 446 |
| Keyboard Accelerator Structures..... | 452 |
| Keyboard Accelerator Messages | 453 |

| | |
|---|------------|
| Keyboard Input..... | 466 |
| About Keyboard Input | 466 |
| Keyboard-Input Model | 466 |
| Keyboard-Input Reference..... | 467 |
| Keyboard-Input Functions..... | 467 |
| Keyboard-Input Structures..... | 509 |
| Keyboard-Input Messages | 517 |
| Chapter 9: Windowing..... | 537 |
| Dialog Boxes | 537 |
| About Dialog Boxes..... | 537 |
| Dialog Box Reference..... | 537 |
| Dialog Box Functions | 537 |
| Dialog Box Structures | 582 |
| Dialog Box Messages | 595 |
| Messages and Message Queues..... | 603 |
| About Messages and Message Queues | 604 |
| Win32 Messages..... | 604 |
| Message Types | 604 |
| Message Routing | 606 |
| Message Handling | 608 |
| Message Filtering | 610 |
| Posting and Sending Messages..... | 611 |
| Message Deadlocks | 612 |
| Broadcasting Messages..... | 612 |
| Query Messages..... | 613 |
| Message and Message Queue Reference | 614 |
| Message and Message Queue Functions | 614 |
| Message and Message Queue Structures | 645 |
| Message and Message Queue Messages..... | 646 |
| Multiple Document Interface | 648 |
| About the Multiple Document Interface | 648 |
| Frame, Client, and Child Windows | 649 |
| Child Window Creation..... | 650 |
| Child Window Activation..... | 650 |
| Multiple Document Menus | 650 |
| Multiple Document Accelerators..... | 651 |
| Child Window Size and Arrangement..... | 651 |
| Icon Title Windows | 652 |
| Child Window Data | 652 |

| | |
|---|------------|
| Multiple Document Interface Reference | 653 |
| Multiple Document Interface Functions | 653 |
| Multiple Document Interface Structures | 659 |
| Multiple Document Interface Messages | 661 |
| Timers | 673 |
| About Timers | 673 |
| Timer Operations | 673 |
| High-Resolution Timer | 674 |
| Timer Reference | 674 |
| Timer Functions | 674 |
| Timer Messages | 679 |
| Window Classes | 680 |
| About Window Classes | 680 |
| Types of Window Classes | 681 |
| Getting More Information About Window Classes | 681 |
| Window Procedures | 681 |
| About Window Procedures | 681 |
| Window Procedure Reference | 682 |
| Window Procedure Functions | 682 |
| Window Properties | 686 |
| About Window Properties | 686 |
| Assigning Window Properties | 686 |
| Enumerating Window Properties | 686 |
| Window Property Reference | 687 |
| Window Property Functions | 687 |
| Windows | 694 |
| About Windows | 694 |
| Desktop Window | 694 |
| Application Windows | 695 |
| Getting More Information About Windows | 697 |
| Appendix A | 699 |
| Appendix B | 705 |

CHAPTER 1

Introduction

Welcome to the *Microsoft Win32 Developer's Reference Library*, your comprehensive reference guide to the Win32 development environment. This pack, 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 between the three levels of MSDN subscription, 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 find easily the particular programming element for which you're looking. 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 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 are 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 gather easily 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 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 explanation 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 find quickly the information you need.

How the Win32 Library Is Designed

The Win32 Library and all packs in the Windows Programming Reference Series are 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 a feel that are consistent with their electronic means of disseminating Microsoft reference information. 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 respective function reference pages.

The reason for maintaining such integration is simple: to make it easy for you—the developer of Windows applications—to use the tools and get the ongoing information you need to create quality programs. By providing a “common interface” among reference resources, your familiarity with the Win32 Library reference material can be applied immediately 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 too, 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 instead of figuring out how to use your tools. The structure and design of the Win32 Library provide you with a comprehensive, pre-sharpened toolset to build compelling Windows applications.

CHAPTER 2

What's in This Volume?

Just like the first volume, this second volume of the *Microsoft Win32 Developer's Reference Library—Volume 2: Windows User Interface*—contains reference material that pertains to a certain area of the Win32 programming environment: in this case, reference information about the Windows User Interface. Almost every application that's written to run on the Windows group of operating systems uses the Windows User Interface, and that makes this volume an important part of the core programming reference needed by any Windows application programmer.

Let's get specific about this user interface. There are all sorts of different user interface elements, but there are certain basic, primitive user interface elements that constitute the building blocks upon which Windows applications are built. These basic user interface elements are rounded out in their entirety in the following list:

- Controls
- Resources
- User Input
- Windowing

Even a quick look at this list of four basic elements begs a more detailed description. As done with each volume's Chapter 2, let's go into more detail about each of these categories.

Controls

Controls are programmatic elements that enable applications to perform simple input and output, and are generally used to get feedback from users through the use of dialog boxes. Controls are found in almost every Windows application and are as follows:

- Buttons
- Combo Boxes
- Edit Controls
- List Boxes
- Rich Edit Controls
- Scroll Bars
- Static Controls

Because these controls are as common as they are (try using a dialog box without buttons), descriptions of each aren't provided here; instead, suffice it to say that these controls are the base programming tools used to get feedback from users in dialog boxes and other places.

Resources

Resources can be used by application developers to enable users that interact with their applications to facilitate either communication or the exchange of information between application and user. Resources come in two flavors: standard resources and custom (application-defined) resources. Technically, a resource is binary data that you can add to the executable file of a Win32 application, but perhaps the most effective way to introduce Windows User Interface resources is to list the standard resources available in Win32:

- Carets
- Cursors
- Icons
- Menus
- Strings

User Input

The programmatic elements and reference information that support user input are reasonably self-explanatory; they support the capability of applications to accept and manipulate user input from various devices. User input incorporates the following areas of programming reference:

- Common Dialog Box Library
- Mouse Input
- Keyboard Accelerators
- Keyboard Input

Windowing

To develop applications on the Windows platform, you generally have to provide the basic programming code behind creating, manipulating, and maintaining windows. The section devoted to providing programmatic reference for such windowing activities is termed aptly Windowing. The following list outlines the technologies or user interface material geared toward providing windowing reference:

- Dialog Boxes
- Messages and Message Queues
- Multiple-Document Interfaces (MDIs)
- Timers
- Window Classes
- Window Procedures
- Window Properties
- Windows

In Part 2 of this volume, each of these four basic elements and their respective subcategories is explained in detail, with overview and explanatory material at the beginning of each, and detailed reference material following. Each section provides a comprehensive reference for you to thumb through as you create your application's user interface.

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, too; not because there is information that isn't needed, but 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, time consuming, and 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 get there quickly and efficiently).

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 either have worked with MSDN or 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 had experience with MSDN Online, you might 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 between 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 the most out of MSDN. 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 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 one 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 difference 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, as possible. These strengths and medium 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 becomes available, without them 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 linked also to the Internet as a browser-like window. To coordinate further MSDN with the immediacy of the Internet, MSDN Online has dedicated a section of the site to MSDN subscribers that enable 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 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 or 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 you will notice that is shared between MSDN and MSDN Online is the interface—the interfaces 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 had 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 probably will continue to use MSDN Online and the additional features provided with your MSDN subscription.

MSDN Subscriptions

If you’re wondering whether you might benefit from a subscription to MSDN, but not 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 it even chances giving you an approximation on what each subscription level will set you back.

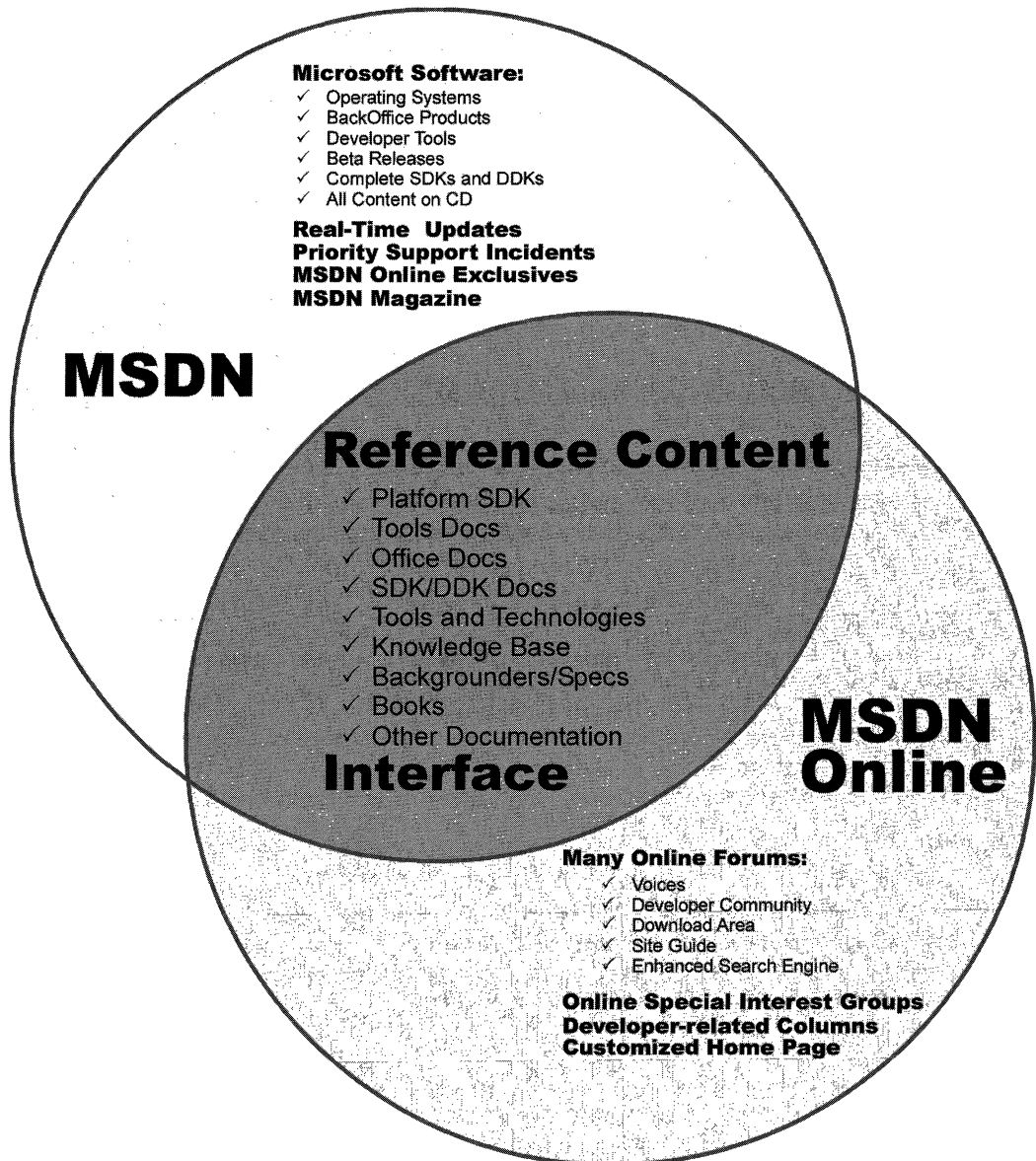


Figure 3-1: The similarities and differences in coverage between MSDN and MSDN Online.

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 might 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 get:

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

MSDN Professional Subscription

The MSDN 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 4.0 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 MSDN 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 it 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 a current customer renewing your subscription, you again get a break in the box, this time in the nature of a \$200 rebate. You get that break also if you're an existing Library subscriber who's upgrading to a Professional subscription.

The MSDN Universal subscription takes a big jump in price, sitting at \$2,499. If you're upgrading from the Professional subscription, the price drops to \$1,999; 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 both 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, whoever's in charge of purchasing should look into the Microsoft Open License program; the Open License program gives purchasing breaks for customers who 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 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 the reseller carries 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 move 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.

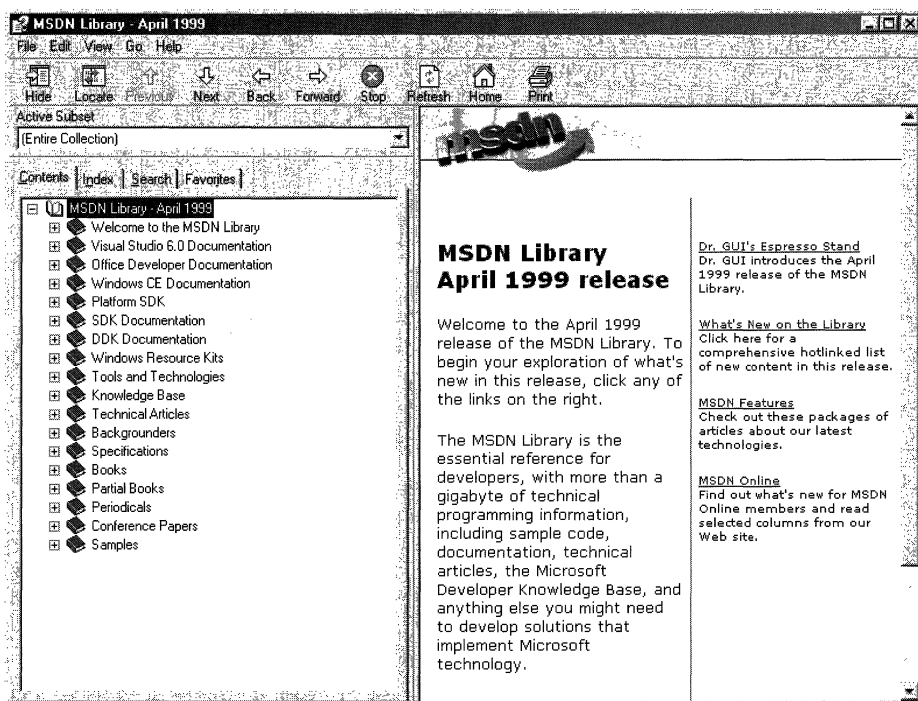


Figure 3-2: The MSDN interface.

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

Exploring MSDN

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

Basic exploration through MSDN is simple, and a lot like moving 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 display its contents 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 choose **Navigation Tabs**, and they'll appear.

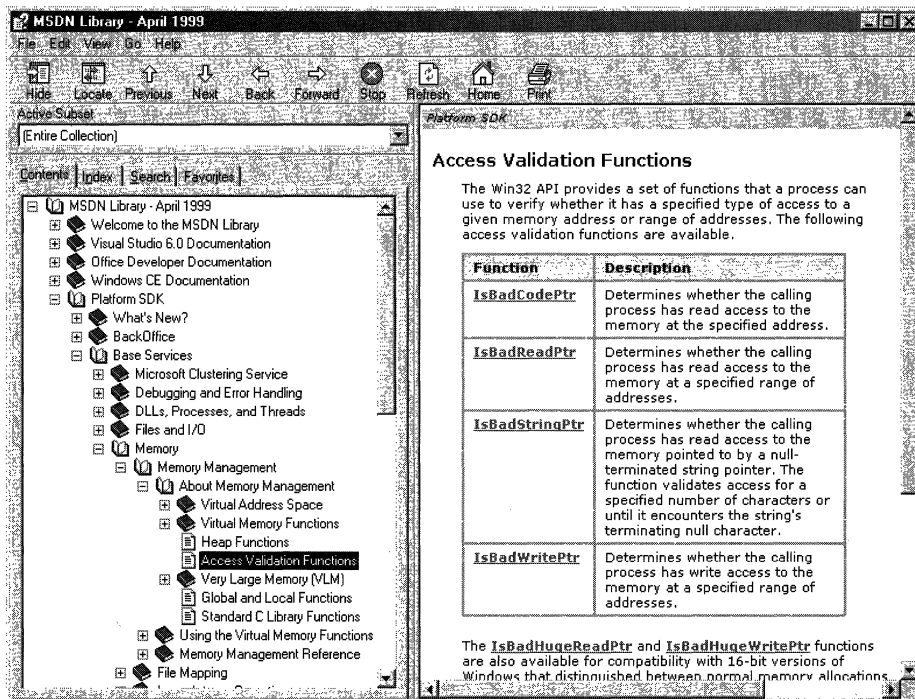


Figure 3-3: Basic exploration of MSDN.

The four tabs in the left pane of MSDN—increasingly referred to as property sheets these days—are the primary means of moving 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 exploration tools greatly improve your MSDN experience.

The **Active Subset** drop-down box is a filter mechanism; choose the subset of MSDN information with which you're interested in working 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, and enabling you, thereby, to 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 dimmed (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, this bunch of filtering options essentially mirrors 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 Subset** menu item. You're presented with the window shown in Figure 3-4.

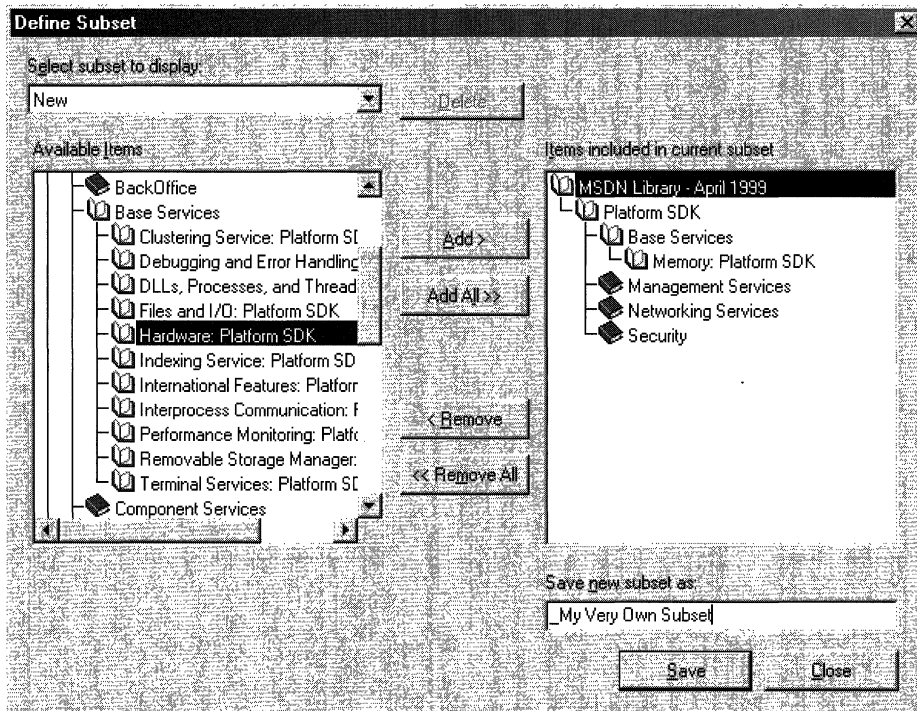


Figure 3-4: The Define Subset window.

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 also can 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 **Delete** button nearby.

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 explore 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 references 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, instead of 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 your 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 listing of available subsets in the Active Subset drop-down box. Also, by using an underscore, you can see immediately 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, weather, stock quotes, and other collections of information or news that suit your tastes or interests. You even can 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 specifies the 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.

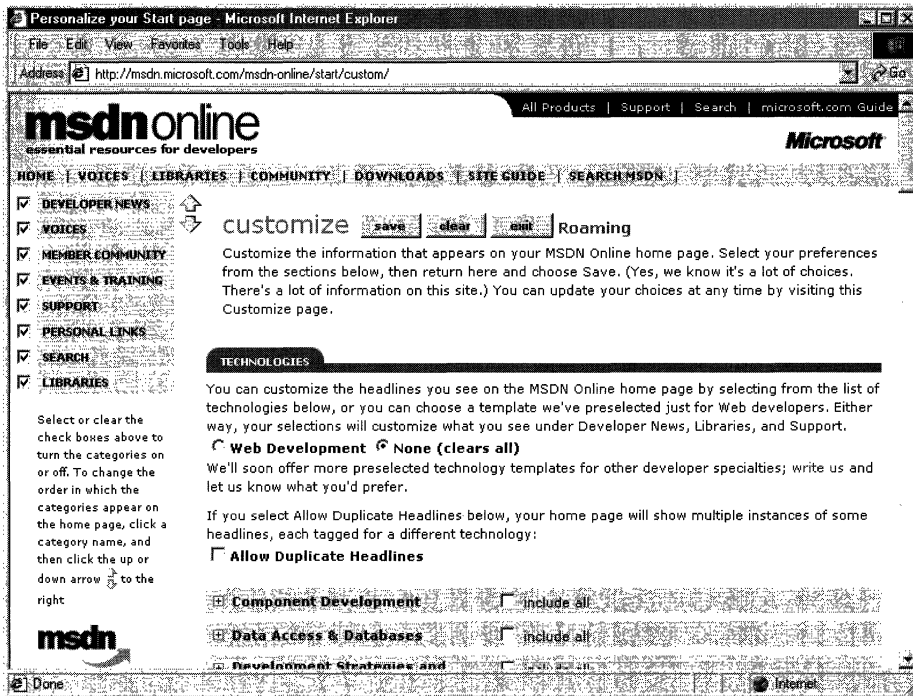


Figure 3-5: The MSDN Online customization page.

As you can see from Figure 3-5, there are lots of technologies from which to choose. If you're interested in Web development, you can choose the **Web Development** radio button near the top of the Technologies section, and a pre-defined subset of Web-oriented technologies is selected. For more Win32 Library-oriented technologies, you can go through and 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 choose also which categories are included in the information MSDN Online presents to you, as well as their arranged order. The available categories are the following:

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

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 check box—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 toolbar. In other words, if you clear the category, it won't be part of your customized MSDN Online page's headlines, but it'll be still 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*.

Exploring MSDN Online

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

The available menu categories—which group the available sites and features within MSDN Online—are the following:

| | |
|-------------|------------|
| Home | Voices |
| Libraries | Community |
| Downloads | Site Guide |
| Search MSDN | |

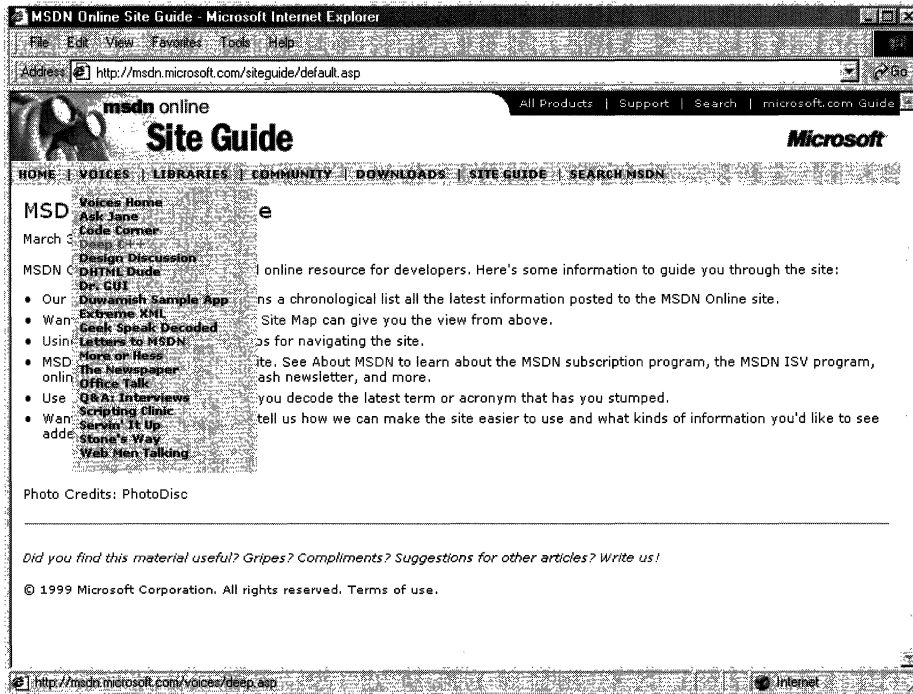


Figure 3-6: The MSDN Online navigation toolbar with its drop-down menus.

The navigation toolbar 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 contain the features available to developers visiting MSDN Online.

Home is already familiar; clicking on **Home** in the navigation toolbar takes you to the MSDN Online home page that you've customized (perhaps), 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 make up 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.

There is a bunch of different "voices" in the *Voices* site, each and adds its own particular twist to the issues that developers face. 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.

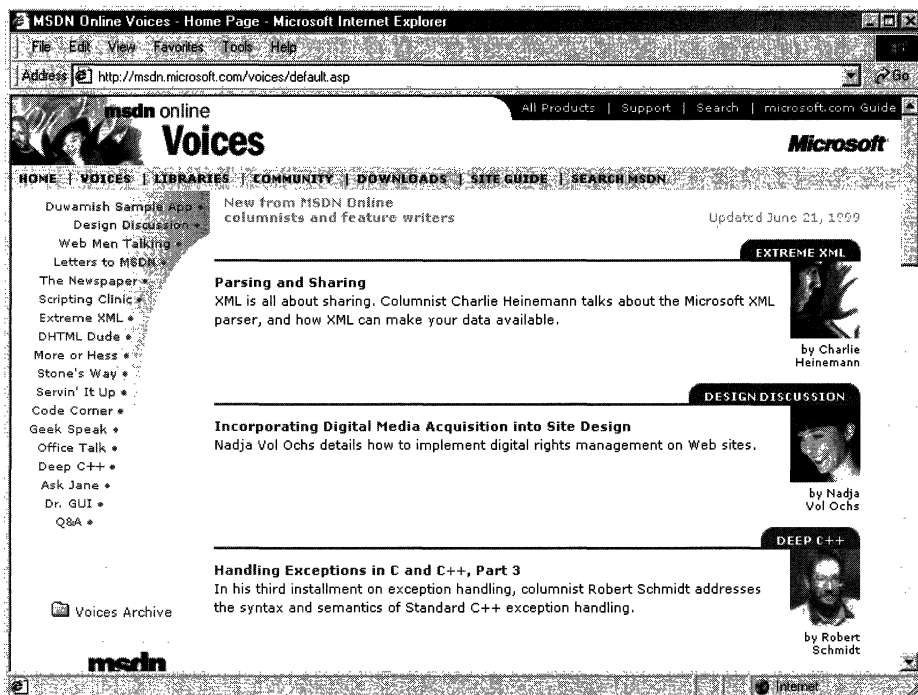


Figure 3-7: The Voices home page.

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.

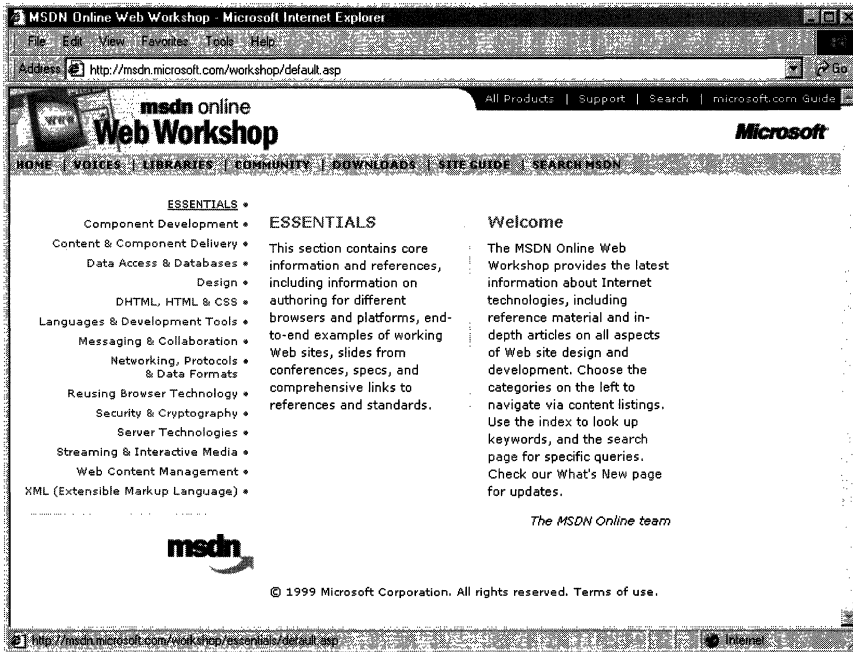


Figure 3-8: The Web Workshop home page, with its bulleted list of exploration start points.

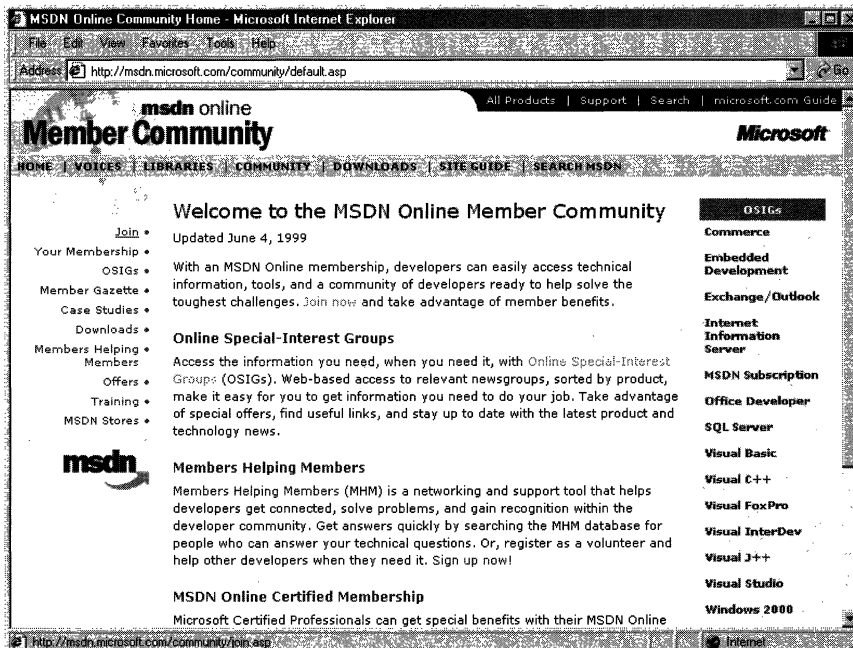


Figure 3-9: The Community home page.

The **Downloads** site is where developers can find all sorts of useable 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 to the latest and greatest releases over the Internet, 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.

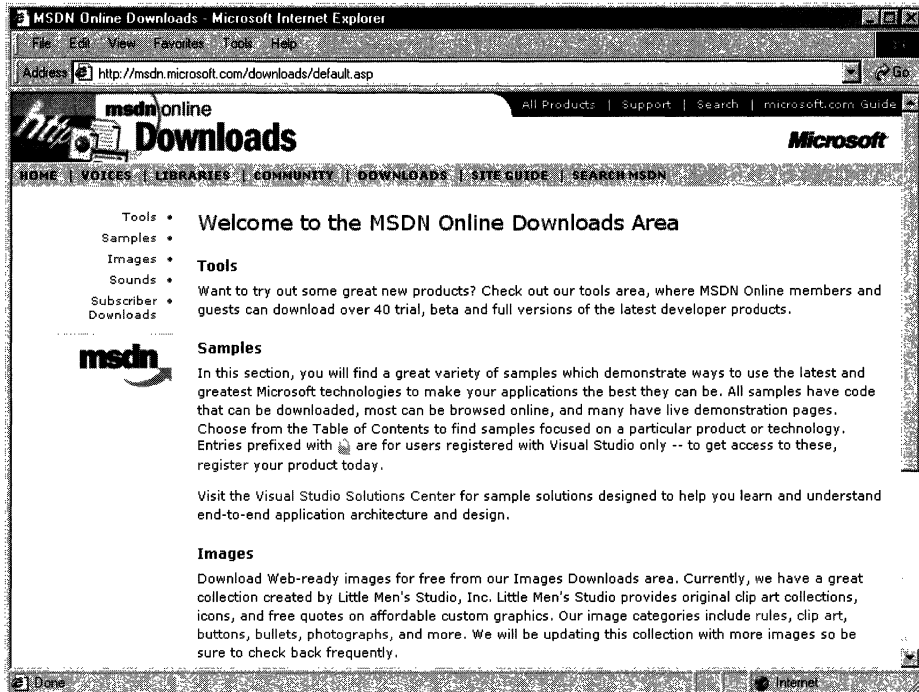


Figure 3-10: The Downloads home page.

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 of the libraries (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.

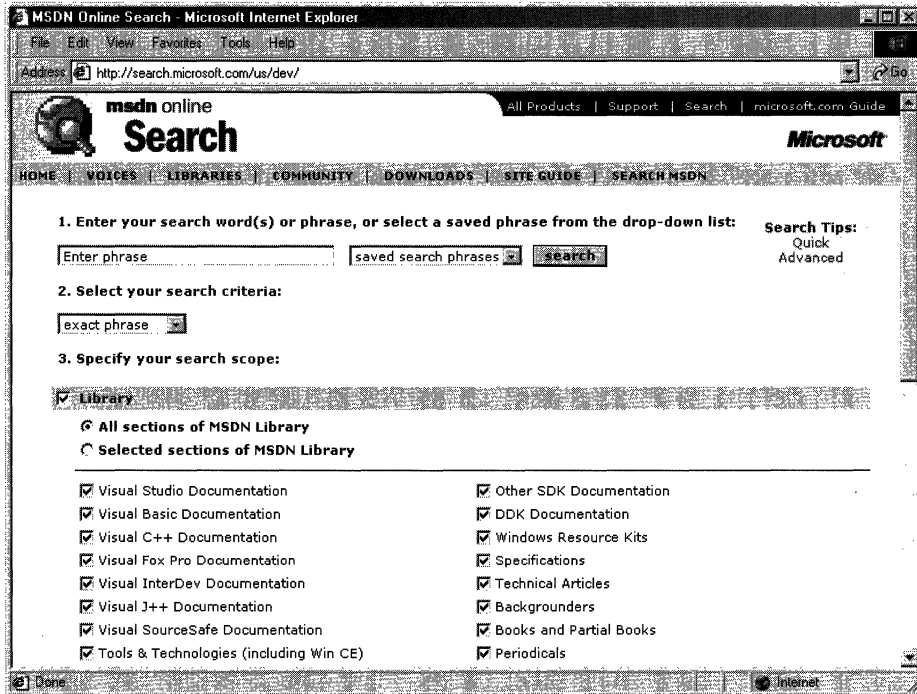


Figure 3-11: The Search MSDN home page.

MSDN Online Registered Users

You might 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.

Some features of MSDN Online require registration before you can take advantage of their offerings. For example, becoming a member of an OSIG requires registration. That feature alone is enough of a reason 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 get also 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 also can 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.

The Windows Programming Reference Series

The Windows Programming Reference Series provides developers with timely, concise, and focused material on a given topic, enabling them to get their work done as efficiently as possible. In addition to providing reference material for Microsoft technologies, each Pack 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 currently includes the following Packs:

- Win32 Library
- Active Directory Services Library
- Networking Services Library

In the near future (subject, of course, to technology release schedules, demand, and other forces that can impact publication decisions), you can look for these prospective Windows Programming Reference Series Packs that cover the following material:

- COM/DCOM 2.0 Library
- Web Reference Library
- Web Technologies Library

What else might you find in the future? Planned topics, such as a Security Pack, Language Reference Pack, MFC Pack, BackOffice Pack, or other pertinent topics that developers using Microsoft products need in order to get the most out of their development efforts, are prime subjects for future membership in the Windows Programming Reference Series. If you have feedback you want to provide on such packs, or on the Windows Programming Reference Series in general, you can send e-mail to the following address:

winprs@microsoft.com

If you're sending e-mail about a particular pack, make sure you put the name of the pack in the subject line. For example, an e-mail about the Win32 Library would have a subject line that reads "Win32 Library." 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 comments, concerns, or (especially) compliments get to the right place.

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.

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.

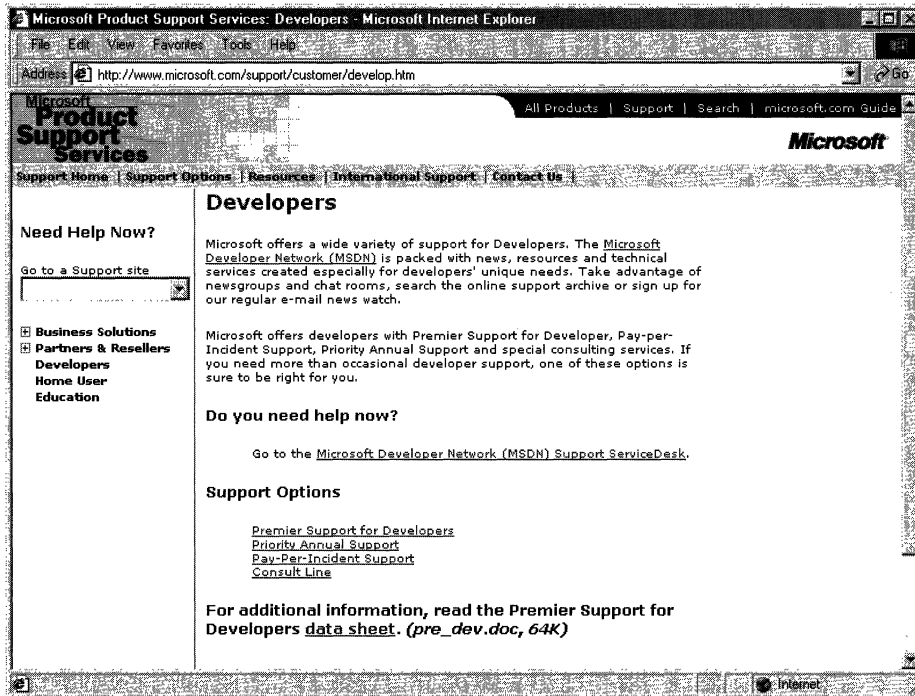


Figure 4-1: The Product Services Support page for developers.

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 human 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:

winprs@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 2

This chapter is the second of the five-part collection of common programming errors that is included in the *Microsoft Win32 Developer's Reference 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 Validation

Volume 3: RPC Errors and Kernel-Mode Specifiers

Volume 4: Buffer Overflows and Miscellaneous Errors

Volume 5: Memory Abuse and Miscalculations

This of course is Volume 2, and the errors and examples found in this chapter provide insights that can help you avoid problems with invalid validations in your programming projects. So without further ado, here they are!

Avoiding Invalid Validation

Parameter validation can be a complex process. Validation is often skipped in what appears to be a “private interface” for a given application or component; the problem arises when these “private interfaces” are exposed to other components or applications and, therefore, become callable by “non-private” code. The best weapon against this common programming error is to know that careful validation is necessary for any exposed function.

One way to sum up this section is “never assume.” Another way to sum up this section is “private interfaces need checks, too.” The following is a list of rules that can help you avoid these programming problems:

- Validate all objects referenced by generic handles.
- Don't assume correlation between parameters—verify all supposedly correlated data.
- Exception handling is not always the answer. Check return values and error codes, whenever possible.
- Include parameter validation in alternate (private) interfaces, or reject calls from untrusted sources.
- Carefully review *all* code paths, not just the primary code path.
- In general, treat all data as suspect.


```

                                NULL);
if (INT_SUCCESS (Status)) {
    return status;
}
Status = ZwWriteFile(req->handle,
                    ...);

```

Remarks

By the time the **ZwWriteFile()** function gets control, this handle might be invalid—or, worse, the handle might be for an open file to which the caller doesn't have write access.

Verify Correlated Parameters

Some functions assume that the values of two independent input fields are bound by a correlation, without explicitly verifying that the correlation is satisfied. Don't rely on implicit correlation—verify it.

Example

```

typedef struct _INPUT_STRUCTURE {
    ULONG Type;
    union {
        UCHAR String[256];
        ULONG ID;
    };
} INPUT_STRUCTURE, *PINPUT_STRUCTURE;

NTSTATUS
FalseCorrelationVictim(
    PVOID Buffer,
    ULONG BufferSize
)
{
    PINPUT_STRUCTURE InStruct = (PINPUT_STRUCTURE)Buffer;
    UCHAR CapturedString[256];
    //
    // Make sure the union field was correctly passed.
    //
    if (BufferSize < sizeof(ULONG))
        return STATUS_INVALID_PARAMETER;
    //
    // If Type == 1, this is a name packet. Get the string
    // out of the buffer.
    //

```

(continued)

(continued)

```

if (InStruct->Type == 1) {
    RtlCopyMemory(CapturedString,
                  InStruct->String,
                  256);
}
[...]
```

Remarks

In this example, the **FalseCorrelationVictim()** function has been “fooled” into believing that the size of the buffer was correctly indicated by the value of the **Type** field in the input structure. The real size of the buffer is in **BufferSize**, which has been verified only to be at least large enough to cover the **Type** field. This type of attack can be particularly insidious when it is possible for an input buffer to contain variably sized components, the size of which might be calculated as the difference between **BufferSize** and the size of the fixed portion of the input structure data—for instance, a buffer header followed by a data packet. If the routine can be manipulated into “believing” that the fixed portion is larger than the size of the indicated packet, large values will be calculated for the trailing buffer size when the difference wraps around zero. In many cases, this problem is created by checking a passed value and utilizing a calculated value that should match.

Limits of Exception Handling

Exception handling can be quite useful for catching and handling many problems. However, exception handling can cause problems if it is overused, as the following sections explain.

Kernel mode NULL dereference is unsafe, even when protected by try-except

The most common case of this problem is when the code assumes that dereferencing spoofed or incorrect values will be caught by exception handling and, therefore, the code does not sufficiently check input values. Don’t fall for this—there are many ways to “fool” exception handling. Even NULL can be mapped as a valid address. To avoid this common error, check values explicitly.

Example

```

NTSTATUS
YetAnotherBadFunction(
    PVOID Param,
    ULONG Size
)
{
    PCHAR    *Buffer;

    try {
        Buffer = ExAllocatePoolTag(PagedPool,
```

```

        size,
        ' guB');
    // Exception will occur if there isn't a page
    // in this process mapped at address 0x0.
    // If there is a page there, this corrupts
    // the legitimate user of that page.

    RtlCopyMemory(Buffer,
                  Param,
                  Size);
    [...]
} except (EXCEPTION_EXECUTE_HANDLER) {
    return GetExceptionCode();
}

// If there was a page mapped at address 0, and
// the allocation failed, we won't have taken an
// exception above. Instead, we'll bugcheck,
// attempting to free NULL here.

ExFreePool(Buffer);
}

```

Use `__leave` in the try block of a try-finally

Return, **break**, **continue**, and **goto** from the **try** block of a **try-finally** cause unwind; this approach negatively impacts performance (even in error handling code). A better approach is to use **__leave**. **Return** from the **try** block of a **try-finally** causes function execution to end after the last statement in the **finally** block.

Example

```

VOID          *Buffer1, Buffer2;

Open(&Foo):

try {
    Buffer1 = malloc(BUFFER_SIZE);

    if (!Buffer1) {
        // This causes unwind; unwind is slow.
        // Store the value to return in a temp
        // and use "__leave;"
        return FALSE;
    }
}

```

(continued)

(continued)

```

    Buffer2 = malloc(BUFFER_SIZE);
    if (!Buffer2) {
        // This should also be a __leave.
        return FALSE;
    }
} finally {
    Close(foo);
}

// The following code is not executed when
// the returns in the try-finally execute.
if (Buffer1) {
    free(Buffer1);
}

if (Buffer2) {
    free(Buffer2);
}

```

Ramifications of returning from a finally block

Return from a **finally** block halts unwind, effectively handling an exception, if one was raised. **Return** from a **finally** block after **return** from a **try** block results in the **finally**'s return value being returned; loss of the **try** block's value.

Example

```

try {
    if (!Open(&foo)) {
        RaiseException(STATUS_FILE_INVALID,
                      EXCEPTION_NONCONTINUABLE,
                      0,
                      NULL);
    }
} finally {
    if (foo) {
        Close(foo);
    }

    return;
}

// The exception raised is handled here.
}

```

Be wary of execution order

An **except** block's filter argument of **try-except** block containing a **try-finally** is evaluated before the **finally**, which occurs before the exception handler.

Example

```

BOOLEAN
Filter(
    VOID
)
{
    // 4.
}

VOID
FNC(
    VOID
)
{
    // 1.
    try {
        // 2.
        try {
            // 3. Exception occurs here.
        } finally {
            // 5.
        }
        // 6. (not executed when an exception
        // occurs at point 3)
    } except (filter()) {
        // 7.
    }
    // 8.
}

```

Remarks

In this example, blocks are executed in order numbered for the case where a single exception is raised in block 3. The **filter** (4) and **except** (7) blocks do not execute unless an exception occurs in block 2, 3, 5, or 6. Block 6 does not execute if an exception occurs in block 2, 3, or 5. When an exception occurs in block 5 or 6, the filter is executed before block 7.

Avoid relying on exceptions instead of correct validation

Taking exceptions due to bad memory references in user mode, even within the protected block of a **try-except**, must be avoided.

Stacks are built from a range of committed pages, followed by a guard page. This guard page, when tripped by stack expansion (for example, because of recursion), turns into a mapped page, and the handler for the guard page exception (in this case the kernel) creates another guard page. This process is repeated for each stack page required until the lower limit of the stack is reached; since a given thread's stack limits are known, the kernel handles this expansion.

When the lower limit of the stack is reached, the guard page is not created. Any further attempt to grow the stack results in an access violation, as the unmapped page is touched (or no access violation occurs, as the data at the mapped page is trashed).

When a thread takes an exception in an out-of-stack condition, its *process is terminated*, without recourse to any stack-based exception handlers.

Touching a guard page outside of a thread's stack also results in a guard page exception, and the page is mapped. Such action does not result in stack expansion, because it isn't within the current thread's stack base and stack limits.

Thus, if an attacker can cause an out of range access that happens to coincide with the guard page of another thread's stack, that thread becomes limited to the currently committed stack (now including the guard page that was just mapped). An attempt to use more stack than this results in process termination.

Alternate Code Paths

There are often multiple means of changing the state of an object. Occasionally, alternate means of performing the change are missing test checks that the more usual entry point is being afforded. Since it is difficult to verify that all routes have the same access checks, using common validation code will simplify the process of checking all possible code paths.

Example

```
NTSTATUS
ChangeObjectToStateA(HANDLE Object)
{
    if (VALID_HANDLE(Object)) {
        if (UserHasAdminPrivilege(Object)) {
            ChangeToStateA(Object);
        } else {
            return(STATUS_ACCESS_DENIED);
        }
    } else {
        return(STATUS_INVALID_HANDLE);
    }
    return(STATUS_SUCCESS);
}
```

```
NTSTATUS
GetInfoFromObject(
    HANDLE Object,
    PVOID Buffer,
    ULONG BufferSize
)
{
    STATE PreviousState;
    if (!INVALID_HANDLE(Object)) {
        return(STATUS_INVALID_HANDLE);
    }
    //
    // NOTE: Target object must be in state A to retrieve info.
    //
    ChangeToStateA(Object);
    ReadObjectInfo(Object, Buffer, BufferSize);
    RevertToPreviousState(Object);
    return(STATUS_SUCCESS);
}
```

Remarks

The problem with this example is that the **GetInfoFromObject()** function has created a timing window in which a non-administrative user can access **Object** from a different thread while in state A. Had the user merely attempted to change to state A by using the **ChangeObjectToStateA()** function, a failure due to insufficient privileges would have resulted. Although the **RevertToPreviousState()** function call places **Object** back into an allowed state to the user prior to returning, asynchronous access to the object in state A was available to the user throughout the call to the **ReadObjectInfo()** function.

Trusted Data Sources

Many application components expect all inputs to be controlled by some other code. If a component is receiving formatted data from a “trusted” data source and makes assumptions as to the format of the input data, it must verify that the data came from the trusted source. Unless the data source can be verified 100 percent as a “trusted” entity, all data must be treated as suspect. Since this is a very general class of attack, many previously cited examples are specific manifestations.

In rare cases of developer negligence, this class might appear as a complete lack of any data verification, with the code citing the “internal” or “private” nature of the interface as verification that the entity providing the data is trustworthy and has correctly formatted the data. Data-processing bugs in these cases are frequently written off as bugs in the “trusted” component, further exposing the incorrectly secured interface to attack.

More commonly, this class of programming error is discovered when analyzing the behavior of a component that has already verified portions of the data correctly, or that has a previous relationship with the entity supplying the data in which correctly verified information has been successfully received and processed. These attacks are much subtler because they require an attacker to know enough about the interface to provide data that is partly or mostly verifiable. Additionally, these bugs are tedious and difficult to track and eradicate, by similar reasoning.

Example

```
typedef struct _PACKET {
    ULONG PacketSize;
    ULONG CommandCode;
    ULONG EncryptedPasswordLength;
    UCHAR Data[];
} PACKET, *PPACKET;

NTSTATUS
)
ReceivePacket(
    PPACKET Packet,
    ULONG PacketSize
)
{
    PCHAR EndOfBuffer = (PCHAR)Packet + PacketSize;
    PCHAR PasswordBuffer = NULL;
    PCHAR UserName = Packet->Data;

    if (PacketSize < sizeof (PACKET) ||
        PacketSize < Packet->PacketSize ||
        PacketSize < EncryptedPasswordLength ||
        PacketSize < strlen(UserName) + 1) {
        return STATUS_INVALID_PARAMETER;
    }
    [...]
}
```

Remarks

The programming error here is subtle and made more difficult by the fact that it occurs in the very code that is attempting to verify that the packet is valid. There is no guarantee that the user name string indicated in the packet is NULL terminated, potentially causing an exception when **strlen** violates the buffer size of the indicated packet.

Solutions 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.

Avoiding Invalid Validation

1. **Working with Handle-Based Objects:** Validate all objects referenced by generic handles.
2. **Verify Correlated Parameters:** Don't assume correlation between parameters. Verify all supposedly correlated data.
3. **Limits of Exception Handling:** Exception handling is not always the answer. Check return values and error codes, whenever possible.
4. **Alternate Code Paths:** Include parameter validation in alternate (private) interfaces, or reject calls from untrusted sources.
5. **Trusted Data Sources:** Treat all data as suspect.

CHAPTER 6

Controls

Controls

A *control* is a child window that an application uses in conjunction with another window to perform simple input and output (I/O) tasks. Controls are most often used within dialog boxes, but they can be used also in other windows. Controls within dialog boxes provide the user with the means to type text, choose options, and direct a dialog box to complete its action. Controls in other windows provide a variety of services, such as letting the user choose commands, view status, and view and edit text.

About Controls

Controls, like other windows, belong to a window class, either predefined or owner-defined. The window class and the corresponding window procedure define the properties of the control, as well as its appearance, behavior, and purpose. An application can create controls individually by specifying the name of the window class when calling the **CreateWindowEx** function. An application also can direct the system to create controls for a dialog box by specifying the controls in the dialog-box template.

Predefined Controls

The system provides several predefined window classes for controls. Controls belonging to these window classes are called *predefined controls*. An application creates a predefined control of a particular type by specifying the appropriate window class name in either the **CreateWindowEx** function or the dialog-box template. The following are the predefined window classes:

| Name | Description |
|----------|---|
| BUTTON | Creates button controls. These controls typically notify the parent window when the user chooses the control. For more information, see <i>Buttons</i> . |
| COMBOBOX | Creates combo boxes. These controls are a combination of list boxes and edit controls, letting the user choose and edit items. For more information, see <i>Combo Boxes</i> . |
| EDIT | Creates edit controls. These controls let the user view and edit text. For more information, see <i>Edit Controls</i> . |
| LISTBOX | Creates list boxes. These controls display a list from which the user can select one or more items. For more information, see <i>List Boxes</i> . |

| | |
|----------------|---|
| RICHEDIT | Creates Rich Edit version 1.0 controls. These controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see <i>Rich-Edit Controls</i> . |
| RICHEDIT_CLASS | Creates Rich Edit version 2.0 controls. These controls let the user view and edit text with character and paragraph formatting, and can include embedded COM objects. For more information, see <i>Rich-Edit Controls</i> . |
| SCROLLBAR | Creates scroll-bar controls. These controls let the user choose the direction and distance to scroll information in a related window. For more information, see <i>Scroll Bars</i> . |
| STATIC | Creates static controls. These controls often act as labels for other controls. For more information, see <i>Static Controls</i> . |

Each predefined window class has a corresponding set of *control styles* that enable an application to vary the appearance and behavior of the controls the class creates. For example, the `BUTTON` class supports styles to create push buttons, radio buttons, check boxes, and group boxes. An application specifies the style when creating the control.

Each predefined window class has a corresponding set of notification and control messages. Applications rely on the notification messages to determine when the user has provided input to the controls. For example, a push button sends a `BN_CLICKED` message to the parent window when the user clicks the button. Applications use the control messages to retrieve information from the controls, and to manipulate the appearance and behavior of the controls. For example, an application can send a `BM_GETCHECK` message to a check box to determine whether it currently contains a check mark.

Most applications make extensive use of predefined controls in dialog boxes and other windows. Because predefined controls offer many capabilities, a full discussion of each is beyond the scope of this topic.

Control Reference

Control Messages

WM_GETFONT

An application sends a `WM_GETFONT` message to a control to retrieve the font with which the control is currently drawing its text.

To send this message, call the `SendMessage` function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_GETFONT,           // message to send
```

```

(WPARAM) wParam;           // not used; must be zero
(LPARAM) lParam;           // not used; must be zero
);

```

Parameters

This message has no parameters.

Return Values

The return value is a handle to the font used by the control, or NULL if the control is using the system font.

! 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

Controls Overview, Control Messages, **WM_SETFONT**

WM_SETFONT

An application sends a **WM_SETFONT** message to specify the font that a control is to use when drawing text.

To send this message, call the **SendMessage** function with the following parameters.

```

SendMessage(
  (HWND) hwnd,              // handle to destination window
  WM_SETFONT,               // message to send
  (WPARAM) wParam,         // handle to font
  (LPARAM) lParam,         // redraw option
);

```

Parameters

wParam

Handle to the font. If this parameter is NULL, the control uses the default system font to draw text.

lParam

The low-order word of *lParam* specifies whether the control should be redrawn immediately upon setting the font. If this parameter is TRUE, the control redraws itself.

Return Values

This message does not return a value.

Remarks

The **WM_SETFONT** message applies to all controls, not just those in dialog boxes.

The best time for the owner of a dialog-box control to set the font of the control is when it receives the **WM_INITDIALOG** message. The application should call the **DeleteObject** function to delete the font when it is no longer needed; for example, after it destroys the control.

The size of the control does not change as a result of receiving this message. To avoid clipping text that does not fit within the boundaries of the control, the application should correct the size of the control window before it sets the font.

When a dialog box uses the **DS_SETFONT** style to set the text in its controls, the system sends the **WM_SETFONT** message to the dialog-box procedure before it creates the controls. An application can create a dialog box that contains the **DS_SETFONT** style by calling any of the following functions:

- **CreateDialogIndirect**
- **CreateDialogIndirectParam**
- **DialogBoxIndirect**
- **DialogBoxIndirectParam**



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

Controls Overview, Control Messages, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **DeleteObject**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DLGTEMPLATE**, **MAKELPARAM**, **WM_INITDIALOG**

Buttons

Dialog boxes and controls support communication between an application and the user. A *button* is a control that the user can click to provide input to an application.

About Buttons

There are several types of buttons, and one or more button styles to distinguish among buttons of the same type. The user clicks a button using the mouse or keyboard. Clicking

a button typically changes its visual appearance and state (from checked to cleared, for example). The system, button, and application cooperate in changing the button's appearance and state. A button can send messages to its parent window, and a parent window can send messages to a button. Some buttons are painted by the system, some by the application. Buttons can be used alone or in groups, and can appear with or without application-defined text (a label). They belong to the `BUTTON` window class.

Although an application can use buttons in overlapped, pop-up, and child windows, they are designed for use in dialog boxes, where the system standardizes their behavior. If an application uses buttons outside dialog boxes, it increases the risk that the application might behave in a nonstandard fashion. Applications typically use either buttons in dialog boxes or window subclassing to create customized buttons.

Button Reference

Button Functions

CheckDlgButton

The `CheckDlgButton` function changes the check state of a button control.

```
BOOL CheckDlgButton(
    HWND hDlg,           // handle to dialog box
    int nIDButton,      // button identifier
    UINT uCheck,        // check state
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the button.

nIDButton

[in] Specifies the identifier of the button to modify.

uCheck

[in] Specifies the check state of the button. This parameter can be one of the following values:

| Value | Meaning |
|--------------------------------|---|
| <code>BST_CHECKED</code> | Sets the button state to checked. |
| <code>BST_INDETERMINATE</code> | Sets the button state to shaded, indicating an indeterminate state. Use this value only if the button has the <code>BS_3STATE</code> or <code>BS_AUTO3STATE</code> style. |
| <code>BST_UNCHECKED</code> | Sets the button state to cleared. |

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 **CheckDlgButton** function sends a **BM_SETCHECK** message to the specified button control in the specified dialog box.

! 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

Buttons Overview, Button Functions, **CheckRadioButton**, **IsDlgButtonChecked**

CheckRadioButton

The **CheckRadioButton** function adds a check mark (checks) to a specified radio button in a group and removes a check mark (clears) from all other radio buttons in the group.

```
BOOL CheckRadioButton(  
    HWND hDlg,           // handle to dialog box  
    int nIDFirstButton, // identifier of first button in group  
    int nIDLastButton,  // identifier of last button in group  
    int nIDCheckButton  // identifier of button to select  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the radio button.

nIDFirstButton

[in] Specifies the identifier of the first radio button in the group.

nIDLastButton

[in] Specifies the identifier of the last radio button in the group.

nIDCheckButton

[in] Specifies the identifier of the radio button to select.

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 **CheckRadioButton** function sends a **BM_SETCHECK** message to each of the radio buttons in the indicated group.

! 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

Buttons Overview, Button Functions, **BM_SETCHECK**, **CheckDlgButton**, **IsDlgButtonChecked**

IsDlgButtonChecked

The **IsDlgButtonChecked** function determines whether a button control has a check mark next to it, or whether a three-state button control is shaded, checked, or neither.

```
UINT IsDlgButtonChecked(  
    HWND hDlg, // handle to dialog box  
    int nIDButton // button identifier  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the button control.

nIDButton

[in] Specifies the identifier of the button control.

Return Values

The return value from a button created with the **BS_AUTOCHECKBOX**, **BS_AUTORADIOBUTTON**, **BS_AUTO3STATE**, **BS_CHECKBOX**, **BS_RADIOBUTTON**, or **BS_3STATE** style can be one of the following:

| Value | Meaning |
|-------------------|--|
| BST_CHECKED | Button is checked. |
| BST_INDETERMINATE | Button is shaded, indicating an indeterminate state (applies only if the button has the BS_3STATE or BS_AUTO3STATE style). |
| BST_UNCHECKED | Button is cleared. |

If the button has any other style, the return value is zero.

Remarks

The **IsDlgButtonChecked** function sends a **BM_GETCHECK** message to the specified button control.

! 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

Buttons Overview, Button Functions, **CheckDlgButton**

Button Messages

BM_CLICK

An application sends a **BM_CLICK** message to simulate the user clicking a button. This message causes the button to receive the **WM_LBUTTONDOWN** and **WM_LBUTTONUP** messages, and the button's parent window to receive a **BN_CLICKED** notification message.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    BM_CLICK,             // message to send
    (LPARAM) wParam,     // not used; must be zero
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

This message has no parameters.

Return Values

This message does not return a value.

Remarks

If the button is in a dialog box and the dialog box is not active, the **BM_CLICK** message might fail. To ensure success in this situation, call the **SetActiveWindow** function to activate the dialog box before sending the **BM_CLICK** message to the button.

! 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`.

+ See Also

Buttons Overview, *Button Messages*, **BN_CLICKED**, **SetActiveWindow**, **WM_LBUTTONDOWN**, **WM_LBUTTONUP**

BM_GETCHECK

An application sends a **BM_GETCHECK** message to retrieve the check state of a radio button or check box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    BM_GETCHECK,          // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

The return value from a button created with the **BS_AUTOCHECKBOX**, **BS_AUTORADIOBUTTON**, **BS_AUTO3STATE**, **BS_CHECKBOX**, **BS_RADIOBUTTON**, or **BS_3STATE** style can be one of the following:

| Value | Meaning |
|-------------------|--|
| BST_CHECKED | Button is checked. |
| BST_INDETERMINATE | Button is shaded, indicating an indeterminate state (applies only if the button has the BS_3STATE or BS_AUTO3STATE style). |
| BST_UNCHECKED | Button is cleared. |

If the button has any other style, 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 winuser.h; include windows.h.

+ See Also

Buttons Overview, Button Messages, **BM_GETSTATE**, **BM_SETCHECK**

BM_GETIMAGE

An application sends a **BM_GETIMAGE** message to retrieve a handle to the image (either icon or bitmap) associated with the button.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    BM_GETIMAGE,          // message to send
    (WPARAM) wParam,     // image type
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies the type of image to associate with the button. This parameter can be one of the following values:

IMAGE_BITMAP
IMAGE_ICON

lParam

This parameter is not used.

Return Values

The return value is a handle to the image, if any; otherwise, it is NULL.

! 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

Buttons Overview, Button Messages, **BM_SETIMAGE**

BM_GETSTATE

An application sends a **BM_GETSTATE** message to determine the state of a button or check box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    BM_GETSTATE,         // message to send
    (WPARAM) wParam,     // not used; must be zero
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

This message has no parameters.

Return Values

The return value specifies the current state of the button. You can use the following values to extract information about the state:

| Value | Meaning |
|--------|--|
| 0x0003 | Specifies the check state (radio buttons and check boxes only). A value of <code>BST_UNCHECKED</code> indicates the button is cleared; a value of <code>BST_CHECKED</code> indicates the button is checked. A radio button is checked when it contains a dot; a check box is checked when it contains an X. A value of <code>BST_INDETERMINATE</code> indicates the check state is indeterminate (applies only if the button has the <code>BS_3STATE</code> or <code>BS_AUTO3STATE</code> style). A three-state check box is shaded when its state is indeterminate. |

(continued)

(continued)

| Value | Meaning |
|-------------------|--|
| BST_CHECKED | Indicates the button is checked. |
| BST_FOCUS | Specifies the focus state. A nonzero value indicates that the button has the keyboard focus. |
| BST_INDETERMINATE | Indicates the button is shaded because the state of the button is indeterminate. This value applies only if the button has the BS_3STATE or BS_AUTO3STATE style. |
| BST_PUSHED | Specifies the highlight state. A nonzero value indicates that the button is highlighted. A button is highlighted automatically when the user positions the cursor over it, and presses and holds the left mouse button. The highlighting is removed when the user releases the mouse button. |
| BST_UNCHECKED | Indicates the button is cleared. Same as a return value of 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 winuser.h; include windows.h.

See Also

Buttons Overview, Button Messages, **BM_GETCHECK**, **BM_SETSTATE**

BM_SETCHECK

An application sends a **BM_SETCHECK** message to set the check state of a radio button or check box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    BM_SETCHECK,         // message to send  
    (WPARAM) wParam,     // check state  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

wParam

Specifies the check state. This parameter can be one of the following values:

| Value | Meaning |
|-------------------|---|
| BST_CHECKED | Sets the button state to checked. |
| BST_INDETERMINATE | Sets the button state to shaded, indicating an indeterminate state. Use this value only if the button has the BS_3STATE or BS_AUTO3STATE style. |
| BST_UNCHECKED | Sets the button state to cleared. |

IParam

This parameter is not used.

Return Values

This message always returns zero.

Remarks

The **BM_SETCHECK** message has no effect on push buttons.

! 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

Buttons Overview, Button Messages, **BM_GETCHECK**, **BM_GETSTATE**, **BM_SETSTATE**

BM_SETIMAGE

An application sends a **BM_SETIMAGE** message to associate a new image (icon or bitmap) with the button.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    BM_SETIMAGE,          // message to send
    (WPARAM) wParam,     // image type
    (LPARAM) lParam,     // handle to the image (HANDLE)
);
```

Parameters

wParam

Specifies the type of image to associate with the button. This parameter can be one of the following values:

IMAGE_BITMAP
IMAGE_ICON

lParam

Handle to the image to associate with the button.

Return Values

The return value is a handle to the image previously associated with the button, if any; otherwise, it is NULL.

! 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

Buttons Overview, Button Messages, **BM_GETIMAGE**

BM_SETSTATE

An application sends a **BM_SETSTATE** message to change the highlight state of a button. The highlight state indicates whether the button is highlighted as if the user had pushed it.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    BM_SETSTATE,          // message to send  
    (WPARAM) wParam,      // highlight state  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

Specifies whether the button is to be highlighted. A value of TRUE highlights the button; a value of FALSE removes any highlighting.

lParam

This parameter is not used.

Return Values

This message always returns zero.

Remarks

Highlighting only affects the appearance of a button. It has no effect on the check state of a radio button or check box.

A button is highlighted automatically when the user positions the cursor over it, and presses and holds the left mouse button. The highlighting is removed when the user releases the mouse button.

! 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

Buttons Overview, Button Messages, **BM_GETSTATE**, **BM_SETCHECK**

BM_SETSTYLE

An application sends a **BM_SETSTYLE** message to change the style of a button.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    BM_SETSTYLE,          // message to send  
    (LPARAM) wParam,      // button style  
    (LPARAM) lParam;      // redraw state  
);
```

Parameters

wParam

Specifies the new button style. This parameter can be a combination of button styles. For a table of button styles, see *Button Styles*.

lParam

The low-order word of *lParam* specifies whether the button is to be redrawn. A value of TRUE redraws the button; a value of FALSE does not redraw the button.

Return Values

This message always returns 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 winuser.h; include windows.h.

+ See Also

Buttons Overview, Button Messages, **LOWORD**

BN_CLICKED

The **BN_CLICKED** notification code is sent when the user clicks a button.

The parent window of the button receives the **BN_CLICKED** notification code through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,      // identifier of button, BN_CLICKED
    LPARAM lParam       // handle to button (HWND)
);
```

Parameters

wParam

The low-order word contains the button's control identifier.

The high-order word specifies the notification message.

lParam

Handle to the button.

Remarks

A disabled button does not send a **BN_CLICKED** notification message to its parent window.

! 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

Buttons Overview, Button Messages, **HIWORD**, **LOWORD**, **WM_COMMAND**

BN_DBLCLK

The **BN_DBLCLK** notification code is sent when the user double-clicks a button. This notification is sent automatically for **BS_USERBUTTON**, **BS_RADIOBUTTON**, and **BS_OWNERDRAW** buttons. Other button types send **BN_DBLCLK** only if they have the **BS_NOTIFY** style.

The parent window of the button receives the **BN_DBLCLK** notification code through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,      // identifier of button, BN_DBLCLK  
    LPARAM lParam       // handle to button (HWND)  
);
```

Parameters

wParam

The low-order word contains the button's control identifier.

The high-order word specifies the notification message.

lParam

Handle to the button.

Remarks

BN_DBLCLK is the same as the **BN_DOUBLECLICKED** notification message.

 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`.

 See Also

Buttons Overview, Button Messages, **BN_CLICKED**, **BN_DOUBLECLICKED**, **HIWORD**, **LOWORD**, **WM_COMMAND**

BN_DOUBLECLICKED

The **BN_DOUBLECLICKED** notification code is sent when the user double-clicks a button. This notification is sent automatically for **BS_USERBUTTON**, **BS_RADIOBUTTON**, and **BS_OWNERDRAW** buttons. Other button types send **BN_DOUBLECLICKED** only if they have the **BS_NOTIFY** style.

The parent window of the button receives the **BN_DOUBLECLICKED** notification code through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,          // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,     // identifier of button, BN_DOUBLECLICKED  
    LPARAM lParam      // handle to button (HWND)  
);
```

Parameters

wParam

The low-order word contains the button's control identifier.

The high-order word specifies the notification message.

lParam

Handle to the button.

Remarks

BN_DOUBLECLICKED is the same as the **BN_DBLCLK** notification message.

! 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

Buttons Overview, Button Messages, **BN_DBLCLK**, **DRAWITEMSTRUCT**, **HIWORD**, **LOWORD**, **WM_COMMAND**, **WM_DRAWITEM**

BN_KILLFOCUS

The **BN_KILLFOCUS** notification code is sent when a button loses the keyboard focus. The button must have the **BS_NOTIFY** style to send this notification message.

The parent window of the button receives the **BN_KILLFOCUS** notification code through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,     // identifier of button, BN_KILLFOCUS
    LPARAM lParam      // handle to button (HWND)
);
```

Parameters

wParam

The low-order word contains the button's control identifier.

The high-order word specifies the notification message.

lParam

Handle to the button.

! 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`.

+ See Also

Buttons Overview, Button Messages, **BN_SETFOCUS**, **HIWORD**, **LOWORD**, **WM_COMMAND**

BN_SETFOCUS

The **BN_SETFOCUS** notification code is sent when a button receives the keyboard focus. The button must have the **BS_NOTIFY** style to send this notification message.

The parent window of the button receives the **BN_SETFOCUS** notification code through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,     // identifier of button, BN_SETFOCUS
    LPARAM lParam      // handle to button (HWND)
);
```

Parameters

wParam

The low-order word contains the button's control identifier.

The high-order word specifies the notification message.

lParam

Handle to the button.

! 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`.

+ See Also

Buttons Overview, Button Messages, **BN_KILLFOCUS**, **HIWORD**, **LOWORD**, **WM_COMMAND**

WM_CTLCOLORBTN

The **WM_CTLCOLORBTN** message is sent to the parent window of a button before drawing the button. The parent window can change the button's text and background colors. However, only owner-drawn buttons respond to the parent window processing this message.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_CTLCOLORBTN
    WPARAM wParam,       // handle to button display context (HDC)
    LPARAM lParam        // handle to button (HWND)
);

```

Parameters

wParam

Handle to the display context for the button.

lParam

Handle to the button.

Return Values

If an application processes this message, it must return a handle to a brush. The system uses the brush to paint the background of the button.

Remarks

By default, the **DefWindowProc** function selects the default system colors for the button. Buttons with the **BS_PUSHBUTTON**, **BS_DEFPUSHBUTTON**, or **BS_PUSHLIKE** styles do not use the returned brush; buttons with these styles are always drawn with the default system colors. Drawing push buttons requires several different brushes—face, highlight, and shadow—but the **WM_CTLCOLORBTN** message allows only one brush to be returned. To provide a custom appearance for push buttons, use an owner-drawn button.

The system does not destroy automatically the returned brush. It is the application's responsibility to destroy the brush when it is no longer needed.

The **WM_CTLCOLORBTN** message is never sent between threads; it is sent only within one thread.

The text color of a check box or radio button applies to the box or button, its check mark, and the text. The focus rectangle for these buttons remains the system default color (typically, black). The text color of a group box applies to the text, but not to the line that defines the box. The text color of a push button applies only to its focus rectangle; it does not affect the color of the text.

If a dialog-box procedure handles this message, then it should cast the desired return value to a **BOOL** and return the value directly. If the dialog-box procedure returns **FALSE**, then default message handling is performed. The **DWL_MSGRESULT** value set by the **SetWindowLong** function is ignored.

! 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

Buttons Overview, Button Messages, **DefWindowProc**, **RealizePalette**, **SelectPalette**

Button Styles

If you create a button by specifying the **BUTTON** class with the **CreateWindow** or **CreateWindowEx** function, you can specify a combination of the following button styles:

| Style | Meaning |
|------------------|--|
| BS_3STATE | Creates a button that is the same as a check box, except that the box can be shaded as well as checked or cleared. Use the shaded state to show that the state of the check box is not determined. |

(continued)

(continued)

| Style | Meaning |
|--------------------|---|
| BS_AUTO3STATE | Creates a button that is the same as a three-state check box, except that the box changes its state when the user selects it. The state cycles through checked, shaded, and cleared. |
| BS_AUTOCHECKBOX | Creates a button that is the same as a check box, except that the check state automatically toggles between checked and cleared each time the user selects the check box. |
| BS_AUTORADIOBUTTON | Creates a button that is the same as a radio button, except that, when the user selects it, the system automatically sets the button's check state to checked, and automatically sets the check state for all other buttons in the same group to cleared. |
| BS_BITMAP | Specifies that the button displays a bitmap. |
| BS_BOTTOM | Places text at the bottom of the button rectangle. |
| BS_CENTER | Centers text horizontally in the button rectangle. |
| BS_CHECKBOX | Creates a small, empty check box with text. By default, the text is displayed to the right of the check box. To display the text to the left of the check box, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style). |
| BS_DEFPUSHBUTTON | Creates a push button that behaves like a BS_PUSHBUTTON style button, but also has a heavy black border. If the button is in a dialog box, the user can select the button by pressing the ENTER key, even when the button does not have the input focus. This style is useful for enabling the user to quickly select the most likely (default) option. |
| BS_FLAT | Specifies that the button is two-dimensional; it does not use the default shading to create a three-dimensional image. |
| BS_GROUPBOX | Creates a rectangle in which other controls can be grouped. Any text associated with this style is displayed in the rectangle's upper-left corner. |
| BS_ICON | Specifies that the button displays an icon. |
| BS_LEFT | Left-justifies the text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is left justified on the right side of the check box or radio button. |
| BS_LEFTTEXT | Places text on the left side of the radio button or check box when combined with a radio-button or check-box style. Same as the BS_RIGHTBUTTON style. |
| BS_MULTILINE | Wraps the button text to multiple lines if the text string is too long to fit on a single line in the button rectangle. |

| Style | Meaning |
|----------------|---|
| BS_NOTIFY | Enables a button to send BN_KILLFOCUS and BN_SETFOCUS notification messages to its parent window. Note that buttons send the BN_CLICKED notification message regardless of whether it has this style. To get BN_DBLCLK notification messages, the button must have the BS_RADIOBUTTON or BS_OWNERDRAW style. |
| BS_OWNERDRAW | Creates an owner-drawn button. The owner window receives a WM_MEASUREITEM message when the button is created and a WM_DRAWITEM message when a visual aspect of the button has changed. Do not combine the BS_OWNERDRAW style with any other button styles. |
| BS_PUSHBUTTON | Creates a push button that posts a WM_COMMAND message to the owner window when the user selects the button. |
| BS_PUSHLIKE | Makes a button (such as a check box, three-state check box, or radio button) look and act like a push button. The button looks raised when it is not pushed or checked, and sunken when it is pushed or checked. |
| BS_RADIOBUTTON | Creates a small circle with text. By default, the text is displayed to the right of the circle. To display the text to the left of the circle, combine this flag with the BS_LEFTTEXT style (or with the equivalent BS_RIGHTBUTTON style). Use radio buttons for groups of related, but mutually exclusive, choices. |
| BS_RIGHT | Right-justifies text in the button rectangle. However, if the button is a check box or radio button that does not have the BS_RIGHTBUTTON style, the text is right-justified on the right side of the check box or radio button. |
| BS_RIGHTBUTTON | Positions a radio button's circle or a check box's square on the right side of the button rectangle. Same as the BS_LEFTTEXT style. |
| BS_TEXT | Specifies that the button displays text. |
| BS_TOP | Places text at the top of the button rectangle. |
| BS_USERBUTTON | Obsolete, but provided for compatibility with 16-bit versions of Windows. Win32-based applications should use BS_OWNERDRAW instead. |
| BS_VCENTER | Places text in the middle (vertically) of the button rectangle. |

Combo Boxes

A *combo box* is a unique type of control, defined by the COMBOBOX class, that combines much of the functionality of a list box and an edit control.

About Combo Boxes

The Win32 API provides three types of combo boxes:

- Simple combo boxes (CBS_SIMPLE)
- Drop-down combo boxes (CBS_DROPDOWN)
- Drop-down list boxes (CBS_DROPDOWNLIST)

There is also a number of combo-box styles that define specific properties. For example, two styles enable an application to create an owner-drawn combo box, making the application responsible for displaying information in the control.

A combo box consists of a list and a selection field. The list presents the options that a user can select, and the selection field displays the current selection. Except in drop-down list boxes, the selection field is an edit control and can be used to enter text not available in the list.

Combo-Box Types and Styles

Combo boxes can be characterized by type and style. Combo-box types determine whether the combo-box list is a drop-down list and whether the selection field is an edit control. A drop-down list appears only when the user opens it, so it uses less screen space than a list that is always visible. If the selection field is an edit control, the user can enter information not available in the list; otherwise, the user can select only items in the list.

The following table shows the three combo-box types, and indicates their drop-down list and edit control attributes:

| Combo-box type | Drop-down list | Edit control |
|-----------------------|-----------------------|---------------------|
| Drop-down combo box | Yes | Yes |
| Drop-down list box | Yes | No |
| Simple combo box | No | Yes |

Combo-box styles define specific properties of a combo box. You can combine styles; however, some styles apply only to certain combo-box types.

Combo-Box Reference

Combo-Box Functions

DlgDirListComboBox

The **DlgDirListComboBox** function replaces the contents of a combo box with the names of the subdirectories and files in a specified directory. You can filter the list of names by specifying a set of file attributes. The list of names can include mapped drive letters.

```
int DlgDirListComboBox(  
    HWND hDlg,           // handle to dialog box with combo box  
    LPTSTR lpPathSpec,  // path or file name  
    int nIDComboBox,    // combo-box identifier  
    int nIDStaticPath,  // static control identifier  
    UINT uFiletype      // file attributes to display  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the combo box.

lpPathSpec

[in/out] Pointer to a buffer containing a null-terminated string that specifies an absolute path, relative path, or file name. An absolute path can begin with a drive letter (for example, d:\) or a UNC name (for example, *machinename*\sharename).

The function splits the string into a directory and a file name. The function searches the directory for names that match the file name. If the string does not specify a directory, the function searches the current directory.

If the string includes a file name, the file name must contain at least one wildcard character (? or *). If the string does not include a file name, the function behaves as if you had specified the asterisk wildcard character (*) as the file name. All names in the specified directory that match the file name and have the attributes specified by the *uFileType* parameter are added to the list displayed in the combo box.

nIDComboBox

[in] Specifies the identifier of a combo box in the *hDlg* dialog box. If this parameter is zero, **DlgDirListComboBox** does not try to fill a combo box.

nIDStaticPath

[in] Specifies the identifier of a static control in the *hDlg* dialog box.

DlgDirListComboBox sets the text of this control to display the current drive and directory. This parameter can be zero if you do not want to display the current drive and directory.

uFiletype

[in] A set of bit flags that specify the attributes of the files or directories to be added to the combo box. This parameter can be a combination of the following values:

| Value | Meaning |
|---------------|---|
| DDL_ARCHIVE | Includes archived files. |
| DDL_DIRECTORY | Includes subdirectories, which are enclosed in brackets ([]). |
| DDL_DRIVES | All mapped drives are added to the list. Drives are listed in the form [-x-], where x is the drive letter. |
| DDL_EXCLUSIVE | Includes only files with the specified attributes. By default, read-write files are listed even if DDL_READWRITE is not specified. |
| DDL_HIDDEN | Includes hidden files. |
| DDL_POSTMSGS | If this flag is set, DlgDirListComboBox uses the PostMessage function to send messages to the combo box. If this flag is not set, DlgDirListComboBox uses the SendMessage function. |
| DDL_READONLY | Includes read-only files. |
| DDL_READWRITE | Includes read-write files with no additional attributes; this is the default setting. |
| DDL_SYSTEM | Includes system files. |

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. For example, if the string specified by *lpPathSpec* is not a valid path, the function fails. To get extended error information, call **GetLastError**.

Remarks

If *lpPathSpec* specifies a directory, **DlgDirListComboBox** changes the current directory to the specified directory before filling the combo box. The text of the static control identified by the *nIDStaticPath* parameter is set to the name of the new current directory.

DlgDirListComboBox sends the **CB_RESETCONTENT** and **CB_DIR** messages to the combo box.

Windows NT 4.0 and later: If *uFileType* includes the DDL_DIRECTORY flag and *lpPathSpec* specifies a first-level directory, such as C:\TEMP, the combo box will always include a “.” entry for the root directory. This is true even if the root directory has hidden or system attributes, and the DDL_HIDDEN and DDL_SYSTEM flags are not specified. The root directory of an NTFS volume has both hidden and system attributes.

Windows NT/2000: The list displays long file names, if any.

Windows 95: The list displays short file names (the 8.3 form). You can use the **SHGetFileInfo** or **GetFullPathName** functions to get the corresponding long file name.

! 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

Combo Boxes Overview, Combo-Box Functions, **DlgDirList**, **DlgDirSelectComboBoxEx**, **GetFullPathName**, **SHGetFileInfo**

DlgDirSelectComboBoxEx

The **DlgDirSelectComboBoxEx** function retrieves the current selection from a combo box filled by using the **DlgDirListComboBox** function. The selection is interpreted as a drive letter, file, or directory name.

```
BOOL DlgDirSelectComboBoxEx(  
    HWND hDlg,           // handle to a dialog box  
    LPTSTR lpString,     // pointer to buffer for path string  
    int nCount,         // number of characters in path string  
    int nIDComboBox     // combo-box identifier  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the combo box.

lpString

[out] Pointer to the buffer that receives the selected path.

nCount

[in] Specifies the length, in characters, of the buffer pointed to by the *lpString* parameter.

nIDComboBox

[in] Specifies the integer identifier of the combo-box control in the dialog box.

Return Values

If the current selection is a directory name, the return value is nonzero.

If the current selection is not a directory name, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the current selection specifies a directory name or drive letter, the **DlgDirSelectComboBoxEx** function removes the enclosing brackets (and hyphens, for drive letters), so the name or letter is ready to be inserted into a new path or file name. If there is no selection, the contents of the buffer pointed to by *lpString* do not change.

The **DlgDirSelectComboBoxEx** function does not allow more than one file name to be returned from a combo box.

DlgDirSelectComboBoxEx sends **CB_GETCURSEL** and **CB_GETLBTEXT** messages to the combo box.

In the Win32 API, you can use this function with all three types of combo boxes (CBS_SIMPLE, CBS_DROPDOWN, and CBS_DROPDOWNLIST).

! 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

Combo Boxes Overview, Combo-Box Functions, **CB_GETCURSEL**, **CB_GETLBTEXT**, **DlgDirListComboBox**

GetComboBoxInfo

The **GetComboBoxInfo** function retrieves information about the specified combo box.

```
BOOL GetComboBoxInfo(
    HWND hwndCombo, // handle to combo box
    PCOMBOBOXINFO pcbi // combo-box information
);
```

Parameters

hwndCombo

[in] Handle to the combo box.

pcbi

[out] Pointer to a **COMBOBOXINFO** structure that receives the information.

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**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Combo Boxes Overview, Combo-Box Functions, **COMBOBOXINFO**, **GetListBoxInfo**

Combo-Box Structures

COMBOBOXINFO

The **COMBOBOXINFO** structure contains combo-box status information.

```
typedef struct tagCOMBOBOXINFO {
    DWORD cbSize;
    RECT rcItem;
    RECT rcButton;
    DWORD stateButton;
    HWND hwndCombo;
    HWND hwndItem;
    HWND hwndList;
} COMBOBOXINFO, *PCOMBOBOXINFO, *LPCOMBOBOXINFO;
```

Members

cbSize

Specifies the size, in bytes, of the structure.

rcItem

Pointer to a **RECT** structure that specifies the coordinates of the edit box.

rcButton

Pointer to a **RECT** structure that specifies the coordinates of the button that contains the drop-down arrow.

stateButton

Specifies the combo-box button state. This parameter can be one of the following values:

| Value | Meaning |
|------------------------|---------------------------------------|
| 0 | The button exists and is not pressed. |
| STATE_SYSTEM_INVISIBLE | There is no button. |
| STATE_SYSTEM_PRESSED | The button is pressed. |

hwndCombo

Handle to the combo box.

hwndItem

Handle to the edit box.

hwndList

Handle to the drop-down list.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Combo Boxes Overview, Combo-Box Structures, **GetComboBoxInfo**, **RECT**

COMPAREITEMSTRUCT

The **COMPAREITEMSTRUCT** structure supplies the identifiers and application-supplied data for two items in a sorted, owner-drawn list box or combo box.

Whenever an application adds a new item to an owner-drawn list box or combo box created with the CBS_SORT or LBS_SORT style, the system sends the owner a **WM_COMPAREITEM** message. The *lParam* parameter of the message contains a long pointer to a **COMPAREITEMSTRUCT** structure. Upon receiving the message, the owner compares the two items and returns a value indicating which item sorts before the other.

```
typedef struct tagCOMPAREITEMSTRUCT {
    UINT    CtlType;
    UINT    CtlID;
    HWND    hwndItem;
    UINT    itemID1;
    ULONG_PTR itemData1;
    UINT    itemID2;
```

```
ULONG_PTR itemData2;  
DWORD dwLocaleId  
} COMPAREITEMSTRUCT;
```

Members

CtlType

Specifies either ODT_LISTBOX (an owner-drawn list box) or ODT_COMBOBOX (an owner-drawn combo box).

CtlID

Specifies the identifier of the list box or combo box.

hwndItem

Handle to the control.

itemID1

Specifies the index of the first item in the list box or combo box being compared.

This member will be -1 if the item has not been inserted, or when searching for a potential item in the list box or combo box.

itemData1

Specifies application-supplied data for the first item being compared. This value was passed as the *lParam* parameter of the message that added the item to the list box or combo box.

itemID2

Specifies the index of the second item in the list box or combo box being compared.

itemData2

Specifies application-supplied data for the second item being compared. This value was passed as the *lParam* parameter of the message that added the item to the list box or combo box.

This member will be -1 if the item has not been inserted, or when searching for a potential item in the list box or combo box.

dwLocaleId

Specifies the locale identifier. To create a locale identifier, use the **MAKELCID** 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 `winuser.h`; include `windows.h`.

+ See Also

Combo Boxes Overview, Combo-Box Structures, **MAKELCID**, **WM_COMPAREITEM**

DRAWITEMSTRUCT

The **DRAWITEMSTRUCT** structure provides information the owner window must have to determine how to paint an owner-drawn control or menu item. The owner window of the owner-drawn control or menu item receives a pointer to this structure as the *IParam* parameter of the **WM_DRAWITEM** message.

```
typedef struct tagDRAWITEMSTRUCT {
    UINT    CtlType;
    UINT    CtlID;
    UINT    ItemID;
    UINT    ItemAction;
    UINT    ItemState;
    HWND    hwndItem;
    HDC     hdc;
    RECT    rcItem;
    ULONG_PTR ItemData;
} DRAWITEMSTRUCT;
```

Members

CtlType

Specifies the control type. This member can be one of the values shown in the following table:

| Value | Meaning |
|--------------|----------------------------|
| ODT_BUTTON | Owner-drawn button |
| ODT_COMBOBOX | Owner-drawn combo box |
| ODT_LISTBOX | Owner-drawn list box |
| ODT_LISTVIEW | List view control |
| ODT_MENU | Owner-drawn menu item |
| ODT_STATIC | Owner-drawn static control |
| ODT_TAB | Tab control |

CtlID

Specifies the identifier of the combo box, list box, button, or static control. This member is not used for a menu item.

itemID

Specifies the menu item identifier for a menu item or the index of the item in a list box or combo box. For an empty list box or combo box, this member can be -1 . This allows the application to draw only the focus rectangle at the coordinates specified by the **rcItem** member, even though there are no items in the control. This indicates to the user whether the list box or combo box has the focus. How the bits are set in the **itemAction** member determines whether the rectangle is to be drawn as though the list box or combo box has the focus.

itemAction

Specifies the drawing action required. This member can be one or more of the values shown in the following table:

| Value | Meaning |
|----------------|--|
| ODA_DRAWENTIRE | The entire control needs to be drawn. |
| ODA_FOCUS | The control has lost or gained the keyboard focus. The itemState member should be checked to determine whether the control has the focus. |
| ODA_SELECT | The selection status has changed. The itemState member should be checked to determine the new selection state. |

itemState

Specifies the visual state of the item after the current drawing action takes place. This member can be a combination of the values shown in the following table:

| Value | Meaning |
|------------------|--|
| ODS_CHECKED | The menu item is to be checked. This bit is used only in a menu. |
| ODS_COMBOBOXEDIT | The drawing takes place in the selection field (edit control) of an owner-drawn combo box. |
| ODS_DEFAULT | The item is the default item. |
| ODS_DISABLED | The item is to be drawn as disabled. |
| ODS_FOCUS | The item has the keyboard focus. |
| ODS_GRAYED | The item is to be dimmed. This bit is used only in a menu. |
| ODS_HOTLIGHT | Windows 98, Windows 2000: The item is being hot-tracked; that is, the item will be highlighted when the mouse is on the item. |
| ODS_INACTIVE | Windows 98, Windows 2000: The item is inactive, and the window associated with the menu is inactive. |
| ODS_NOACCEL | Windows 2000: The control is drawn without the keyboard accelerator cues. |
| ODS_NOFOCUSRECT | Windows 2000: The control is drawn without focus indicator cues. |
| ODS_SELECTED | The menu item's status is selected. |

hwndItem

Handle to the control for combo boxes, list boxes, buttons, and static controls. For menus, this member is a handle to the menu containing the item.

hDC

Handle to a device context; this device context must be used when performing drawing operations on the control.

rcItem

Specifies a rectangle that defines the boundaries of the control to be drawn. This rectangle is in the device context specified by the **hDC** member. The system automatically clips anything that the owner window draws in the device context for combo boxes, list boxes, and buttons, but does not clip menu items. When drawing menu items, the owner window must not draw outside the boundaries of the rectangle defined by the **rcItem** member.

itemData

Specifies the application-defined value associated with the menu item. For a control, this parameter specifies the value last assigned to the list box or combo box by the **LB_SETITEMDATA** or **CB_SETITEMDATA** message. If the list box or combo box has the **LBS_HASSTRINGS** or **CBS_HASSTRINGS** style, this value is initially zero. Otherwise, this value is initially the value that was passed to the list box or combo box in the *lParam* parameter of one of the following messages:

- **CB_ADDSTRING**
- **CB_INSERTSTRING**
- **LB_ADDSTRING**
- **LB_INSERTSTRING**

If **ctlType** is **ODT_BUTTON** or **ODT_STATIC**, then **itemData** 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 `winuser.h`; include `windows.h`.

+ See Also

Combo Boxes Overview, Combo-Box Structures, **CB_ADDSTRING**, **CB_INSERTSTRING**, **CB_SETITEMDATA**, **LB_ADDSTRING**, **LB_INSERTSTRING**, **LB_SETITEMDATA**, **WM_DRAWITEM**

MEASUREITEMSTRUCT

The **MEASUREITEMSTRUCT** structure informs the system of the dimensions of an owner-drawn control or menu item. This allows the system to process user interaction with the control correctly.

```
typedef struct tagMEASUREITEMSTRUCT {
    UINT    CtlType;
    UINT    CtlID;
    UINT    itemID;
    UINT    itemWidth;
    UINT    itemHeight;
    ULONG_PTR itemData;
} MEASUREITEMSTRUCT;
```

Members

CtlType

Specifies the control type. This member can be one of the values shown in the following table:

| Value | Meaning |
|--------------|-------------------------------|
| ODT_BUTTON | Owner-drawn button |
| ODT_COMBOBOX | Owner-drawn combo box |
| ODT_LISTBOX | Owner-drawn list box |
| ODT_LISTVIEW | Owner-drawn list view control |
| ODT_MENU | Owner-drawn menu |

CtlID

Specifies the identifier of the combo box, list box, or button. This member is not used for a menu.

itemID

Specifies the identifier for a menu item or the position of a list-box or combo-box item. This value is specified for a list box only if it has the `LBS_OWNERDRAWVARIABLE` style; this value is specified for a combo box only if it has the `CBS_OWNERDRAWVARIABLE` style.

itemWidth

Specifies the width, in pixels, of a menu item. Before returning from the message, the owner of the owner-drawn menu item must fill this member.

itemHeight

Specifies the height, in pixels, of an individual item in a list box or a menu. Before returning from the message, the owner of the owner-drawn combo box, list box, or menu item must fill out this member.

itemData

Specifies the application-defined value associated with the menu item. For a control, this member specifies the value last assigned to the list box or combo box by the `LB_SETITEMDATA` or `CB_SETITEMDATA` message. If the list box or combo box has the `LB_HASSTRINGS` or `CB_HASSTRINGS` style, this value is initially zero. Otherwise, this value is initially the value passed to the list box or combo box in the *lParam* parameter of one of the following messages:

- **CB_ADDSTRING**
- **CB_INSERTSTRING**
- **LB_ADDSTRING**
- **LB_INSERTSTRING**

Remarks

The owner window of an owner-drawn control receives a pointer to the **MEASUREITEMSTRUCT** structure as the *lParam* parameter of a **WM_MEASUREITEM** message. The owner-drawn control sends this message to its owner window when the control is created, then the owner fills in the appropriate members in the structure for the control and returns. This structure is common to all owner-drawn controls.

If an application does not fill the appropriate members of **MEASUREITEMSTRUCT**, the control or menu item might not be drawn properly.

! 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

Combo Boxes Overview, Combo-Box Structures, **CB_ADDSTRING**, **CB_INSERTSTRING**, **CB_SETITEMDATA**, **LB_ADDSTRING**, **LB_INSERTSTRING**, **LB_SETITEMDATA**, **WM_MEASUREITEM**

Combo-Box Messages

CB_ADDSTRING

An application sends a **CB_ADDSTRING** message to add a string to the list box of a combo box. If the combo box does not have the **CBS_SORT** style, the string is added to the end of the list. Otherwise, the string is inserted into the list, and the list is sorted.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_ADDSTRING,         // message to send
    (LPARAM) wParam,      // not used; must be zero
    (LPARAM) lParam;      // string to add (LPCTSTR)
);
```

Parameters

wParam

This parameter is not used.

lParam

Pointer to the null-terminated string to be added. If you create the combo box with an owner-drawn style but without the CBS_HASSTRINGS style, the value of the *lParam* parameter is stored as item data instead of the string to which it would otherwise point. The item data can be retrieved or modified by sending the **CB_GETITEMDATA** or **CB_SETITEMDATA** message.

Return Values

The return value is the zero-based index to the string in the list box of the combo box. If an error occurs, the return value is **CB_ERR**. If insufficient space is available to store the new string, the return value is **CB_ERRSPACE**.

Remarks

If you create an owner-drawn combo box with the CBS_SORT style, but without the CBS_HASSTRINGS style, the **WM_COMPAREITEM** message is sent one or more times to the owner of the combo box, so the new item can be placed properly in the list.

To insert a string at a specific location within the list, use the **CB_INSERTSTRING** message.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_DIR**, **CB_INSERTSTRING**, **WM_COMPAREITEM**

CB_DELETESTRING

An application sends a **CB_DELETESTRING** message to delete a string in the list box of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_DELETESTRING,      // message to send
```

```
(WPARAM) wParam:      // item to delete
(LPARAM) lParam:      // not used; must be zero
);
```

Parameters

wParam

Specifies the zero-based index of the string to delete.

lParam

This parameter is not used.

Return Values

The return value is a count of the strings remaining in the list. If the *wParam* parameter specifies an index greater than the number of items in the list, the return value is `CB_ERR`.

Remarks

If you create the combo box with an owner-drawn style, but without the `CBS_HASSTRINGS` style, the system sends a **WM_DELETEITEM** message to the owner of the combo box, so the application can free any additional data associated with the item.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_RESETCONTENT**, **WM_DELETEITEM**

CB_DIR

An application sends a **CB_DIR** message to a combo box to add names to the list displayed by the combo box. The message adds the names of directories and files that match a specified string and set of file attributes. **CB_DIR** also can add mapped drive letters to the list.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,      // handle to destination window
    CB_DIR,          // message to send
```

```

(WPARAM) wParam;           // file attributes (UINT)
(LPARAM) lParam;           // file name (LPCTSTR)
);

```

Parameters

wParam

Specifies the attributes of the files or directories to be added to the combo box. This parameter can be one or more of the following values:

| Value | Meaning |
|---------------|--|
| DDL_ARCHIVE | Includes archived files. |
| DDL_DIRECTORY | Includes subdirectories, which are enclosed in brackets ([]). |
| DDL_DRIVES | All mapped drives are added to the list. Drives are listed in the form [-x-], where x is the drive letter. |
| DDL_EXCLUSIVE | Includes only files with the specified attributes. By default, read-write files are listed even if DDL_READWRITE is not specified. |
| DDL_HIDDEN | Includes hidden files. |
| DDL_READONLY | Includes read-only files. |
| DDL_READWRITE | Includes read-write files with no additional attributes. This is the default. |
| DDL_SYSTEM | Includes system files. |

lParam

Pointer to a null-terminated string that specifies an absolute path, relative path, or file name. An absolute path can begin with a drive letter (for example, d:\) or a UNC name (for example, \\machinename\sharename).

If the string specifies a file name or directory that has the attributes specified by the *wParam* parameter, the file name or directory is added to the list. If the file or directory name contains wildcard characters (? or *), all files or directories that match the wildcard expression and have the attributes specified by the *wParam* parameter are added to the list displayed in the combo box.

Return Values

If the message succeeds, the return value is the zero-based index of the last name added to the list.

If an error occurs, the return value is CB_ERR. If there is insufficient space to store the new strings, the return value is CB_ERRSPACE.

Remarks

Windows NT 4.0 and later: If *wParam* includes the DDL_DIRECTORY flag and *lParam* specifies all the subdirectories of a first-level directory, such as C:\TEMP*, the list box will always include a “..” entry for the root directory. This is true even if the root directory

has hidden or system attributes, and the DDL_HIDDEN and DDL_SYSTEM flags are not specified. The root directory of an NTFS volume has hidden and system attributes.

Windows NT/2000: The list displays long file names, if any.

Windows 95: The list displays short file names (the 8.3 form). You can use the **SHGetFileInfo** or **GetFullPathName** functions to get the corresponding long file name.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_ADDSTRING**, **CB_INSERTSTRING**, **DlgDirListComboBox**

CB_FINDSTRING

An application sends a **CB_FINDSTRING** message to search the list box of a combo box for an item beginning with the characters in a specified string.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_FINDSTRING,        // message to send
    (WPARAM) wParam,     // index of item preceding start
    (LPARAM) lParam;     // search string (LPCSTR)
);
```

Parameters

wParam

Specifies the zero-based index of the item preceding the first item to be searched.

When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *wParam* parameter. If *wParam* is -1, the entire list box is searched from the beginning.

lParam

Pointer to the null-terminated string that contains the characters for which to search. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Values

The return value is the zero-based index of the matching item. If the search is unsuccessful, it is `CB_ERR`.

Remarks

If you create the combo box with an owner-drawn style, but without the `CBS_HASSTRINGS` style, what the **CB_FINDSTRING** message does depends on whether your application uses the `CBS_SORT` style. If you use the `CBS_SORT` style, **WM_COMPAREITEM** messages are sent to the owner of the combo box to determine which item matches the specified string. If you do not use the `CBS_SORT` style, the **CB_FINDSTRING** message searches for a list item that matches the value of the *IParam* 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 `winuser.h`; include `windows.h`.

+ See Also

Combo Boxes Overview, Combo-Box Messages, **CB_FINDSTRINGEXACT**, **CB_SELECTSTRING**, **CB_SETCURSEL**, **WM_COMPAREITEM**

CB_FINDSTRINGEXACT

An application sends a **CB_FINDSTRINGEXACT** message to find the first list-box string in a combo box that matches the string specified in the *IParam* parameter.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_FINDSTRINGEXACT,   // message to send
    (WPARAM) wParam,     // index of item preceding start
    (LPARAM) lParam;     // search string (LPCSTR)
);
```

Parameters

wParam

Specifies the zero-based index of the item preceding the first item to be searched.

When the search reaches the bottom of the list box, it continues from the top of the list box back to the item specified by the *wParam* parameter. If *wParam* is `-1`, the entire list box is searched from the beginning.

IParam

Pointer to the null-terminated string for which to search. This string can contain a complete file name, including the extension. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Values

The return value is the zero-based index of the matching item. If the search is unsuccessful, it is `CB_ERR`.

Remarks

If you create the combo box with an owner-drawn style, but without the `CBS_HASSTRINGS` style, what the `CB_FINDSTRINGEXACT` message does depends on whether your application uses the `CBS_SORT` style. If you use the `CBS_SORT` style, `WM_COMPAREITEM` messages are sent to the owner of the combo box to determine which item matches the specified string. If you do not use the `CBS_SORT` style, the `CB_FINDSTRINGEXACT` message searches for a list item that matches the value of the *IParam* 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 `winuser.h`; include `windows.h`.

+ See Also

Combo Boxes Overview, Combo-Box Messages, `CB_FINDSTRING`, `CB_SELECTSTRING`, `WM_COMPAREITEM`

CB_GETCOUNT

An application sends a `CB_GETCOUNT` message to retrieve the number of items in the list box of a combo box.

To send this message, call the `SendMessage` function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_GETCOUNT,        // message to send
    (WPARAM) wParam,     // not used; must be zero
    (LPARAM) lParam,     // not used; must be zero
);
```

Parameters

This message has no parameters.

Return Values

The return value is the number of items in the list box. If an error occurs, it is `CB_ERR`.

Remarks

The index is zero-based, so the returned count is one greater than the index value of the last item.

! 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

Combo Boxes Overview, Combo-Box Messages

CB_GETCURSEL

An application sends a **CB_GETCURSEL** message to retrieve the index of the currently selected item, if any, in the list box of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_GETCURSEL,         // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

The return value is the zero-based index of the currently selected item. If no item is selected, it is `CB_ERR`.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_SELECTSTRING**, **CB_SETCURSEL**

CB_GETDROPPEDCONTROLRECT

An application sends a **CB_GETDROPPEDCONTROLRECT** message to retrieve the screen coordinates of the drop-down list box of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_GETDROPPEDCONTROLRECT, // message to send
    (WPARAM) wParam,     // not used, must be zero
    (LPARAM) lParam;     // screen coordinates (RECT *)
);
```

Parameters

wParam

This parameter is not used.

lParam

Pointer to the **RECT** structure that receives the coordinates.

Return Values

If the message succeeds, the return value is nonzero.

If the message 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 winuser.h; include windows.h.

+ See Also

Combo Boxes Overview, Combo-Box Messages, **RECT**

CB_GETDROPPEDSTATE

An application sends a **CB_GETDROPPEDSTATE** message to determine whether the list box of a combo box is dropped down.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_GETDROPPEDSTATE,   // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

If the list box is visible, the return value is **TRUE**; otherwise, it is **FALSE**.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_SHOWDROPDOWN**

CB_GETDROPPEDWIDTH

An application sends the **CB_GETDROPPEDWIDTH** message to retrieve the minimum allowable width, in pixels, of the list box of a combo box with either the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_GETDROPPEDWIDTH,   // message to send
    (WPARAM) wParam,     // not used; must be zero
    (LPARAM) lParam:     // not used; must be zero
);
```

Parameters

This message has no parameters.

Return Values

If the message succeeds, the return value is the width, in pixels.

If the message fails, the return value is `CB_ERR`.

Remarks

By default, the minimum allowable width of the drop-down list box is zero. The width of the list box is either the minimum width allowable or the combo-box width, whichever is larger.

Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Combo Boxes Overview, Combo-Box Messages, **CB_SETDROPPEDWIDTH**

CB_GETEDITSEL

An application sends a **CB_GETEDITSEL** message to get the starting and ending character positions of the current selection in the edit control of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_GETEDITSEL,       // message to send
    (WPARAM) wParam,     // starting position (LPDWORD)
    (LPARAM) lParam:     // ending position (LPDWORD)
);
```

Parameters

wParam

Pointer to a **DWORD** value that receives the starting position of the selection. This parameter can be NULL.

lParam

Pointer to a **DWORD** value that receives the ending position of the selection. This parameter can be NULL.

Return Values

The return value is a zero-based **DWORD** value with the starting position of the selection in the low-order word, and with the ending position of the first character after the last selected character in the high-order word.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_SETEXTSEL**, **HIWORD**, **LOWORD**

CB_GETTEXTENDEDUI

An application sends a **CB_GETTEXTENDEDUI** message to determine whether a combo box has the default user interface or the extended user interface.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_GETTEXTENDEDUI,    // message to send  
    (WPARAM) wParam,     // not used; must be zero  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

If the combo box has the extended user interface, the return value is TRUE; otherwise, it is FALSE.

Remarks

By default, the F4 key opens or closes the list, and the DOWN ARROW key changes the current selection. In a combo box with the extended user interface, the F4 key is disabled, and pressing the DOWN ARROW key opens the drop-down list.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_SETEXTENDEDUI**

CB_GETHORIZONTALTEXT

An application sends the **CB_GETHORIZONTALTEXT** message to retrieve from a combo box the width, in pixels, by which the list box can be scrolled horizontally (the scrollable width). This is applicable only if the list box has a horizontal scroll bar.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_GETHORIZONTALTEXT, // message to send  
    (WPARAM) wParam;     // not used; must be zero  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

The return value is the scrollable width, in pixels.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Combo Boxes Overview, Combo-Box Messages, **CB_SETHORIZONTALEXTENT**

CB_GETITEMDATA

An application sends a **CB_GETITEMDATA** message to a combo box to retrieve the application-supplied value associated with the specified item in the combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_GETITEMDATA,       // message to send
    (WPARAM) wParam,     // item index
    (LPARAM) lParam:     // not used; must be zero
);
```

Parameters

wParam

Specifies the zero-based index of the item.

lParam

This parameter is not used.

Return Values

The return value is the value associated with the item. If an error occurs, it is **CB_ERR**.

If the item is in an owner-drawn combo box created without the **CBS_HASSTRINGS** style, the return value is the value contained in the *lParam* parameter of the **CB_ADDSTRING** or **CB_INSERTSTRING** message that added the item to the combo box. If the **CBS_HASSTRINGS** style was not used, the return value is the *lParam* parameter contained in a **CB_SETITEMDATA** message.

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

Combo Boxes Overview, Combo-Box Messages, **CB_ADDSTRING**, **CB_INSERTSTRING**, **CB_SETITEMDATA**

CB_GETITEMHEIGHT

An application sends a **CB_GETITEMHEIGHT** message to determine the height of list items or the selection field in a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_GETITEMHEIGHT,     // message to send  
    (WPARAM) wParam,     // item index  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

wParam

Specifies the combo-box component whose height is to be retrieved.

This parameter must be -1 to retrieve the height of the selection field. It must be zero to retrieve the height of list items, unless the combo box has the **CBS_OWNERDRAWVARIABLE** style. In that case, the *wParam* parameter is the zero-based index of a specific list item.

lParam

This parameter is not used.

Return Values

The return value is the height, in pixels, of the list items in a combo box. If the combo box has the **CBS_OWNERDRAWVARIABLE** style, it is the height of the item specified by the *wParam* parameter. If *wParam* is -1 , the return value is the height of the edit-control (or static-text) portion of the combo box. If an error occurs, the return value is **CB_ERR**.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_SETITEMHEIGHT**, **WM_MEASUREITEM**

CB_GETLBTEXT

An application sends a **CB_GETLBTEXT** message to retrieve a string from the list of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_GETLBTEXT,         // message to send  
    (WPARAM) wParam,      // item index  
    (LPARAM) lParam;      // receives string (LPCSTR)  
);
```

Parameters

wParam

Specifies the zero-based index of the string to retrieve.

lParam

Pointer to the buffer that receives the string. The buffer must have sufficient space for the string and a terminating null character. You can send a **CB_GETLBTEXTLEN** message prior to the **CB_GETLBTEXT** message to retrieve the length, in bytes, of the string.

Return Values

The return value is the length of the string, in bytes, excluding the terminating null character. If *wParam* does not specify a valid index, the return value is **CB_ERR**.

Remarks

If you create the combo box with an owner-drawn style, but without the **CBS_HASSTRINGS** style, the buffer pointed to by *lParam* receives the data associated with the item.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_GETLBTEXTLEN**

CB_GETLBTEXTLEN

An application sends a **CB_GETLBTEXTLEN** message to retrieve the length, in characters, of a string in the list of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_GETLBTEXTLEN,      // message to send  
    (WPARAM) wParam,      // item index  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

Specifies the zero-based index of the string.

lParam

This parameter is not used.

Return Values

The return value is the length of the string, in characters, excluding the terminating null character. Under certain conditions, this value may actually be greater than the length of the text. For more information, see the Remarks section.

If the *wParam* parameter does not specify a valid index, the return value is **CB_ERR**.

Remarks

Under certain conditions, the return value is larger than the actual length of the text. This occurs with certain mixtures of ANSI and Unicode, and is due to the operating system allowing for the possible existence of double-byte character set (DBCS) characters within the text. The return value, however, always will be at least as large as the actual length of the text; so, you can use it always to guide buffer allocation. This behavior can occur when an application uses both ANSI functions and common dialog boxes, which use Unicode.

To obtain the exact length of the text, use the **WM_GETTEXT**, **LB_GETTEXT**, or **CB_GETLBTEXT** message, or the **GetWindowText** 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.

+ See Also

Combo Boxes Overview, Combo-Box Messages, **CB_GETLBTEXT**, **GetWindowText**, **LB_GETTEXT**, **WM_GETTEXT**

CB_GETLOCALE

An application sends a **CB_GETLOCALE** message to retrieve the current locale of the combo box. The locale is used to determine the correct sorting order of both displayed text for combo boxes with the CBS_SORT style and text added by using the **CB_ADDSTRING** message.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_GETLOCALE,         // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam,      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

The return value specifies the current locale of the combo box. The high word contains the country/region code, and the low-order word contains the language identifier.

Remarks

The language identifier is made up of a sublanguage identifier and a primary language identifier. The **PRIMARYLANGID** macro obtains the primary language identifier, and the **SUBLANGID** macro obtains the sublanguage identifier.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_ADDSTRING**, **CB_SETLOCALE**, **HIWORD**, **LOWORD**, **PRIMARYLANGID**, **SUBLANGID**

CB_GETTOPINDEX

An application sends the **CB_GETTOPINDEX** message to retrieve the zero-based index of the first visible item in the list-box portion of a combo box. Initially, the item with index 0 is at the top of the list box, but if the list-box contents have been scrolled, another item may be at the top.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_GETTOPINDEX,       // message to send  
    (WPARAM) wParam,      // not used: must be zero  
    (LPARAM) lParam,      // not used: must be zero  
);
```

Parameters

This message has no parameters.

Return Values

If the message is successful, the return value is the index of the first visible item in the list box of the combo box.

If the message fails, the return value is **CB_ERR**.

Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Combo Boxes Overview, Combo-Box Messages, **CB_SETTOPINDEX**

CB_INITSTORAGE

An application sends the **CB_INITSTORAGE** message before adding a large number of items to the list-box portion of a combo box. This message allocates memory for storing list-box items.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_INITSTORAGE,       // message to send  
    (LPARAM) wParam,      // number of items to add  
    (LPARAM) lParam;      // memory to allocate  
);
```

Parameters

wParam

Specifies the number of items to add.

lParam

Specifies the amount of memory to allocate for item strings, in bytes.

Return Values

If the message is successful, the return value is the total number of items for which memory has been pre-allocated; that is, the total number of items added by all successful **CB_INITSTORAGE** messages.

If the message fails, the return value is **CB_ERRSPACE**.

Windows NT/ 2000: For Windows NT version 4.0, this message does not allocate the specified amount of memory; however, it always returns the value specified in the *wParam* parameter. For Windows 2000, the message allocates memory, and returns the success and error values described above.

Remarks

The **CB_INITSTORAGE** message helps speed up the initialization of combo boxes that have a large number of items (over 100). It reserves the specified amount of memory, so that subsequent **CB_ADDSTRING**, **CB_INSERTSTRING**, and **CB_DIR** messages take the shortest time possible. You can use estimates for the *wParam* and *lParam* parameters. If you overestimate, the extra memory is allocated; if you underestimate, the normal allocation is used for items that exceed the requested amount.

! Requirements

Windows NT/2000: Requires Windows 2000.

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

Combo Boxes Overview, Combo-Box Messages, **CB_ADDSTRING**, **CB_DIR**, **CB_INSERTSTRING**

CB_INSERTSTRING

An application sends a **CB_INSERTSTRING** message to insert a string into the list box of a combo box. Unlike the **CB_ADDSTRING** message, the **CB_INSERTSTRING** message does not cause a list with the **CBS_SORT** style to be sorted.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_INSERTSTRING,      // message to send  
    (WPARAM) wParam,     // item index  
    (LPARAM) lParam;     // string to insert (LPCTSTR)  
);
```

Parameters

wParam

Specifies the zero-based index of the position at which to insert the string. If this parameter is -1 , the string is added to the end of the list.

lParam

Pointer to the null-terminated string to be inserted. If you create the combo box with an owner-drawn style, but without the **CBS_HASSTRINGS** style, the value of the *lParam* parameter is stored, instead of the string to which it would otherwise point.

Return Values

The return value is the index of the position at which the string was inserted. If an error occurs, the return value is **CB_ERR**. If there is insufficient space available to store the new string, it is **CB_ERRSPACE**.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_ADDSTRING**, **CB_DIR**

CB_LIMITTEXT

An application sends a **CB_LIMITTEXT** message to limit the length of the text the user can type into the edit control of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_LIMITTEXT,         // message to send  
    (WPARAM) wParam;      // max characters  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

Specifies the maximum number of characters the user can enter, not including the null terminator. If this parameter is zero, the text length is limited to 0x7FFFFFFE characters.

lParam

This parameter is not used.

Return Values

The return value is always TRUE.

Remarks

If the combo box does not have the CBS_AUTOHSCROLL style, setting the text limit to be larger than the size of the edit control has no effect.

The **CB_LIMITTEXT** message limits only the text the user can enter. It has no effect on any text already in the edit control when the message is sent, and it has no effect on the length of the text copied to the edit control when a string in the list box is selected.

The default limit to the text a user can enter in the edit control is 30,000 characters.

! 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

Combo Boxes Overview, Combo-Box Messages

CB_RESETCONTENT

An application sends a **CB_RESETCONTENT** message to remove all items from the list box and edit control of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_RESETCONTENT,      // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

This message always returns **CB_OKAY**.

Remarks

If you create the combo box with an owner-drawn style, but without the **CBS_HASSTRINGS** style, the owner of the combo box receives a **WM_DELETEITEM** message for each item in the combo box.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_DELETESTRING**, **WM_DELETEITEM**

CB_SELECTSTRING

An application sends a **CB_SELECTSTRING** message to search the list of a combo box for an item that begins with the characters in a specified string. If a matching item is found, it is selected and copied to the edit control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    CB_SELECTSTRING,      // message to send
    (LPARAM) wParam,      // item of item preceding start
    (LPARAM) lParam;      // search string
);
```

Parameters

wParam

Specifies the zero-based index of the item preceding the first item to be searched. When the search reaches the bottom of the list, it continues from the top of the list back to the item specified by the *wParam* parameter. If *wParam* is -1 , the entire list is searched from the beginning.

lParam

Pointer to the null-terminated string that contains the characters for which to search. The search is not case-sensitive, so this string can contain any combination of uppercase and lowercase letters.

Return Values

If the string is found, the return value is the index of the selected item. If the search is unsuccessful, the return value is `CB_ERR`, and the current selection is not changed.

Remarks

A string is selected only if the characters from the starting point match the characters in the prefix string.

If you create the combo box with an owner-drawn style, but without the `CBS_HASSTRINGS` style, what the `CB_SELECTSTRING` message does depends on whether you use the `CBS_SORT` style. If the `CBS_SORT` style is used, the system sends `WM_COMPAREITEM` messages to the owner of the combo box to determine which item matches the specified string. If you do not use the `CBS_SORT` style, `CB_SELECTSTRING` attempts to match the `DWORD` value against the value of the *lParam* 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 `winuser.h`; include `windows.h`.

See Also

Combo Boxes Overview, Combo-Box Messages, `CB_FINDSTRING`, `CB_FINDSTRINGEXACT`, `CB_SETCURSEL`, `WM_COMPAREITEM`

CB_SETCURSEL

An application sends a **CB_SETCURSEL** message to select a string in the list of a combo box. If necessary, the list scrolls the string into view. The text in the edit control of the combo box changes to reflect the new selection, and any previous selection in the list is removed.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_SETCURSEL,        // message to send  
    (WPARAM) wParam;     // item index  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

wParam

Specifies the zero-based index of the string to select. If this parameter is -1 , any current selection in the list is removed, and the edit control is cleared.

lParam

This parameter is not used.

Return Values

If the message is successful, the return value is the index of the item selected. If *wParam* is greater than the number of items in the list or if *wParam* is -1 , the return value is **CB_ERR**, and the selection is cleared.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_FINDSTRING**, **CB_GETCURSEL**, **CB_SELECTSTRING**

CB_SETDROPPEDWIDTH

An application sends the **CB_SETDROPPEDWIDTH** message to set the maximum width allowable, in pixels, of the list box of a combo box with either the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_SETDROPPEDWIDTH,   // message to send  
    (WPARAM) wParam;     // width of list box  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

wParam

Specifies the width of the list box, in pixels.

lParam

This parameter is not used.

Return Values

If the message is successful, the return value is the new width of the list box.

If the message fails, the return value is `CB_ERR`.

Remarks

By default, the minimum allowable width of the drop-down list box is zero. The width of the list box is either the minimum width allowable or the combo-box width, whichever is larger.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Combo Boxes Overview, Combo-Box Messages, `CB_GETDROPPEDWIDTH`

CB_SETEDITSEL

An application sends a `CB_SETEDITSEL` message to select characters in the edit control of a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    CB_SETEDITSEL,        // message to send  
    (WPARAM) wParam;      // not used; must be zero  
    (LPARAM) lParam;      // start and end positions  
);
```

Parameters

wParam

This parameter is not used.

lParam

The low-order word of *lParam* specifies the starting position. If the low-order word is -1, the selection, if any, is removed.

The high-order word of *lParam* specifies the ending position. If the high-order word is -1, all text from the starting position to the last character in the edit control is selected.

Return Values

If the message succeeds, the return value is TRUE. If the message is sent to a combo box with the CBS_DROPDOWNLIST style, it is CB_ERR.

Remarks

The positions are zero-based. The first character of the edit control is in the zero position. The first character after the last selected character is in the ending position. For example, to select the first four characters of the edit control, use a starting position of 0 and an ending position of 4.

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

Combo Boxes Overview, Combo-Box Messages, **CB_GETEDITSEL**, **MAKELPARAM**

CB_SETEXTENDEDUI

An application sends a **CB_SETEXTENDEDUI** message to select either the default user interface or extended user interface for a combo box that has either the CBS_DROPDOWN or CBS_DROPDOWNLIST style.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    CB_SETEXTENDEDUI,     // message to send
    (WPARAM) wParam,     // user interface type
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies whether the combo box uses the extended user interface or the default user interface. A value of TRUE selects the extended user interface; a value of FALSE selects the standard user interface.

lParam

This parameter is not used.

Return Values

If the operation succeeds, the return value is CB_OKAY. If an error occurs, it is CB_ERR.

Remarks

By default, the F4 key opens or closes the list, and the DOWN ARROW key changes the current selection. In the extended user interface, the F4 key is disabled, and the DOWN ARROW key opens the drop-down list.

! 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

Combo Boxes Overview, Combo-Box Messages, **CB_GETEXTENDEDUI**

CB_SETHORIZONTALTEXT

An application sends the **CB_SETHORIZONTALTEXT** message to set the width, in pixels, by which a list box can be scrolled horizontally (the scrollable width). If the width of the list box is smaller than this value, the horizontal scroll bar horizontally scrolls items in the list box. If the width of the list box is equal to or greater than this value, the horizontal scroll bar is hidden or, if the combo box has the CBS_DISABLENOSCROLL style, disabled.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_SETHORIZONTALEXTENT, // message to send
    (WPARAM) wParam,     // scrollable width
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies the scrollable width of the list box, in pixels.

lParam

This parameter is not used.

Return Values

No return value.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Combo Boxes Overview, Combo-Box Messages, **CB_GETHORIZONTALEXTENT**

CB_SETITEMDATA

An application sends a **CB_SETITEMDATA** message to set the value associated with the specified item in a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_SETITEMDATA,       // message to send
    (WPARAM) wParam,     // item index
    (LPARAM) lParam;     // item data (DWORD)
);
```

Parameters

wParam

Specifies the item's zero-based index.

lParam

Specifies the new value to be associated with the item.

Return Values

If an error occurs, the return value is `CB_ERR`.

Remarks

If the specified item is in an owner-drawn combo box created without the `CBS_HASSTRINGS` style, this message replaces the value in the *lParam* parameter of the `CB_ADDSTRING` or `CB_INSERTSTRING` message that added the item to the combo box.

! 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

Combo Boxes Overview, Combo-Box Messages, `CB_ADDSTRING`, `CB_GETITEMDATA`, `CB_INSERTSTRING`

CB_SETITEMHEIGHT

An application sends a `CB_SETITEMHEIGHT` message to set the height of list items or the selection field in a combo box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    CB_SETITEMHEIGHT,     // message to send
    (WPARAM) wParam,     // item index
    (LPARAM) lParam;     // item height
);
```

Parameters

wParam

Specifies the component of the combo box for which to set the height.

This parameter must be `-1` to set the height of the selection field. It must be zero to set the height of list items, unless the combo box has the `CBS_OWNERDRAWVARIABLE` style. In that case, the *wParam* parameter is the zero-based index of a specific list item.

lParam

Specifies the height, in pixels, of the combo-box component identified by *wParam*.

Return Values

If the index or height is invalid, the return value is `CB_ERR`.

Remarks

The selection-field height in a combo box is set independently from the height of the list items. An application must ensure that the height of the selection field is not smaller than the height of a particular list item.

! 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

Combo Boxes Overview, Combo-Box Messages, `CB_GETITEMHEIGHT`, `WM_MEASUREITEM`

CB_SETLOCALE

An application sends a `CB_SETLOCALE` message to set the current locale of the combo box. If the combo box has the `CBS_SORT` style and strings are added using `CB_ADDSTRING`, the locale of a combo box affects how list items are sorted.

To send this message, call the `SendMessage` function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_SETLOCALE,         // message to send  
    (LPARAM) wParam,      // locale identifier (WORD)  
    (LPARAM) lParam,      // not used; must be zero  
);
```

Parameters

wParam

Specifies the locale identifier for the combo box to use for sorting when adding text.

lParam

This parameter is not used.

Return Values

The return value is the previous locale identifier. If *wParam* specifies a locale not installed on the system, the return value is `CB_ERR`, and the current combo box locale is not changed.

Remarks

Use the `MAKELCID` macro to construct a locale identifier, and the `MAKELANGID` macro to construct a language identifier. The language identifier is made up of a primary language identifier and a sublanguage identifier.

! 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

Combo Boxes Overview, Combo-Box Messages, `CB_ADDSTRING`, `CB_GETLOCALE`, `MAKELANGID`, `MAKELCID`

CB_SETTOPINDEX

An application sends the `CB_SETTOPINDEX` message to ensure that a particular item is visible in the list box of a combo box. The system scrolls the list-box contents, so that either the specified item appears at the top of the list box or the maximum scroll range has been reached.

To send this message, call the `SendMessage` function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    CB_SETTOPINDEX,       // message to send
    (WPARAM) wParam,     // item list index
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies the zero-based index of the list item.

lParam

This parameter is not used.

Return Values

If the message is successful, the return value is zero.

If the message fails, the return value is `CB_ERR`.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Combo Boxes Overview, Combo-Box Messages, `CB_GETTOPINDEX`

CB_SHOWDROPDOWN

An application sends a `CB_SHOWDROPDOWN` message to show or hide the list box of a combo box that has either the `CBS_DROPDOWN` or `CBS_DROPDOWNLIST` style.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    CB_SHOWDROPDOWN,     // message to send  
    (WPARAM) wParam,     // show state  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

wParam

Specifies whether the drop-down list box is to be shown or hidden. A value of `TRUE` shows the list box; a value of `FALSE` hides it.

lParam

This parameter is not used.

Return Values

The return value is always `TRUE`.

Remarks

This message has no effect on a combo box created with the `CBS_SIMPLE` 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 winuser.h; include windows.h.

+ See Also

Combo Boxes Overview, Combo-Box Messages, **CB_GETDROPPEDSTATE**

CBN_CLOSEUP

The **CBN_CLOSEUP** notification message is sent when the list box of a combo box has been closed. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_COMMAND  
    WPARAM wParam,       // combo-box identifier, CBN_CLOSEUP  
    LPARAM lParam        // handle to combo box (HWND)  
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

If the user changed the current selection, the combo box also sends the **CBN_SELCHANGE** notification when the drop-down list closes. In general, you cannot predict the order in which notifications will be sent. In particular, a **CBN_SELCHANGE** notification message may occur either before or after a **CBN_CLOSEUP** notification message.

To execute a specific process each time the user selects a list item, you can handle either the **CBN_SELCHANGE** or **CBN_CLOSEUP** notification message. Typically, you would wait for the **CBN_CLOSEUP** notification before processing a change in the current selection. This can be particularly important if a significant amount of processing is required.

This notification message is not sent to a combo box that has the **CBS_SIMPLE** 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 winuser.h; include windows.h.

+ See Also

Combo Boxes Overview, Combo-Box Messages, **CBN_DROPDOWN**, **CBN_SELCHANGE**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_DBLCLK

The **CBN_DBLCLK** notification message is sent when the user double-clicks a string in the list box of a combo box. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,      // combo-box identifier, CBN_DBLCLK
    LPARAM lParam       // handle to combo box (HWND)
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

This notification message occurs only for a combo box with the **CBS_SIMPLE** style. In a combo box with the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** style, a double-click cannot occur, because a single click closes the list box.

! 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

Combo Boxes Overview, Combo-Box Messages, **CBN_SELCHANGE**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_DROPDOWN

The **CBN_DROPDOWN** notification message is sent when the list box of a combo box is about to be made visible. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_COMMAND  
    WPARAM wParam,      // combo-box identifier, CBN_DROPDOWN  
    LPARAM lParam        // handle to combo box (HWND)  
);
```

Parameters

wParam

- The low-order word specifies the control identifier of the combo box.
- The high-order word specifies the notification message.

lParam

- Handle to the combo box.

Remarks

This notification message is sent only if the combo box has either the **CBS_DROPDOWN** or **CBS_DROPDOWNLIST** 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 winuser.h; include windows.h.

+ See Also

Combo Boxes Overview, Combo-Box Messages, **CBN_CLOSEUP**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_EDITCHANGE

The **CBN_EDITCHANGE** notification message is sent after the user has taken an action that might have altered the text in the edit control portion of a combo box. Unlike the **CBN_EDITUPDATE** notification message, this notification message is sent after the system updates the screen. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,      // combo-box identifier, CBN_EDITCHANGE  
    LPARAM lParam       // handle to combo box (HWND)  
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

If the combo box has the CBS_DROPDOWNLIST style, this notification message is not sent.

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

Combo Boxes Overview, Combo-Box Messages, **CBN_EDITUPDATE**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_EDITUPDATE

The **CBN_EDITUPDATE** notification message is sent when the edit control portion of a combo box is about to display altered text. This notification message is sent after the control has formatted the text, but before it displays the text. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_COMMAND  
    WPARAM wParam,       // combo-box identifier, CBN_EDITUPDATE  
    LPARAM lParam        // handle to combo box (HWND)  
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

If the combo box has the CBS_DROPDOWNLIST style, this notification message is not sent.

! 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

Combo Boxes Overview, Combo-Box Messages, **CBN_EDITCHANGE**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_ERRSPACE

The **CBN_ERRSPACE** notification message is sent when a combo box cannot allocate enough memory to meet a specific request. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_COMMAND  
    WPARAM wParam,       // combo-box identifier, CBN_ERRSPACE  
    LPARAM lParam        // handle to combo box (HWND)  
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

! 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

Combo Boxes Overview, Combo-Box Messages, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_KILLFOCUS

The **CBN_KILLFOCUS** notification message is sent when a combo box loses the keyboard focus. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,      // combo-box identifier, CBN_KILLFOCUS
    LPARAM lParam       // handle to combo box (HWND)
);

```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

! 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

Combo Boxes Overview, Combo-Box Messages, **CBN_SETFOCUS**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_SELCHANGE

The **CBN_SELCHANGE** notification message is sent when the user changes the current selection in the list box of a combo box. The user can change the selection either by clicking in the list box or using the arrow keys. The parent window of the combo box receives this notification in the form of a **WM_COMMAND** message with **CBN_SELCHANGE** in the high-order word of the *wParam* parameter.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,      // combo-box identifier, CBN_SELCHANGE
    LPARAM lParam       // handle to combo box (HWND)
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

To get the index of the current selection, send the **CB_GETCURSEL** message to the control.

The **CBN_SELCHANGE** notification message is not sent when the current selection is set using the **CB_SETCURSEL** message.

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

Combo Boxes Overview, Combo-Box Messages, **CBN_CLOSEUP**, **CBN_DBLCLK**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_SELENDCANCEL

The **CBN_SELENDCANCEL** notification message is sent when the user selects an item, but then selects another control or closes the dialog box. It indicates that the user's initial selection is to be ignored. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,          // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,     // combo-box identifier, CBN_SELENDCANCEL  
    LPARAM lParam      // handle to combo box (HWND)  
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

In a combo box with the **CBS_SIMPLE** style, the **CBN_SELENDCANCEL** notification message is not sent. The **CBN_SELENDOK** notification message is sent immediately before every **CBN_SELCHANGE** notification message.

! 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

Combo Boxes Overview, Combo-Box Messages, **CBN_SELCHANGE**, **CBN_SELENDOK**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_SELENDOK

The **CBN_SELENDOK** notification message is sent when the user selects a list item, or selects an item and then closes the list. It indicates that the user's selection is to be processed. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,      // combo-box identifier, CBN_SELENDOK
    LPARAM lParam       // handle to combo box (HWND)
);
```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

Remarks

In a combo box with the **CBS_SIMPLE** style, the **CBN_SELENDOK** notification message is sent immediately before every **CBN_SELCHANGE** notification message.

! 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

Combo Boxes Overview, Combo-Box Messages, **CBN_SELCHANGE**, **CBN_SELENCANCEL**, **HIWORD**, **LOWORD**, **WM_COMMAND**

CBN_SETFOCUS

The **CBN_SETFOCUS** notification message is sent when a combo box receives the keyboard focus. The parent window of the combo box receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
```

```

UINT uMsg, // WM_COMMAND
WPARAM wParam, // combo-box identifier, CBN_SETFOCUS
LPARAM lParam // handle to combo box (HWND)
);

```

Parameters

wParam

The low-order word specifies the control identifier of the combo box.

The high-order word specifies the notification message.

lParam

Handle to the combo box.

! 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

Combo Boxes Overview, Combo-Box Messages, **CBN_KILLFOCUS**, **HIWORD**, **LOWORD**, **WM_COMMAND**

WM_COMPAREITEM

The system sends the **WM_COMPAREITEM** message to determine the relative position of a new item in the sorted list of an owner-drawn combo box or list box. Whenever the application adds a new item, the system sends this message to the owner of a combo box or list box created with either the CBS_SORT or LBS_SORT style.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd, // handle to window
    UINT uMsg, // WM_COMPAREITEM (UINT)
    WPARAM wParam, // control identifier
    LPARAM lParam // item data (LPCOMPAREITEMSTRUCT)
);

```

Parameters

wParam

Specifies the identifier of the control that sent the **WM_COMPAREITEM** message.

lParam

Pointer to a **COMPAREITEMSTRUCT** structure that contains the identifiers and application-supplied data for two items in the combo box or list box.

Return Values

The return value indicates the relative position of the two items. It may be any of the values shown in the following table:

| Value | Meaning |
|-------|---|
| -1 | Item 1 precedes item 2 in the sorted order. |
| 0 | Items 1 and 2 are equivalent in the sorted order. |
| 1 | Item 1 follows item 2 in the sorted order. |

Remarks

When the owner of an owner-drawn combo box or list box receives this message, the owner returns a value indicating which of the items specified by the **COMPAREITEMSTRUCT** structure will appear before the other. Typically, the system sends this message several times until it determines the exact position for the new item.

If a dialog-box procedure handles this message, it should cast the desired return value to a **BOOL** and return the value directly. The **DWL_MSGRESULT** value set by the **SetWindowLong** function is ignored.

! 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

Combo Boxes Overview, Combo-Box Messages, **COMPAREITEMSTRUCT**

WM_DRAWITEM

The **WM_DRAWITEM** message is sent to the owner window of an owner-drawn button, combo box, list box, or menu when a visual aspect of the button, combo box, list box, or menu has changed.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_DRAWITEM
    WPARAM wParam,       // pointer to DRAWITEMSTRUCT
    LPARAM lParam)

```

(continued)

(continued)

```
WPARAM wParam, // control identifier (UINT)
LPARAM lParam // item data (LPDRAWITEMSTRUCT)
);
```

Parameters

wParam

Specifies the identifier of the control that sent the **WM_DRAWITEM** message. If the message was sent by a menu, this parameter is zero.

lParam

Pointer to a **DRAWITEMSTRUCT** structure containing information about the item to be drawn and the type of drawing required.

Return Values

If an application processes this message, it should return TRUE.

Remarks

By default, the **DefWindowProc** function draws the focus rectangle for an owner-drawn list-box item.

The **itemAction** member of the **DRAWITEMSTRUCT** structure specifies the drawing operation that an application should perform.

Before returning from processing this message, an application should ensure that the device context identified by the **hDC** member of the **DRAWITEMSTRUCT** structure is in the default state.

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

Combo Boxes Overview, Combo-Box Messages, **DefWindowProc**, **DRAWITEMSTRUCT**

WM_MEASUREITEM

The **WM_MEASUREITEM** message is sent to the owner window of an owner-drawn button, combo box, list box, list-view control, or menu item when the control or menu is created.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MEASUREITEM
    WPARAM wParam,      // control identifier (UINT)
    LPARAM lParam       // item data (LPMEASUREITEMSTRUCT)
);

```

Parameters

wParam

Contains the value of the **CtlID** member of the **MEASUREITEMSTRUCT** structure pointed to by the *lParam* parameter. This value identifies the control that sent the **WM_MEASUREITEM** message.

If the value is zero, the message was sent by a menu. If the value is nonzero, the message was sent by either a combo box or a list box. If the value is nonzero, and the value of the **itemID** member of the **MEASUREITEMSTRUCT** pointed to by *lParam* is (UINT)-1, the message was sent by a combo-box edit field.

lParam

Pointer to a **MEASUREITEMSTRUCT** structure that contains the dimensions of the owner-drawn control or menu item.

Return Values

If an application processes this message, it should return TRUE.

Remarks

When the owner window receives the **WM_MEASUREITEM** message, the owner fills in the **MEASUREITEMSTRUCT** structure pointed to by the *lParam* parameter of the message, and returns; this informs the system of the dimensions of the control. If a list box or combo box is created with either the **LBS_OWNERDRAWVARIABLE** or **CBS_OWNERDRAWVARIABLE** style, this message is sent to the owner for each item in the control; otherwise, this message is sent once.

The system sends the **WM_MEASUREITEM** message to the owner window of combo boxes and list boxes created with the **OWNERDRAWFIXED** style before sending the **WM_INITDIALOG** message. As a result, when the owner receives this message, the system has not yet determined the height and width of the font used in the control; function calls and calculations requiring these values should occur in the main function of the application or library.

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

Combo Boxes Overview, Combo-Box Messages, **MEASUREITEMSTRUCT**, **WM_INITDIALOG**

Combo-Box Styles

To create a combo box using the **CreateWindow** or **CreateWindowEx** function, specify the **COMBOBOX** class, appropriate window-style constants, and a combination of the following combo-box styles:

| Style | Description |
|-----------------------------|---|
| CBS_AUTOHSCROLL | Automatically scrolls the text in an edit control to the right when the user types a character at the end of the line. If this style is not set, only text that fits within the rectangular boundary is allowed. |
| CBS_DISABLENOSCROLL | Shows a disabled vertical scroll bar in the list box when the box does not contain enough items to scroll. Without this style, the scroll bar is hidden when the list box does not contain enough items. |
| CBS_DROPDOWN | Similar to CBS_SIMPLE , except that the list box is not displayed unless the user selects an icon next to the edit control. |
| CBS_DROPDOWNLIST | Similar to CBS_DROPDOWN , except that the edit control is replaced by a static text item that displays the current selection in the list box. |
| CBS_HASSTRINGS | Specifies that an owner-drawn combo box contains items consisting of strings. The combo box maintains the memory and address for the strings so the application can use the CB_GETLBTEXT message to retrieve the text for a particular item. <i>For accessibility issues, see Exposing Owner-Drawn Combo-Box Items.</i> |
| CBS_LOWERCASE | Converts to lowercase all text in both the selection field and list. |
| CBS_NOINTEGRALHEIGHT | Specifies that the size of the combo box is exactly the size specified by the application when it created the combo box. Normally, the system sizes a combo box so that it does not display partial items. |

| Style | Description |
|-----------------------|--|
| CBS_OEMCONVERT | Converts text entered in the combo-box edit control from the Windows character set to the OEM character set, and then back to the Windows set. This ensures proper character conversion when the application calls the CharToOem function to convert a Windows string in the combo box to OEM characters. This style is most useful for combo boxes that contain file names, and applies only to combo boxes created with the CBS_SIMPLE or CBS_DROPDOWN style. |
| CBS_OWNERDRAWFIXED | Specifies that the owner of the list box is responsible for drawing its contents, and that the items in the list box are all the same height. The owner window receives a WM_MEASUREITEM message when the combo box is created, and a WM_DRAWITEM message when a visual aspect of the combo box has changed. |
| CBS_OWNERDRAWVARIABLE | Specifies that the owner of the list box is responsible for drawing its contents, and that the items in the list box are variable in height. The owner window receives a WM_MEASUREITEM message for each item in the combo box when you create the combo box, and a WM_DRAWITEM message when a visual aspect of the combo box has changed. |
| CBS_SIMPLE | Displays the list box at all times. The current selection in the list box is displayed in the edit control. |
| CBS_SORT | Automatically sorts strings added to the list box. |
| CBS_UPPERCASE | Converts to uppercase all text in both the selection field and list. |

Edit Controls

The Win32 API provides dialog boxes and controls to support communication between the application and the user. An *edit control* is a rectangular control window typically used in a dialog box to permit the user to enter and edit text by typing on the keyboard.

Edit controls support both the Unicode character set in which characters are two bytes, and ANSI character sets in which characters are one byte. For more information about Unicode and ANSI character sets, see Chapter 12, *Unicode*, in the Base Services volume.

The Win32 API also provides support for rich-edit controls, which support many features not available in system edit controls. For more information, see *Rich-Edit Controls*, which

are described in an overview later in this chapter, and in detail on the DVD that accompanies the *Microsoft Win32 Developer's Reference Library*.

About Edit Controls

An edit control is selected and receives the input focus when a user either clicks the mouse inside it or presses the TAB key. After it is selected, the edit control displays its text (if any) and a flashing caret that indicates the insertion point. The user can then enter text, move the insertion point, or select text to be edited by using the keyboard or the mouse. An edit control can send notification messages to its parent window in the form of **WM_COMMAND** messages. For more information about messages from an edit control, see *Edit Control Notification Messages*. A parent window can send messages to an edit control in a dialog box by calling the **SendDlgItemMessage** function. Each of the messages sent to edit controls is discussed in this overview.

The system provides both single-line edit controls (sometimes called SLEs) and multiline edit controls (sometimes called MLEs). Edit controls belong to the EDIT window class.

A combo box is a control that combines much of the functionality of an edit control and a list box. In a combo box, the edit control displays the current selection, and the list box presents options a user can select.

Many developers use the dialog boxes provided in the common dialog-box library (Comdlg32.dll) to perform tasks that otherwise might require customized edit controls.

Getting Information About Edit Control Programming Elements

For detailed information on Edit Control reference, please refer to the *Microsoft Win32 Developer's Reference Library* companion DVD bundled in the back of the Base Services volume.

Rich-Edit Controls

Rich-edit controls provide a programming interface for formatting text. However, an application must implement any UI components necessary to make formatting operations available to the user. A rich-edit control is a window in which the user can enter, edit, format, print, and save text. The text can be assigned character and paragraph formatting, and can include embedded COM objects.

Rich-edit controls support almost all of the messages and notification messages used with multiline edit controls. Thus, applications that already use edit controls can be easily changed to use rich-edit controls. Additional messages and notifications enable applications to access the functionality unique to rich-edit controls. Beginning with Rich Edit 2.0, there are also single-line or multiline, capabilities and plain or rich text. For information about edit controls, see *Edit Controls*.

About Rich-Edit Controls

The original specification for rich-edit controls is Rich Edit 1.0; the current specification is Rich Edit 3.0. Each version of rich edit is a superset of the preceding one, except that only Asian versions of Rich Edit 1.0 have a vertical text option. Before creating a rich-edit control, you should call the **LoadLibrary** function to verify which version of Rich Edit is installed. The following table shows which DLL corresponds to which version of rich edit. Note that the name of the file did not change from version 2.0 to version 3.0. This allows version 2.0 to be upgraded to version 3.0 without breaking existing code:

| Rich Edit version | DLL |
|-------------------|--------------|
| 1.0 | Riched32.dll |
| 2.0 | Riched20.dll |
| 3.0 | Riched20.dll |

Windows NT/Windows2000

Microsoft Windows NT version 4.0 includes both Rich Edit 1.0 and 2.0. Windows 2000 includes Rich Edit 3.0 with a Rich Edit 1.0 emulator.

Windows 98

Windows 98 includes both Rich Edit 1.0 and 2.0.

Windows 95

Windows 95 includes only Rich Edit 1.0. However, Riched20.dll is compatible with Windows 95, and may be installed if an application that uses Rich Edit 2.0 has been installed.

Getting More Information About Rich-Edit Controls

The companion DVD that is bundled inside the Base Services volume of the *Microsoft Win32 Developer's Reference Library* has the complete set of reference information for Rich-Edit Controls, which incorporates the Text Object Model. Publishing constraints associated with volumes in the Windows Programming Reference Series—which are governed by the mission to provide concise, compact, and portable reference books—did not allow Rich-Edit Controls to be included in the printed version. (Rich-Edit Controls is approximately 200 pages—by itself!)

However, in order to provide you with the most complete and comprehensive guide to Win32 development, the *Microsoft Win32 Developer's Reference Library* includes all of its information in electronic form on the DVD. If you have not gone already, go through the installation process on the companion DVD, and the entire body of Rich-Edit Control programming information (and much, much more) will be a click away.

Scroll Bars

A window in a Win32-based application can display a data object, such as a document or bitmap, that is larger than the window's client area. When provided with a *scroll bar*, the user can scroll a data object in the client area to bring into view the portions of the object that extend beyond the borders of the window.

About Scroll Bars

Scroll bars should be included in any window for which the content of the client area extends beyond the window's borders. A scroll bar's orientation determines the direction in which scrolling occurs when the user operates the scroll bar. A horizontal scroll bar enables the user to scroll the content of a window to the left or right. A vertical scroll bar enables the user to scroll the content of a window up or down.

Scroll-Bar Reference

Scroll-Bar Functions

EnableScrollBar

The **EnableScrollBar** function enables or disables one or both scroll-bar arrows.

```
BOOL EnableScrollBar(
    HWND hWnd, // handle to window or scroll bar
    UINT wSBflags, // scroll-bar type
    UINT wArrows // scroll-bar arrow options
);
```

Parameters

hWnd

[in] Handle to a window or a scroll-bar control, depending on the value of the *wSBflags* parameter.

wSBflags

[in] Specifies the scroll-bar type. This parameter can be one of the following values:

| Value | Meaning |
|---------|--|
| SB_BOTH | Enables or disables the arrows on the horizontal and vertical scroll bars that are associated with the specified window. The <i>hWnd</i> parameter must be the handle to the window. |
| SB_CTL | Indicates that the scroll bar is a scroll-bar control. The <i>hWnd</i> must be the handle to the scroll-bar control. |

| Value | Meaning |
|---------|---|
| SB_HORZ | Enables or disables the arrows on the horizontal scroll bar that is associated with the specified window. The <i>hWnd</i> parameter must be the handle to the window. |
| SB_VERT | Enables or disables the arrows on the vertical scroll bar that is associated with the specified window. The <i>hWnd</i> parameter must be the handle to the window. |

wArrows

[in] Specifies whether the scroll-bar arrows are enabled or disabled, and indicates the arrows that are enabled or disabled. This parameter can be one of the following values:

| Value | Meaning |
|-------------------|---|
| ESB_DISABLE_BOTH | Disables both arrows on a scroll bar. |
| ESB_DISABLE_DOWN | Disables the down arrow on a vertical scroll bar. |
| ESB_DISABLE_LEFT | Disables the left arrow on a horizontal scroll bar. |
| ESB_DISABLE_LTUP | Disables the left arrow on a horizontal scroll bar or the up arrow on a vertical scroll bar. |
| ESB_DISABLE_RIGHT | Disables the right arrow on a horizontal scroll bar. |
| ESB_DISABLE_RTDN | Disables the right arrow on a horizontal scroll bar or the down arrow on a vertical scroll bar. |
| ESB_DISABLE_UP | Disables the up arrow on a vertical scroll bar. |
| ESB_ENABLE_BOTH | Enables both arrows on a scroll bar. |

Return Values

If the arrows are enabled or disabled as specified, the return value is nonzero.

If the arrows are already in the requested state, or if an error occurs, 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: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Scroll Bars Overview, Scroll-Bar Functions, **ShowScrollBar**

GetScrollBarInfo

The **GetScrollBarInfo** function retrieves information about the specified scroll bar.

```

BOOL GetScrollBarInfo(
    HWND hwnd,           // handle to window
    LONG idObject,      // scroll-bar object
    SCROLLBARINFO psbi // scroll-bar information
);

```

Parameters

hwnd

[in] Handle to a window associated with the scroll bar whose information is to be retrieved. If the *idObject* parameter is OBJID_CLIENT, *hwnd* is a handle to a scroll-bar control. Otherwise, *hwnd* is a handle to a window created with the WS_VSCROLL style and/or the WS_HSCROLL style.

idObject

[in] Specifies the scroll-bar object. This parameter can be one of the following values:

| Value | Meaning |
|---------------|--|
| OBJID_CLIENT | The <i>hwnd</i> parameter is a handle to a scroll-bar control. |
| OBJID_HSCROLL | The horizontal scroll bar of the <i>hwnd</i> window. |
| OBJID_VSCROLL | The vertical scroll bar of the <i>hwnd</i> window. |

psbi

[out] Pointer to a **SCROLLBARINFO** structure to receive the information. Before calling **GetScrollBarInfo**, set the **cbSize** member to **sizeof(SCROLLBARINFO)**.

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**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Scroll Bars Overview, Scroll-Bar Functions, **SCROLLBARINFO**

GetScrollInfo

The **GetScrollInfo** function retrieves the parameters of a scroll bar, including the minimum and maximum scrolling positions, page size, and position of the scroll box (thumb).

```

BOOL GetScrollInfo(
    HWND hwnd,           // handle to window
    int fnBar,          // scroll-bar type
    LPSCROLLINFO lpsi // scroll-bar parameters
);

```

Parameters

hwnd

[in] Handle to a scroll-bar control or a window with a standard scroll bar, depending on the value of the *fnBar* parameter.

fnBar

[in] Specifies the type of scroll bar for which to retrieve parameters. This parameter can be one of the following values:

| Value | Meaning |
|---------|--|
| SB_CTL | Retrieves the parameters for a scroll-bar control. The <i>hwnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Retrieves the parameters for the window's standard horizontal scroll bar. |
| SB_VERT | Retrieves the parameters for the window's standard vertical scroll bar. |

lpsi

[in/out] Pointer to a **SCROLLINFO** structure. Before calling **GetScrollInfo**, set the **cbSize** member of the structure to **sizeof(SCROLLINFO)**, and set the **fMask** member to specify the scroll-bar parameters to retrieve. Before returning, the function copies the specified parameters to the appropriate members of the structure.

The **fMask** member can be one or more of the following values:

| Value | Meaning |
|--------------|---|
| SIF_PAGE | Copies the scroll page to the nPage member of the SCROLLINFO structure pointed to by <i>lpsi</i> . |
| SIF_POS | Copies the scroll position to the nPos member of the SCROLLINFO structure pointed to by <i>lpsi</i> . |
| SIF_RANGE | Copies the scroll range to the nMin and nMax members of the SCROLLINFO structure pointed to by <i>lpsi</i> . |
| SIF_TRACKPOS | Copies the current scroll-box tracking position to the nTrackPos member of the SCROLLINFO structure pointed to by <i>lpsi</i> . |

Return Values

If the function retrieved any values, the return value is nonzero.

If the function does not retrieve any values, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetScrollInfo** function enables applications to use 32-bit scroll positions. Although the messages that indicate scroll-bar position, **WM_HSCROLL** and **WM_VSCROLL**, provide only 16 bits of position data, the functions **SetScrollInfo** and **GetScrollInfo** provide 32 bits of scroll-bar position data. Thus, an application can call **GetScrollInfo** while processing either the **WM_HSCROLL** or **WM_VSCROLL** message to obtain 32-bit scroll-bar position data.

To get the 32-bit position of the scroll box (thumb) during a **SB_THUMBTRACK** notification in a **WM_HSCROLL** or **WM_VSCROLL** message, call **GetScrollInfo** with the **SIF_TRACKPOS** value in the **fMask** member of the **SCROLLINFO** structure. The function returns the tracking position of the scroll box in the **nTrackPos** member of the **SCROLLINFO** structure. This allows you to get the position of the scroll box as the user moves it. The following sample code illustrates the technique:

```
SCROLLINFO si;
case WM_HSCROLL:
    switch(LOWORD(wparam)) {
        case SB_THUMBTRACK:
            // Initialize SCROLLINFO structure

            ZeroMemory(&si, sizeof(SCROLLINFO));
            si.cbSize = sizeof(SCROLLINFO);
            si.fMask = SIF_TRACKPOS;

            // Call GetScrollInfo to get current tracking
            // position in si.nTrackPos

            if (!GetScrollInfo(hwnd, SB_HORZ, &si) )
                return 1; // GetScrollInfo failed
            break;
    }
}
```



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

Scroll Bars Overview, Scroll-Bar Functions, **SCROLLINFO**, **SetScrollInfo**, **WM_HSCROLL**, **WM_VSCROLL**

GetScrollPos

The **GetScrollPos** function retrieves the current position of the scroll box (thumb) in the specified scroll bar. The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 0 through 100, and the scroll box is in the middle of the bar, the current position is 50.

Note The **GetScrollPos** function is provided for backward compatibility. New applications should use the **GetScrollInfo** function.

```
int GetScrollPos(  
    HWND hWnd, // handle to window  
    int nBar // scroll-bar options  
);
```

Parameters

hWnd

[in] Handle to a scroll-bar control or a window with a standard scroll bar, depending on the value of the *nBar* parameter.

nBar

[in] Specifies the scroll bar to be examined. This parameter can be one of the following values:

| Value | Meaning |
|---------|---|
| SB_CTL | Retrieves the position of the scroll box in a scroll-bar control. The <i>hWnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Retrieves the position of the scroll box in a window's standard horizontal scroll bar. |
| SB_VERT | Retrieves the position of the scroll box in a window's standard vertical scroll bar. |

Return Values

If the function succeeds, the return value is the current position of the scroll box.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetScrollPos** function enables applications to use 32-bit scroll positions. Although the messages that indicate scroll-bar position, **WM_HSCROLL** and **WM_VSCROLL**, are limited to 16 bits of position data, the functions **SetScrollPos**, **SetScrollRange**, **GetScrollPos**, and **GetScrollRange** support 32-bit scroll-bar position data. Thus, an application can call **GetScrollPos** while processing either the **WM_HSCROLL** or **WM_VSCROLL** message to obtain 32-bit scroll-bar position data.

To get the 32-bit position of the scroll box (thumb) during a **SB_THUMBTRACK** notification in a **WM_HSCROLL** or **WM_VSCROLL** message, use the **GetScrollInfo** 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Scroll Bars Overview, Scroll-Bar Functions, **GetScrollInfo**, **GetScrollRange**, **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **WM_HSCROLL**, **WM_VSCROLL**

GetScrollRange

The **GetScrollRange** function retrieves the current minimum and maximum scroll-box (thumb) positions for the specified scroll bar.

Note The **GetScrollRange** function is provided for compatibility only. New applications should use the **GetScrollInfo** function.

```

BOOL GetScrollRange(
    HWND hWnd,           // handle to window
    int nBar,            // scroll-bar options
    LPINT lpMinPos,      // receives minimum position
    LPINT lpMaxPos      // receives maximum position
);

```

Parameters

hWnd

[in] Handle to a scroll-bar control or a window with a standard scroll bar, depending on the value of the *nBar* parameter.

nBar

[in] Specifies the scroll bar from which the positions are retrieved. This parameter can be one of the following values:

| Value | Meaning |
|---------|--|
| SB_CTL | Retrieves the positions of a scroll-bar control. The <i>hWnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Retrieves the positions of the window's standard horizontal scroll bar. |
| SB_VERT | Retrieves the positions of the window's standard vertical scroll bar. |

lpMinPos

[out] Pointer to the integer variable that receives the minimum position.

lpMaxPos

[out] Pointer to the integer variable that receives the maximum position.

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

If the specified window does not have standard scroll bars, or is not a scroll-bar control, the **GetScrollRange** function copies zero to the *lpMinPos* and *lpMaxPos* parameters.

The default range for a standard scroll bar is 0 through 100. The default range for a scroll-bar control is empty (both values are zero).

The messages that indicate scroll-bar position, **WM_HSCROLL** and **WM_VSCROLL**, are limited to 16 bits of position data. However, because **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **GetScrollInfo**, **GetScrollPos**, and **GetScrollRange** support 32-bit scroll-bar position data, there is a way to circumvent the 16-bit barrier of the **WM_HSCROLL** and **WM_VSCROLL** messages. See **GetScrollInfo** for a description of the technique.

! 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

Scroll Bars Overview, Scroll-Bar Functions, **GetScrollInfo**, **GetScrollPos**, **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **WM_HSCROLL**, **WM_VSCROLL**

ScrollDC

The **ScrollDC** function scrolls a rectangle of bits horizontally and vertically.

```

BOOL ScrollDC(
    HDC hDC,           // handle to DC
    int dx,           // horizontal scrolling
    int dy,           // vertical scrolling
    CONST RECT *lprcScroll, // scrolling rectangle
    CONST RECT *lprcClip, // clipping rectangle
    HRGN hrgnUpdate, // handle to scrolling region
    LPRECT lprcUpdate // update rectangle
);

```

Parameters

hDC

[in] Handle to the device context that contains the bits to be scrolled.

dx

[in] Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

dy

[in] Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

lprcScroll

[in] Pointer to a **RECT** structure containing the coordinates of the bits to be scrolled. The only bits affected by the scroll operation are bits in the intersection of this rectangle and the rectangle specified by *lprcClip*. If *lprcScroll* is NULL, the entire client area is used.

lprcClip

[in] Pointer to a **RECT** structure containing the coordinates of the clipping rectangle. The only bits that will be painted are the bits that remain inside this rectangle after the scroll operation has been completed. If *lprcClip* is NULL, the entire client area is used.

hrgnUpdate

[in] Handle to the region uncovered by the scrolling process. **ScrollDC** defines this region; it is not necessarily a rectangle.

lprcUpdate

[out] Pointer to a **RECT** structure that receives the coordinates of the rectangle bounding the scrolling update region. This is the largest rectangular area that requires repainting. When the function returns, the values in the structure are in client

coordinates, regardless of the mapping mode for the specified device context. This allows applications to use the update region in a call to the **InvalidateRgn** function, if required.

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

If the *lprcUpdate* parameter is NULL, the system does not compute the update rectangle. If both the *hrgnUpdate* and *lprcUpdate* parameters are NULL, the system does not compute the update region. If *hrgnUpdate* is not NULL, the system proceeds as though it contains a valid handle to the region uncovered by the scrolling process (defined by **ScrollDC**).

When you must scroll the entire client area of a window, use the **ScrollWindowEx** 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

Scroll Bars Overview, Scroll-Bar Functions, **InvalidateRgn**, **RECT**, **ScrollWindowEx**

ScrollWindow

The **ScrollWindow** function scrolls the content of the specified window's client area.

Note The **ScrollWindow** function is provided for backward compatibility. New applications should use the **ScrollWindowEx** function.

```
BOOL ScrollWindow(  
    HWND hWnd,           // handle to window  
    int XAmount,         // horizontal scrolling  
    int YAmount,         // vertical scrolling  
    CONST RECT *lpRect, // client area  
    CONST RECT *lpClipRect // clipping rectangle  
);
```


Parameters

hWnd

[in] Handle to the window where the client area is to be scrolled.

XAmount

[in] Specifies the amount, in device units, of horizontal scrolling. If the window being scrolled has the CS_OWNDC or CS_CLASSDC style, then this parameter uses logical units rather than device units. This parameter must be a negative value to scroll the content of the window to the left.

YAmount

[in] Specifies the amount, in device units, of vertical scrolling. If the window being scrolled has the CS_OWNDC or CS_CLASSDC style, then this parameter uses logical units rather than device units. This parameter must be a negative value to scroll the content of the window up.

lpRect

[in] Pointer to the **RECT** structure specifying the portion of the client area to be scrolled. If this parameter is NULL, the entire client area is scrolled.

lpClipRect

[in] Pointer to the **RECT** structure containing the coordinates of the clipping rectangle. Only device bits within the clipping rectangle are affected. Bits scrolled from the outside of the rectangle to the inside are painted; bits scrolled from the inside of the rectangle to the outside are not painted.

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

If the caret is in the window being scrolled, **ScrollWindow** automatically hides the caret to prevent it from being erased and, then, restores the caret after the scrolling is finished. The caret position is adjusted accordingly.

The area uncovered by **ScrollWindow** is not repainted, but it is combined into the window's update region. The application eventually receives a **WM_PAINT** message notifying it that the region must be repainted. To repaint the uncovered area at the same time the scrolling is in action, call the **UpdateWindow** function immediately after calling **ScrollWindow**.

If the *lpRect* parameter is NULL, the positions of any child windows in the window are offset by the amount specified by the *XAmount* and *YAmount* parameters; invalid (unpainted) areas in the window are also offset. **ScrollWindow** is faster when *lpRect* is NULL.

If *lpRect* is not NULL, the positions of child windows are not changed and invalid areas in the window are not offset. To prevent updating problems when *lpRect* is not NULL, call **UpdateWindow** to repaint the window before calling **ScrollWindow**.

! 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

Scroll Bars Overview, Scroll-Bar Functions, **RECT**, **ScrollIDC**, **ScrollWindowEx**, **UpdateWindow**

ScrollWindowEx

The **ScrollWindowEx** function scrolls the content of the specified window's client area.

```
int ScrollWindowEx(  
    HWND hWnd,           // handle to window  
    int dx,              // horizontal scrolling  
    int dy,              // vertical scrolling  
    CONST RECT *prcScroll, // client area  
    CONST RECT *prcClip,  // clipping rectangle  
    HRGN hrgnUpdate,     // handle to update region  
    LPRECT prcUpdate,    // invalidated region  
    UINT flags           // scrolling options  
);
```

Parameters

hWnd

[in] Handle to the window where the client area is to be scrolled.

dx

[in] Specifies the amount, in device units, of horizontal scrolling. This parameter must be a negative value to scroll to the left.

dy

[in] Specifies the amount, in device units, of vertical scrolling. This parameter must be a negative value to scroll up.

prcScroll

[in] Pointer to a **RECT** structure that specifies the portion of the client area to be scrolled. If this parameter is NULL, the entire client area is scrolled.

prcClip

[in] Pointer to a **RECT** structure that contains the coordinates of the clipping rectangle. Only device bits within the clipping rectangle are affected. Bits scrolled from the outside of the rectangle to the inside are painted; bits scrolled from the inside of the rectangle to the outside are not painted. This parameter may be NULL.

hrgnUpdate

[in] Handle to the region that is modified to hold the region invalidated by scrolling. This parameter may be NULL.

prcUpdate

[out] Pointer to a **RECT** structure that receives the boundaries of the rectangle invalidated by scrolling. This parameter may be NULL.

flags

[in] Specifies flags that control scrolling. This parameter can be one of the following values:

| Value | Meaning |
|-------------------|--|
| SW_ERASE | Erases the newly invalidated region by sending a WM_ERASEBKGD message to the window when specified with the SW_INVALIDATE flag. |
| SW_INVALIDATE | Invalidates the region identified by the <i>hrgnUpdate</i> parameter after scrolling. |
| SW_SCROLLCHILDREN | Scrolls all child windows that intersect the rectangle pointed to by the <i>prcScroll</i> parameter. The child windows are scrolled by the number of pixels specified by the <i>dx</i> and <i>dy</i> parameters. The system sends a WM_MOVE message to all child windows that intersect the <i>prcScroll</i> rectangle, even if they do not move. |
| SW_SMOOTHSCROLL | Windows 98, Windows 2000: Scrolls using smooth scrolling. Use the HIWORD portion of the <i>flags</i> parameter to indicate how much time the smooth-scrolling operation should take. |

Return Values

If the function succeeds, the return value is SIMPLEREGION (rectangular invalidated region), COMPLEXREGION (nonrectangular invalidated region; overlapping rectangles), or NULLREGION (no invalidated region).

If the function fails, the return value is ERROR. To get extended error information, call **GetLastError**.

Remarks

If the SW_INVALIDATE and SW_ERASE flags are not specified, **ScrollWindowEx** does not invalidate the area that is scrolled from. If either of these flags is set,

ScrollWindowEx invalidates this area. The area is not updated until the application calls the **UpdateWindow** function, calls the **RedrawWindow** function (specifying the `RDW_UPDATENOW` or `RDW_ERASENOW` flag), or retrieves the **WM_PAINT** message from the application queue.

If the window has the `WS_CLIPCHILDREN` style, the returned areas specified by *hrgnUpdate* and *prcUpdate* represent the total area of the scrolled window that must be updated, including any areas in child windows that need updating.

If the `SW_SCROLLCHILDREN` flag is specified, the system does not properly update the screen if part of a child window is scrolled. The part of the scrolled child window that lies outside the source rectangle is not erased and is not properly redrawn in its new destination. To move child windows that do not lie completely within the rectangle specified by *prcScroll*, use the **DeferWindowPos** function. The cursor is repositioned if the `SW_SCROLLCHILDREN` flag is set and the caret rectangle intersects the scroll rectangle.

All input and output coordinates (for *prcScroll*, *prcClip*, *prcUpdate*, and *hrgnUpdate*) are determined as client coordinates, regardless of whether the window has the `CS_OWNDC` or `CS_CLASSDC` class style. Use the **LPtoDP** and **DPToLP** functions to convert to and from logical coordinates, if necessary.

! 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

Scroll Bars Overview, Scroll-Bar Functions, **DeferWindowPos**, **DPToLP**, **LPtoDP**, **RECT**, **RedrawWindow**, **UpdateWindow**

SetScrollInfo

The **SetScrollInfo** function sets the parameters of a scroll bar, including the minimum and maximum scrolling positions, the page size, and the position of the scroll box (thumb). The function also redraws the scroll bar, if requested.

```
int SetScrollInfo(
    HWND hwnd,           // handle to window
    int fnBar,           // scroll bar type
    LPCSCROLLINFO lps1, // scroll parameters
    BOOL fRedraw         // redraw flag
);
```

Parameters

hwnd

[in] Handle to a scroll-bar control or a window with a standard scroll bar, depending on the value of the *fnBar* parameter.

fnBar

[in] Specifies the type of scroll bar for which to set parameters. This parameter can be one of the following values:

| Value | Meaning |
|---------|--|
| SB_CTL | Sets the parameters of a scroll-bar control. The <i>hwnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Sets the parameters of the window's standard horizontal scroll bar. |
| SB_VERT | Sets the parameters of the window's standard vertical scroll bar. |

lpsi

[in] Pointer to a **SCROLLINFO** structure. Before calling **SetScrollInfo**, set the **cbSize** member of the structure to **sizeof(SCROLLINFO)**, set the **fMask** member to indicate the parameters to set, and specify the new parameter values in the appropriate members.

The **fMask** member can be one or more of the following values:

| Value | Meaning |
|---------------------|--|
| SIF_DISABLENOSCROLL | Disables the scroll bar instead of removing it, if the scroll bar's new parameters make the scroll bar unnecessary. |
| SIF_PAGE | Sets the scroll page to the value specified in the nPage member of the SCROLLINFO structure pointed to by <i>lpsi</i> . |
| SIF_POS | Sets the scroll position to the value specified in the nPos member of the SCROLLINFO structure pointed to by <i>lpsi</i> . |
| SIF_RANGE | Sets the scroll range to the value specified in the nMin and nMax members of the SCROLLINFO structure pointed to by <i>lpsi</i> . |

fRedraw

[in] Specifies whether the scroll bar is redrawn to reflect the changes to the scroll bar. If this parameter is TRUE, the scroll bar is redrawn; otherwise, it is not redrawn.

Return Values

The return value is the current position of the scroll box.

Remarks

The **SetScrollInfo** function performs range checking on the values specified by the **nPage** and **nPos** members of the **SCROLLINFO** structure. The **nPage** member must specify a value from 0 to **nMax - nMin + 1**. The **nPos** member must specify a value between **nMin** and **nMax - max(nPage-1, 0)**. If either value is beyond its range, the function sets it to a value that is just within the range.

! 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

Scroll Bars Overview, Scroll-Bar Functions, **GetScrollInfo**, **SCROLLINFO**

SetScrollPos

The **SetScrollPos** function sets the position of the scroll box (thumb) in the specified scroll bar and, if requested, redraws the scroll bar to reflect the new position of the scroll box.

Note The **SetScrollPos** function is provided for backward compatibility. New applications should use the **SetScrollInfo** function.

```
int SetScrollPos(  
    HWND hwnd,        // handle to window  
    int nBar,         // scroll bar  
    int nPos,         // new position of scroll box  
    BOOL bRedraw      // redraw flag  
);
```

Parameters

hwnd

[in] Handle to a scroll-bar control or window with a standard scroll bar, depending on the value of the *nBar* parameter.

nBar

[in] Specifies the scroll bar to be set. This parameter can be one of the following values:

| Value | Meaning |
|---------|--|
| SB_CTL | Sets the position of the scroll box in a scroll-bar control. The <i>hWnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Sets the position of the scroll box in a window's standard horizontal scroll bar. |
| SB_VERT | Sets the position of the scroll box in a window's standard vertical scroll bar. |

nPos

[in] Specifies the new position of the scroll box. The position must be within the scrolling range. For more information about the scrolling range, see the **SetScrollRange** function.

bRedraw

[in] Specifies whether the scroll bar is redrawn to reflect the new scroll-box position. If this parameter is TRUE, the scroll bar is redrawn. If it is FALSE, the scroll bar is not redrawn.

Return Values

If the function succeeds, the return value is the previous position of the scroll box.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the scroll bar is redrawn by a subsequent call to another function, setting the *bRedraw* parameter to FALSE is useful.

Because the messages that indicate scroll-bar position, **WM_HSCROLL** and **WM_VSCROLL**, are limited to 16 bits of position data, applications that rely solely on those messages for position data have a practical maximum value of 65,535 for the **SetScrollPos** function's *nPos* parameter.

However, because the **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **GetScrollInfo**, **GetScrollPos**, and **GetScrollRange** functions support 32-bit scroll-bar position data, there is a way to circumvent the 16-bit barrier of the **WM_HSCROLL** and **WM_VSCROLL** messages. See **GetScrollInfo** for a description of the technique.

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.

Library: Use user32.lib.

See Also

Scroll Bars Overview, Scroll-Bar Functions, **GetScrollInfo**, **GetScrollPos**, **GetScrollRange**, **SetScrollInfo**, **SetScrollRange**

SetScrollRange

The **SetScrollRange** function sets the minimum and maximum position values for the specified scroll bar.

Note The **SetScrollRange** function is provided for backward compatibility. New applications should use the **SetScrollInfo** function.

```

BOOL SetScrollRange(
    HWND hWnd, // handle to window
    int nBar, // scroll bar
    int nMinPos, // minimum scrolling position
    int nMaxPos, // maximum scrolling position
    BOOL bRedraw // redraw flag
);

```

Parameters

hWnd

[in] Handle to a scroll-bar control or a window with a standard scroll bar, depending on the value of the *nBar* parameter.

nBar

[in] Specifies the scroll bar to be set. This parameter can be one of the following values:

| Value | Meaning |
|---------|---|
| SB_CTL | Sets the range of a scroll-bar control. The <i>hWnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Sets the range of a window's standard horizontal scroll bar. |
| SB_VERT | Sets the range of a window's standard vertical scroll bar. |

nMinPos

[in] Specifies the minimum scrolling position.

nMaxPos

[in] Specifies the maximum scrolling position.

bRedraw

[in] Specifies whether the scroll bar should be redrawn to reflect the change. If this parameter is TRUE, the scroll bar is redrawn; if it is FALSE, the scroll bar is not redrawn.

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

You can use **SetScrollRange** to hide the scroll bar by setting *nMinPos* and *nMaxPos* to the same value. An application should not call the **SetScrollRange** function to hide a scroll bar while processing a scroll-bar message. New applications should use the **ShowScrollBar** function to hide the scroll bar.

If the call to **SetScrollRange** immediately follows a call to the **SetScrollPos** function, the *bRedraw* parameter in **SetScrollPos** must be zero to prevent the scroll bar from being drawn twice.

The default range for a standard scroll bar is 0 through 100. The default range for a scroll-bar control is empty (both the *nMinPos* and *nMaxPos* parameter values are zero). The difference between the values specified by the *nMinPos* and *nMaxPos* parameters must not be greater than the value of MAXLONG.

Because the messages that indicate scroll bar position, **WM_HSCROLL** and **WM_VSCROLL**, are limited to 16 bits of position data, applications that rely solely on those messages for position data have a practical maximum value of 65,535 for the **SetScrollRange** function's *nMaxPos* parameter.

However, because the **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **GetScrollInfo**, **GetScrollPos**, and **GetScrollRange** functions support 32-bit scroll-bar position data, there is a way to circumvent the 16-bit barrier of the **WM_HSCROLL** and **WM_VSCROLL** messages. See **GetScrollInfo** for a description of the technique.

! 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.

Library: Use user32.lib.

+ See Also

Scroll Bars Overview, Scroll-Bar Functions, **GetScrollInfo**, **GetScrollPos**, **GetScrollRange**, **SetScrollInfo**, **SetScrollPos**, **ShowScrollBar**

ShowScrollBar

The **ShowScrollBar** function shows or hides the specified scroll bar.

```

BOOL ShowScrollBar(
    HWND hWnd, // handle to window
    int wBar,  // scroll bar
    BOOL bShow // scroll-bar visibility
);

```

Parameters

hWnd

[in] Handle to a scroll-bar control or a window with a standard scroll bar, depending on the value of the *wBar* parameter.

wBar

[in] Specifies the scroll bar(s) to be shown or hidden. This parameter can be one of the following values:

| Value | Meaning |
|---------|--|
| SB_BOTH | Shows or hides a window's standard horizontal and vertical scroll bars. |
| SB_CTL | Shows or hides a scroll-bar control. The <i>hWnd</i> parameter must be the handle to the scroll-bar control. |
| SB_HORZ | Shows or hides a window's standard horizontal scroll bars. |
| SB_VERT | Shows or hides a window's standard vertical scroll bar. |

bShow

[in] Specifies whether the scroll bar is shown or hidden. If this parameter is TRUE, the scroll bar is shown; otherwise, it is hidden.

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

You should not call this function to hide a scroll bar while processing a scroll bar 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

Scroll Bars Overview, Scroll-Bar Functions, **EnableScrollBar**

Scroll-Bar Structures

SCROLLBARINFO

The **SCROLLBARINFO** structure contains scroll-bar information.

```
typedef struct tagSCROLLBARINFO {
    DWORD cbSize;
    RECT rcScrollBar;
    int dxyLineButton;
    int xyThumbTop;
    int xyThumbBottom;
    int reserved;
    DWORD rgstate[CCHILDREN_SCROLLBAR+1];
} SCROLLBARINFO, *PSCROLLBARINFO, *LPSCROLLBARINFO;
```

Members

cbSize

Specifies the size, in bytes, of the structure. Before calling the **GetScrollBarInfo** function, set **cbSize** to **sizeof(SCROLLBARINFO)**.

rcScrollBar

Pointer to a **RECT** structure that indicates the coordinates of the scroll bar.

dxyLineButton

Height or width of the thumb.

xyThumbTop

Position of the top or left of the thumb.

xyThumbBottom

Position of the bottom or right of the thumb.

reserved

Reserved.

rgstate

An array of **DWORD** elements. Each element indicates the state of a scroll-bar component. The following table shows the scroll-bar component that corresponds to each array index:

| Index | Scroll-bar component |
|-------|-------------------------------|
| 0 | The scroll bar itself |
| 1 | The top or right arrow button |

| Index | Scroll-bar component |
|-------|-----------------------------------|
| 2 | The page-up or page-right region |
| 3 | The scroll box (thumb) |
| 4 | The page-down or page-left region |
| 5 | The bottom or left arrow button |

The **DWORD** element for each scroll-bar component can include a combination of the following bit flags:

| Value | Meaning |
|--------------------------|--|
| STATE_SYSTEM_INVISIBLE | For the scroll bar itself, indicates that the specified vertical or horizontal scroll bar does not exist. For the page-up or page-down regions, indicates the thumb is positioned so that the region does not exist. |
| STATE_SYSTEM_OFFSCREEN | For the scroll bar itself, indicates that the window is sized so that the specified vertical or horizontal scroll bar is not displayed currently. |
| STATE_SYSTEM_PRESSED | The arrow button or page region is pressed. |
| STATE_SYSTEM_UNAVAILABLE | The component is disabled. |

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Scroll Bars Overview, Scroll-Bar Structures, **GetScrollBarInfo**, **RECT**

SCROLLINFO

The **SCROLLINFO** structure contains scroll-bar parameters to be set by the **SetScrollInfo** function (or **SBM_SETSCROLLINFO** message), or retrieved by the **GetScrollInfo** function (or **SBM_GETSCROLLINFO** message).

```
typedef struct tagSCROLLINFO {
    UINT cbSize;
    UINT fMask;
    int nMin;
    int nMax;
    UINT nPage;
    int nPos;
```

```

    int nTrackPos;
} SCROLLINFO, *LPSCROLLINFO;
typedef SCROLLINFO CONST *LPCSCROLLINFO;

```

Members

cbSize

Specifies the size, in bytes, of this structure.

fMask

Specifies the scroll-bar parameters to set or retrieve. This member can be a combination of the following values:

| Value | Meaning |
|---------------------|---|
| SIF_ALL | Combination of SIF_PAGE, SIF_POS, SIF_RANGE, and SIF_TRACKPOS. |
| SIF_DISABLENOSCROLL | This value is used only when setting a scroll bar's parameters. If the scroll bar's new parameters make the scroll bar unnecessary, disable the scroll bar, instead of removing it. |
| SIF_PAGE | The nPage member contains the page size for a proportional scroll bar. |
| SIF_POS | The nPos member contains the scroll-box position, which is not updated while the user drags the scroll box. |
| SIF_RANGE | The nMin and nMax members contain the minimum and maximum values for the scrolling range. |
| SIF_TRACKPOS | The nTrackPos member contains the current position of the scroll box while the user is dragging it. |

nMin

Specifies the minimum scrolling position.

nMax

Specifies the maximum scrolling position.

nPage

Specifies the page size. A scroll bar uses this value to determine the appropriate size of the proportional scroll box.

nPos

Specifies the position of the scroll box.

nTrackPos

Specifies the immediate position of a scroll box that the user is dragging. An application can retrieve this value while processing the SB_THUMBTRACK notification message. An application cannot set the immediate scroll position; the **SetScrollInfo** function ignores this member.

! 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.

+ See Also

Scroll Bars Overview, Scroll-Bar Structures, **GetScrollInfo**, **SBM_GETSCROLLINFO**, **SBM_SETSCROLLINFO**, **SetScrollInfo**

Scroll-Bar Messages

SBM_ENABLE_ARROWS

An application sends the **SBM_ENABLE_ARROWS** message to enable or disable one or both arrows of a scroll-bar control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    SBM_ENABLE_ARROWS,    // message to send
    (WPARAM) wParam,     // scroll-bar arrow options
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies whether the scroll-bar arrows are enabled or disabled, and indicates which arrows are enabled or disabled. This parameter can be one of the following values:

| Value | Meaning |
|------------------|---|
| ESB_DISABLE_BOTH | Disables both arrows on a scroll bar. |
| ESB_DISABLE_DOWN | Disables the down arrow on a vertical scroll bar. |
| ESB_DISABLE_LEFT | Disables the left arrow on a horizontal scroll bar. |
| ESB_DISABLE_LTUP | Disables the left arrow on a horizontal scroll bar or the up arrow on a vertical scroll bar. |
| ESB_DISABLE_RTDN | Disables the right arrow on a horizontal scroll bar or the down arrow on a vertical scroll bar. |
| ESB_DISABLE_UP | Disables the up arrow on a vertical scroll bar. |
| ESB_ENABLE_BOTH | Enables both arrows on a scroll bar. |

IParam

This parameter is not used.

Return Values

If the message succeeds, the return value is TRUE; otherwise, it is 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 winuser.h; include windows.h.

+ See Also

Scroll Bars Overview, Scroll-Bar Messages

SBM_GETPOS

An application sends the **SBM_GETPOS** message to retrieve the current position of the scroll box of a scroll-bar control. The current position is a relative value that depends on the current scrolling range. For example, if the scrolling range is 0 through 100, and the scroll box is in the middle of the bar, the current position is 50.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    SBM_GETPOS,           // message to send
    (LPARAM) wParam,      // not used; must be zero
    (LPARAM) lParam:      // not used; must be zero
);
```

Parameters

This message has no parameters.

Return Values

The return value is the current position of the scroll box in the scroll bar.

! 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

Scroll Bars Overview, Scroll-Bar Messages, **SBM_GETRANGE**, **SBM_SETPOS**, **SBM_SETRANGE**, **SBM_SETRANGEREDRAW**

SBM_GETRANGE

An application sends the **SBM_GETRANGE** message to a scroll-bar control to retrieve the minimum and maximum position values for the control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    SBM_GETRANGE,         // message to send  
    (WPARAM) wParam;      // minimum position (LPINT)  
    (LPARAM) lParam;      // maximum position (LPINT)  
);
```

Parameters

wParam

Pointer to a variable that receives the minimum scrolling position.

lParam

Pointer to a variable that receives the maximum scrolling position.

Return Values

This message does not return a 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 winuser.h; include windows.h.

+ See Also

Scroll Bars Overview, Scroll-Bar Messages, **SBM_GETPOS**, **SBM_SETPOS**, **SBM_SETRANGE**, **SBM_SETRANGEREDRAW**

SBM_GETSCROLLINFO

An application sends the **SBM_GETSCROLLINFO** message to retrieve the parameters of a scroll bar.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    SBM_GETSCROLLINFO,    // message to send
    (WPARAM) wParam;     // not used; must be zero
    (LPARAM) lParam;     // parameters (LPSCROLLINFO)
);
```

Parameters

wParam

This parameter is not used.

lParam

Pointer to a **SCROLLINFO** structure. Before sending the message, set the **cbSize** member of the structure to **sizeof(SCROLLINFO)**, and set the **fMask** member to specify the scroll-bar parameters to retrieve. Before returning, the message copies the specified parameters to the appropriate members of the structure.

The **fMask** member can be one or more of the following values:

| Value | Meaning |
|--------------|---|
| SIF_ALL | Combination of SIF_PAGE, SIF_POS, SIF_RANGE, and SIF_TRACKPOS. |
| SIF_PAGE | Copies the scroll page to the nPage member. |
| SIF_POS | Copies the scroll position to the nPos member. |
| SIF_RANGE | Copies the scroll range to the nMin and nMax members. |
| SIF_TRACKPOS | Copies the current scroll-box tracking position to the nTrackPos member. |

Return Values

If the message retrieved any values, the return value is TRUE; otherwise, it is FALSE.

Remarks

The messages that indicate scroll-bar position, **WM_HSCROLL** and **WM_VSCROLL**, provide only 16 bits of position data. However, the **SCROLLINFO** structure used by **SBM_GETSCROLLINFO**, **SBM_SETSCROLLINFO**, **GetScrollInfo**, and **SetScrollInfo** provides 32 bits of scroll-bar position data. You can use these messages and functions while processing either the **WM_HSCROLL** or **WM_VSCROLL** message to obtain 32-bit scroll-bar position data.

To get the 32-bit position of the scroll box (thumb) during a **SB_THUMBTRACK** notification in a **WM_HSCROLL** or **WM_VSCROLL** message, send **SBM_GETSCROLLINFO** with the **SIF_TRACKPOS** value in the **fMask** member of the **SCROLLINFO** structure. The message returns the tracking position of the scroll box in the **nTrackPos** member of the

SCROLLINFO structure. This allows you to get the position of the scroll box as the user moves it. Alternatively, you can use the **GetScrollInfo** function to get the same information.

! 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`.

+ See Also

Scroll Bars Overview, Scroll-Bar Messages, **GetScrollInfo**, **SBM_SETSCROLLINFO**, **SCROLLINFO**, **SetScrollInfo**

SBM_SETPOS

An application sends the **SBM_SETPOS** message to a scroll-bar control to set the position of the scroll box (thumb) and, if requested, redraw the scroll bar to reflect the new position of the scroll box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    SBM_SETPOS,           // message to send  
    (WPARAM) wParam,     // new position  
    (LPARAM) lParam;     // redraw option  
);
```

Parameters

wParam

Specifies the new position of the scroll box. It must be within the scrolling range.

lParam

Specifies whether the scroll bar should be redrawn to reflect the new scroll box position. If this parameter is `TRUE`, the scroll bar is redrawn; if it is `FALSE`, the scroll bar is not redrawn.

Return Values

If the position of the scroll box changed, the return value is the previous position of the scroll box; otherwise, it is zero.

Remarks

If the scroll-bar control is redrawn by a subsequent call to another function, setting the *lParam* parameter to `FALSE` is useful.

! 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

Scroll Bars Overview, Scroll-Bar Messages, **SBM_GETPOS**, **SBM_GETRANGE**, **SBM_SETRANGE**, **SBM_SETRANGEREDRAW**

SBM_SETRANGE

An application sends the **SBM_SETRANGE** message to a scroll-bar control to set the minimum and maximum position values for the control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    SBM_SETRANGE,         // message to send  
    (WPARAM) wParam,      // minimum scrolling position  
    (LPARAM) lParam;      // maximum scrolling position  
);
```

Parameters

wParam

Specifies the minimum scrolling position.

lParam

Specifies the maximum scrolling position.

Return Values

If the position of the scroll box changed, the return value is the previous position of the scroll box; otherwise, it is zero.

Remarks

The default minimum and maximum position values are zero. The difference between the values specified by the *wParam* and *lParam* parameters must not be greater than MAXLONG.

If the minimum and maximum position values are equal, the scroll-bar control is hidden and, in effect, disabled.

! 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

Scroll Bars Overview, Scroll-Bar Messages, **SBM_GETPOS**, **SBM_GETRANGE**, **SBM_SETPOS**, **SBM_SETRANGEREDRAW**

SBM_SETRANGEREDRAW

An application sends the **SBM_SETRANGEREDRAW** message to a scroll-bar control to set the minimum and maximum position values, and to redraw the control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    SBM_SETRANGEREDRAW,   // message to send  
    (WPARAM) wParam,     // minimum scrolling position  
    (LPARAM) lParam;     // maximum scrolling position  
);
```

Parameters

wParam

Specifies the minimum scrolling position.

lParam

Specifies the maximum scrolling position.

Return Values

If the position of the scroll box changed, the return value is the previous position of the scroll box; otherwise, it is zero.

Remarks

The default minimum and maximum position values are zero. The difference between the values specified by the *wParam* and *lParam* parameters must not be greater than MAXLONG.

If the minimum and maximum position values are equal, the scroll-bar control is hidden and, in effect, disabled.

! 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

Scroll Bars Overview, Scroll-Bar Messages, **SBM_GETPOS**, **SBM_GETRANGE**, **SBM_SETPOS**, **SBM_SETRANGE**

SBM_SETSCROLLINFO

An application sends the **SBM_GETSCROLLINFO** message to set the parameters of a scroll bar.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    SBM_SETSCROLLINFO,    // message to send
    (WPARAM) wParam,     // redraw option
    (LPARAM) lParam;     // parameters (LPSCROLLINFO)
);
```

Parameters

wParam

Specifies whether the scroll bar is redrawn to reflect the new scroll-box position. If this parameter is TRUE, the scroll bar is redrawn; if it is FALSE, the scroll bar is not redrawn.

lParam

Pointer to a **SCROLLINFO** structure. Before sending the message, set the **cbSize** member of the structure to **sizeof(SCROLLINFO)**, set the **fMask** member to indicate the parameters to set, and specify the new parameter values in the appropriate members.

The **fMask** member can be one or more of the following values.

| Value | Meaning |
|---------------------|---|
| SIF_DISABLENOSCROLL | Disables the scroll bar instead of removing it, if the scroll bar's new parameters make the scroll bar unnecessary. |
| SIF_PAGE | Sets the scroll page to the value specified in the nPage member. |

| Value | Meaning |
|-----------|--|
| SIF_POS | Sets the scroll position to the value specified in the nPos member. |
| SIF_RANGE | Sets the scroll range to the value specified in the nMin and nMax members. |

Return Values

The return value is the current position of the scroll box.

Remarks

The messages that indicate scroll-bar position, **WM_HSCROLL** and **WM_VSCROLL**, provide only 16 bits of position data. However, the **SCROLLINFO** structure used by **SBM_GETSCROLLINFO**, **SBM_SETSCROLLINFO**, **GetScrollInfo**, and **SetScrollInfo** provides 32 bits of scroll-bar position data. You can use these messages and functions while processing either the **WM_HSCROLL** or **WM_VSCROLL** message to obtain 32-bit scroll-bar position data.

! 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`.

+ See Also

Scroll Bars Overview, Scroll-Bar Messages, **GetScrollInfo**, **SBM_GETSCROLLINFO**, **SCROLLINFO**, **SetScrollInfo**

WM_CTLCOLORSCROLLBAR

The **WM_CTLCOLORSCROLLBAR** message is sent to the parent window of a scroll-bar control when the control is about to be drawn. By responding to this message, the parent window can use the display context handle to set the background color of the scroll-bar control.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_CTLCOLORSCROLLBAR
    WPARAM wParam,      // handle to DC (HDC)
    LPARAM lParam       // handle to scroll bar (HWND)
);

```

Parameters

wParam

Handle to the device context for the scroll-bar control.

lParam

Handle to the scroll bar.

Return Values

If an application processes this message, it must return the handle to a brush. The system uses the brush to paint the background of the scroll-bar control.

Remarks

By default, the **DefWindowProc** function selects the default system colors for the scroll-bar control.

The system does not automatically destroy the returned brush. It is the application's responsibility to destroy the brush when it is no longer needed.

The **WM_CTLCOLORSCROLLBAR** message is never sent between threads; it is only sent within the same thread.

If a dialog-box procedure handles this message, it should cast the desired return value to a **BOOL** and return the value directly. If the dialog-box procedure returns **FALSE**, default message handling is performed. The **DWL_MSGRESULT** value set by the **SetWindowLong** function is ignored.

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

Scroll Bars Overview, Scroll-Bar Messages, **DefWindowProc**, **RealizePalette**, **SelectPalette**, **WM_CTLCOLORBTN**, **WM_CTLCOLORDLG**, **WM_CTLCOLOREDIT**, **WM_CTLCOLORLISTBOX**, **WM_CTLCOLORMSGBOX**, **WM_CTLCOLORSTATIC**

WM_HSCROLL

The **WM_HSCROLL** message is sent to a window when a scroll event occurs in the window's standard horizontal scroll bar. This message is also sent to the owner of a horizontal scroll-bar control when a scroll event occurs in the control.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_HSCROLL
    WPARAM wParam,      // request and position
    LPARAM lParam       // handle to scroll bar (HWND)
);

```

Parameters

wParam

The low-order word specifies a scroll-bar value that indicates the user's scrolling request. This word can be one of the following values:

| Value | Meaning |
|------------------|---|
| SB_ENDSCROLL | Ends scroll. |
| SB_LEFT | Scrolls to the upper left. |
| SB_LINELEFT | Scrolls left by one unit. |
| SB_LINERIGHT | Scrolls right by one unit. |
| SB_PAGELEFT | Scrolls left by the width of the window. |
| SB_PAGERIGHT | Scrolls right by the width of the window. |
| SB_RIGHT | Scrolls to the lower right. |
| SB_THUMBPOSITION | The user has dragged the scroll box (thumb) and released the mouse button. The high-order word indicates the position of the scroll box at the end of the drag operation. |
| SB_THUMBTRACK | The user is dragging the scroll box. This message is sent repeatedly until the user releases the mouse button. The high-order word indicates the position to which the scroll box has been dragged. |

The high-order word specifies the current position of the scroll box if the low-order word is **SB_THUMBPOSITION** or **SB_THUMBTRACK**; otherwise, this word is not used.

lParam

If the message is sent by a scroll bar, this parameter is the handle to the scroll-bar control. If the message is not sent by a scroll bar, this parameter is **NULL**.

Return Values

If an application processes this message, it should return zero.

Remarks

The **SB_THUMBTRACK** notification message is used typically by applications that provide feedback as the user drags the scroll box.

If an application scrolls the content of the window, it must also reset the position of the scroll box by using the **SetScrollPos** function.

Note that the **WM_HSCROLL** message carries only 16 bits of scroll-box position data. Thus, applications that rely solely on **WM_HSCROLL** (and **WM_VSCROLL**) for scroll position data have a practical maximum position value of 65,535.

However, because the **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **GetScrollInfo**, **GetScrollPos**, and **GetScrollRange** functions support 32-bit scroll-bar position data, there is a way to circumvent the 16-bit barrier of the **WM_HSCROLL** and **WM_VSCROLL** messages. See **GetScrollInfo** for a description of the technique.

! 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

Scroll Bars Overview, Scroll-Bar Messages, **GetScrollInfo**, **GetScrollPos**, **GetScrollRange**, **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **WM_VSCROLL**

WM_VSCROLL

The **WM_VSCROLL** message is sent to a window when a scroll event occurs in the window's standard vertical scroll bar. This message is sent also to the owner of a vertical scroll-bar control when a scroll event occurs in the control.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_VSCROLL
    WPARAM wParam,      // request and position
    LPARAM lParam       // handle to scroll bar (HWND)
);
```

Parameters

wParam

The low-order word specifies a scroll-bar value that indicates the user's scrolling request. This parameter can be one of the following values:

| Value | Meaning |
|------------------|---|
| SB_BOTTOM | Scrolls to the lower right. |
| SB_ENDSCROLL | Ends scroll. |
| SB_LINEDOWN | Scrolls one line down. |
| SB_LINEUP | Scrolls one line up. |
| SB_PAGEDOWN | Scrolls one page down. |
| SB_PAGEUP | Scrolls one page up. |
| SB_THUMBPOSITION | The user has dragged the scroll box (thumb) and released the mouse button. The high-order word indicates the position of the scroll box at the end of the drag operation. |
| SB_THUMBTRACK | The user is dragging the scroll box. This message is sent repeatedly until the user releases the mouse button. The high-order word indicates the position to which the scroll box has been dragged. |
| SB_TOP | Scrolls to the upper left. |

The high-order word specifies the current position of the scroll box if the high-order word is `SB_THUMBPOSITION` or `SB_THUMBTRACK`; otherwise, this word is not used.

lParam

If the message is sent by a scroll bar, this parameter is the handle to the scroll-bar control. If the message is not sent by a scroll bar, this parameter is `NULL`.

Return Values

If an application processes this message, it should return zero.

Remarks

The `SB_THUMBTRACK` notification message is used typically by applications that provide feedback as the user drags the scroll box.

If an application scrolls the content of the window, it must reset also the position of the scroll box by using the `SetScrollPos` function.

Note that the `WM_VSCROLL` message carries only 16 bits of scroll-box position data. Thus, applications that rely solely on `WM_VSCROLL` (and `WM_HSCROLL`) for scroll position data have a practical maximum position value of 65,535.

However, because the `SetScrollInfo`, `SetScrollPos`, `SetScrollRange`, `GetScrollInfo`, `GetScrollPos`, and `GetScrollRange` functions support 32-bit scroll-bar position data, there is a way to circumvent the 16-bit barrier of the `WM_HSCROLL` and `WM_VSCROLL` messages. See `GetScrollInfo` for a description of the technique.

! 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

Scroll Bars Overview, Scroll-Bar Messages, **GetScrollInfo**, **GetScrollPos**, **GetScrollRange**, **SetScrollInfo**, **SetScrollPos**, **SetScrollRange**, **WM_HSCROLL**

Scroll-Bar Control Styles

To create a scroll-bar control using the **CreateWindow** or **CreateWindowEx** function, specify the SCROLLBAR class, appropriate window style constants, and a combination of the following scroll-bar control styles. Some of the styles create a scroll-bar control that uses a default width or height. However, you always must specify the x-coordinates and y-coordinates and the other dimensions of the scroll bar when you call **CreateWindow** or **CreateWindowEx**:

| Style | Meaning |
|-----------------|--|
| SBS_BOTTOMALIGN | Aligns the bottom edge of the scroll bar with the bottom edge of the rectangle defined by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . The scroll bar has the default height for system scroll bars. Use this style with the SBS_HORZ style. |
| SBS_HORZ | Designates a horizontal scroll bar. If neither the SBS_BOTTOMALIGN nor SBS_TOPALIGN style is specified, the scroll bar has the height, width, and position specified by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . |
| SBS_LEFTALIGN | Aligns the left edge of the scroll bar with the left edge of the rectangle defined by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . The scroll bar has the default width for system scroll bars. Use this style with the SBS_VERT style. |
| SBS_RIGHTALIGN | Aligns the right edge of the scroll bar with the right edge of the rectangle defined by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . The scroll bar has the default width for system scroll bars. Use this style with the SBS_VERT style. |
| SBS_SIZEBOX | Designates a size box. If you specify neither the SBS_SIZEBOXBOTTOMRIGHTALIGN nor the SBS_SIZEBOXTOPLEFTALIGN style, the size box has the height, width, and position specified by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . |

| Style | Meaning |
|---------------------------------|--|
| SBS_SIZEBOX BOTTOMRIGHTALIGN | Aligns the lower-right corner of the size box with the lower-right corner of the rectangle specified by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . The size box has the default size for system size boxes. Use this style with the SBS_SIZEBOX style. |
| SBS_SIZEBOX TOPLEFTALIGN | Aligns the upper left corner of the size box with the upper left corner of the rectangle specified by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . The size box has the default size for system size boxes. Use this style with the SBS_SIZEBOX style. |
| SBS_SIZEGRIP SBS_TOPALIGN | Same as SBS_SIZEBOX, but with a raised edge. Aligns the top edge of the scroll bar with the top edge of the rectangle defined by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . The scroll bar has the default height for system scroll bars. Use this style with the SBS_HORZ style. |
| SBS_VERT | Designates a vertical scroll bar. If you specify neither the SBS_RIGHTALIGN nor the SBS_LEFTALIGN style, the scroll bar has the height, width, and position specified by the <i>x</i> , <i>y</i> , <i>nWidth</i> , and <i>nHeight</i> parameters of CreateWindowEx . |

Static Controls

Dialog boxes and controls support communication between an application and the user. A *static control* is a control that enables an application to provide the user with informational text and graphics that, typically, require no response.

About Static Controls

Applications often use static controls to label other controls or to separate a group of controls. Although static controls are child windows, they cannot be selected. Therefore, they cannot receive the keyboard focus or have a keyboard interface. A static control that has the SS_NOTIFY style receives mouse input, notifying the parent window when the user clicks or double-clicks the control. Static controls belong to the STATIC window class.

Although static controls can be used in overlapped, pop-up, and child windows, they are designed for use in dialog boxes, where the system standardizes their behavior. By using static controls outside dialog boxes, a developer increases the risk that the application might behave in a nonstandard fashion. Typically, a developer uses either static controls in dialog boxes or the SS_OWNERDRAW style to create customized static controls.

Static-Control Types

There are four types of static controls:

- simple graphics
- text
- image
- owner-drawn

Each type has one or more styles.

Simple Graphics Static Control

A simple graphics static control displays a frame or a filled rectangle. A frame can be drawn in a number of styles, including black, gray, or white. In addition, a frame can be drawn with an etched style to give it a three-dimensional appearance. The frame styles include `SS_BLACKFRAME`, `SS_GRAYFRAME`, `SS_WHITEFRAME`, `SS_ETCHEDHORZ`, `SS_ETCHEDVERT`, and `SS_ETCHEDFRAME`.

A rectangle can be filled with color in one of three styles: black, gray, or white. These styles are defined by the constants `SS_BLACKRECT`, `SS_GRAYRECT`, and `SS_WHITERECT`.

Text Static Control

A text static control displays text in a rectangle in one of five styles:

- left-aligned without word wrap
- left-aligned with word wrap
- centered
- right-aligned
- simple

These styles are defined by the constants `SS_LEFTNOWORDWRAP`, `SS_LEFT`, `SS_CENTER`, `SS_RIGHT`, and `SS_SIMPLE`, respectively. The system rearranges the text in these controls in predefined ways, except for “simple” text, which is not rearranged.

An application can change the text in a text static control at any time by using either the **SetWindowText** function or the **WM_SETTEXT** message.

The system displays as much text as it can in the static control, and clips whatever does not fit. To calculate an appropriate size for the control, retrieve the font metrics for the text. For more information about fonts and font metrics, see *Fonts and Text*.

Image Static Control

An image static control can display bitmaps, icons (including animated icons), or enhanced metafiles. The type of graphic that a particular static control displays depends on the control’s style: `SS_BITMAP`, `SS_ICON`, or `SS_ENHMETAFIELD`. An application specifies the style when it creates the control and specifies a handle to the bitmap, icon,

or metafile for the control to display. After the control is created, an application can associate a different graphic with the control by sending it an **STM_SETIMAGE** message, specifying a handle to the new graphic object. An application can retrieve a handle to the graphic object currently associated with a static control by sending it an **STM_GETIMAGE** message. An application sends messages to a static control by using the **SendDlgItemMessage** function.

Owner-Drawn Static Control

By using the **SS_OWNERDRAW** style, an application can take responsibility for painting a static control. The parent window of an owner-drawn static control (its owner) receives a **WM_DRAWITEM** message whenever the static control needs to be painted. The message includes a pointer to a **DRAWITEMSTRUCT** structure that contains information that the owner window uses when drawing the control.

Static-Control Reference

Static-Control Messages

The following messages are used with static controls (**STM_GETICON** and **STM_SETICON** are used only with icons):

STM_GETICON
STM_GETIMAGE
STM_SETICON
STM_SETIMAGE

Static controls with the **SS_NOTIFY** style send the following notification messages to their parent window:

STN_CLICKED
STN_DBLCLK
STN_DISABLE
STN_ENABLE
WM_CTLCOLORSTATIC

STM_GETICON

An application sends the **STM_GETICON** message to retrieve a handle to the icon associated with a static control that has the **SS_ICON** style.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    STM_GETICON,          // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

The return value is a handle to the icon, or it is NULL if either the static control has no associated icon or an error occurred.

! 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

Static Controls Overview, Static-Control Messages, **STM_SETICON**

STM_GETIMAGE

An application sends an **STM_GETIMAGE** message to retrieve a handle to the image associated with a static control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    STM_GETIMAGE,         // message to send
    (WPARAM) wParam,     // image type
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies the type of image to retrieve. This parameter can be one of the following values:

IMAGE_BITMAP
 IMAGE_CURSOR
 IMAGE_ENHMETAFILE
 IMAGE_ICON

lParam

This parameter is not used.

Return Values

The return value is a handle to the image, if any; otherwise, it is NULL.

! 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.

+ See Also

Static Controls Overview, Static-Control Messages, **STM_SETIMAGE**

STM_SETICON

An application sends the **STM_SETICON** message to associate an icon with an icon control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    STM_SETICON,          // message to send  
    (LPARAM) wParam,      // handle to icon (HICON)  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

Handle to the icon to associate with the icon control.

lParam

This parameter is not used.

Return Values

The return value is a handle to the icon previously associated with the icon control, or zero if an error occurs.

! 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

Static Controls Overview, Static-Control Messages, **LoadIcon**, **STM_GETICON**

STM_SETIMAGE

An application sends an **STM_SETIMAGE** message to associate a new image (icon or bitmap) with a static control.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    STM_SETIMAGE,         // message to send  
    (WPARAM) wParam;      // image type  
    (LPARAM) lParam;      // handle to image (HANDLE)  
);
```

Parameters

wParam

Specifies the type of image to associate with the static control. This parameter can be one of the following values:

IMAGE_BITMAP
IMAGE_CURSOR
IMAGE_ENHMETAFILE
IMAGE_ICON

lParam

Handle to the image to associate with the static control.

Return Values

The return value is a handle to the image previously associated with the static control, if any; otherwise, it is NULL.

! 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.

+ See Also

Static Controls Overview, Static-Control Messages, **STM_GETIMAGE**

STN_CLICKED

The **STN_CLICKED** notification message is sent when the user clicks a static control that has the **SS_NOTIFY** style. The parent window of the control receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,      // control identifier, STN_CLICKED  
    LPARAM lParam       // handle to static control (HWND)  
);
```

Parameters

wParam

The low-order word is a static-control identifier.

The high-order word is the notification message.

lParam

Handle to the static control.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Static Controls Overview, Static-Control Messages, **HIWORD**, **LOWORD**, **STN_DBLCLK**, **WM_COMMAND**

STN_DBLCLK

The **STN_DBLCLK** notification message is sent when the user double-clicks a static control that has the **SS_NOTIFY** style. The parent window of the control receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,      // control identifier, STN_DBLCLK  
    LPARAM lParam       // handle to static control (HWND)  
);
```

Parameters

wParam

The low-order word is a static-control identifier.

The high-order word is the notification message.

lParam

Handle to the static control.

! 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

Static Controls Overview, Static-Control Messages, **HIWORD**, **LOWORD**, **STN_CLICKED**, **WM_COMMAND**

STN_DISABLE

The **STN_DISABLE** notification message is sent when a static control is disabled. The static control must have the **SS_NOTIFY** style to receive this notification message. The parent window of the control receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,          // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,      // control identifier, STN_DISABLE  
    LPARAM lParam       // handle to static control (HWND)  
);
```

Parameters

wParam

The low-order word is a static-control identifier.

The high-order word is the notification message.

lParam

Handle to the static control.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Static Controls Overview, Static-Control Messages, **HIWORD**, **LOWORD**, **STN_ENABLE**, **WM_COMMAND**

STN_ENABLE

The **STN_ENABLE** notification message is sent when a static control is enabled. The static control must have the **SS_NOTIFY** style to receive this notification message. The parent window of the control receives this notification message through the **WM_COMMAND** message.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_COMMAND  
    WPARAM wParam,      // control identifier, STN_ENABLE  
    LPARAM lParam       // handle to static control (HWND)  
);
```

Parameters

wParam

The low-order word is a static-control identifier.

The high-order word is the notification message.

lParam

Handle to the static control.

Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Static Controls Overview, Static-Control Messages, **HIWORD**, **LOWORD**, **STN_DISABLE**, **WM_COMMAND**

WM_CTLCOLORSTATIC

A static control, or an edit control that is read-only or disabled, sends the **WM_CTLCOLORSTATIC** message to its parent window when the control is about to be drawn. By responding to this message, the parent window can use the specified device context handle to set the text and background colors of the static control.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_CTLCOLORSTATIC  
    WPARAM wParam,      // handle to DC (HDC)  
    LPARAM lParam       // handle to static control (HWND)  
);
```

Parameters

wParam

Handle to the device context for the static-control window.

lParam

Handle to the static control.

Return Values

If an application processes this message, the return value is a handle to a brush that the system uses to paint the background of the static control.

Remarks

By default, the **DefWindowProc** function selects the default system colors for the static control.

Edit controls that are not read-only or disabled do not send the **WM_CTLCOLORSTATIC** message; instead, they send the **WM_CTLCOLOREDIT** message. However, for compatibility purposes, the system sends the **WM_CTLCOLOREDIT** message for read-only and disabled edit controls if the application was designed for Windows 3.1 or earlier.

The system does not destroy automatically the returned brush. It is the application's responsibility to destroy the brush when it is no longer needed.

The **WM_CTLCOLORSTATIC** message is never sent between threads; it is sent only within the same thread.

If a dialog-box procedure handles this message, it should cast the desired return value to a **BOOL** and return the value directly. If the dialog-box procedure returns **FALSE**, default message handling is performed. The **DWL_MSGRESULT** value set by the **SetWindowLong** function is ignored.

! 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

Static Controls Overview, Static-Control Messages, **DefWindowProc**, **RealizePalette**, **SelectPalette**, **WM_CTLCOLORBTN**, **WM_CTLCOLORDLG**, **WM_CTLCOLOREDIT**, **WM_CTLCOLORLISTBOX**, **WM_CTLCOLORMSGBOX**, **WM_CTLCOLORSCROLLBAR**

Static-Control Styles

To create a static control using the **CreateWindow** or **CreateWindowEx** function, specify the **STATIC** class, appropriate window style constants, and a combination of the following static-control styles:

| Style | Description |
|----------------|---|
| SS_BITMAP | Specifies a bitmap is to be displayed in the static control. The text is the name of a bitmap (not a file name) defined elsewhere in the resource file. The style ignores the <i>nWidth</i> and <i>nHeight</i> parameters; the control automatically sizes itself to accommodate the bitmap. |
| SS_BLACKFRAME | Specifies a box with a frame drawn in the same color as the window frames. This color is black in the default color scheme. |
| SS_BLACKRECT | Specifies a rectangle filled with the current window frame color. This color is black in the default color scheme. |
| SS_CENTER | Specifies a simple rectangle, and centers the text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next centered line. Words that are longer than the width of the control are truncated. |
| SS_CENTERIMAGE | Specifies that, if the bitmap or icon is smaller than the client area of the static control, the rest of the client area is filled with the color of the pixel in the top-left corner of the bitmap or icon. If the static control contains a single line of text, the text is centered vertically in the client area of the control. |

(continued)

(continued)

| Style | Description |
|---------------------------------------|---|
| SS_ENDELLIPSIS or SS_PATHHELLIPSIS | <p>Windows NT/2000: Replaces part of the string with ellipses (...), if necessary, so that the result fits in the specified rectangle.</p> <p>You can specify SS_END_ELLIPSIS to replace characters at the end of the string, or SS_PATHHELLIPSIS to replace characters in the middle of the string. If the string contains backslash (\) characters, SS_PATHHELLIPSIS preserves as much of the text after the last backslash as possible.</p> |
| SS_ENHMETAFIELD | <p>Specifies an enhanced metafile is to be displayed in the static control. The text is the name of a metafile. An enhanced metafile static control has a fixed size; the metafile is scaled to fit the static control's client area.</p> |
| SS_ETCHEDFRAME | <p>Draws the frame of the static control using the EDGE_ETCHED edge style. For more information, see the DrawEdge function.</p> |
| SS_ETCHEDHORZ | <p>Draws the top and bottom edges of the static control using the EDGE_ETCHED edge style. For more information, see the DrawEdge function.</p> |
| SS_ETCHEDVERT | <p>Draws the left and right edges of the static control using the EDGE_ETCHED edge style. For more information, see the DrawEdge function.</p> |
| SS_GRAYFRAME | <p>Specifies a box with a frame drawn with the same color as the screen background (desktop). This color is gray in the default color scheme.</p> |
| SS_GRAYRECT | <p>Specifies a rectangle filled with the current screen background color. This color is gray in the default color scheme.</p> |
| SS_ICON | <p>Specifies an icon is to be displayed in the dialog box. The text is the name of an icon (not a file name) defined elsewhere in the resource file. The icon can be an animated cursor. The style ignores the <i>nWidth</i> and <i>nHeight</i> parameters; the control automatically sizes itself to accommodate the icon.</p> |

| Style | Description |
|-------------------|--|
| SS_LEFT | Specifies a simple rectangle, and left-aligns the text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next left-aligned line. Words that are longer than the width of the control are truncated. |
| SS_LEFTNOWORDWRAP | Specifies a simple rectangle, and left-aligns the text in the rectangle. Tabs are expanded, but words are not wrapped. Text that extends past the end of a line is clipped. |
| SS_NOPREFIX | Prevents interpretation of ampersand (&) characters in the control's text as accelerator prefix characters; these are displayed with the ampersand removed and the next character in the string underlined. This static control style may be included with any of the defined static controls. You can combine SS_NOPREFIX with other styles. This can be useful when file names or other strings that may contain an ampersand (&) must be displayed in a static control in a dialog box. |
| SS_NOTIFY | Sends the parent window STN_CLICKED , STN_DBLCLK , STN_DISABLE , and STN_ENABLE notification messages when the user clicks or double-clicks the control. |
| SS_OWNERDRAW | Specifies that the owner of the static control is responsible for drawing the control. The owner window receives a WM_DRAWITEM message whenever the control needs to be drawn. |
| SS_REALSIZEIMAGE | Prevents a static icon or bitmap control (that is, static controls that have the SS_ICON or SS_BITMAP style) from being resized as it is loaded or drawn. If the icon or bitmap is larger than the destination area, the image is clipped. |
| SS_RIGHT | Specifies a simple rectangle, and right-aligns the text in the rectangle. The text is formatted before it is displayed. Words that extend past the end of a line are automatically wrapped to the beginning of the next right-aligned line. Words that are longer than the width of the control are truncated. |

(continued)

(continued)

| Style | Description |
|-----------------|---|
| SS_RIGHTJUST | Specifies that the lower right corner of a static control with the SS_BITMAP or SS_ICON style is to remain fixed when the control is resized. Only the top and left sides are adjusted to accommodate a new bitmap or icon. |
| SS_SIMPLE | Specifies a simple rectangle, and displays a single line of left-aligned text in the rectangle. The text line cannot be shortened or altered in any way. The control's parent window or dialog box must not process the WM_CTLCOLORSTATIC message. |
| SS_SUNKEN | Draws a half-sunken border around a static control. |
| SS_WHITEFRAME | Specifies a box with a frame drawn with the same color as the window background. This color is white in the default color scheme. |
| SS_WHITERECT | Specifies a rectangle filled with the current window background color. This color is white in the default color scheme. |
| SS_WORDELLIPSIS | Windows NT/2000: Truncates text that does not fit, and adds ellipses (...). |

CHAPTER 7

Resources

Resources

A *resource* is binary data that you can add to the executable file of a Win32-based application. A resource can be either standard or defined. The data in a *standard resource* describes an icon, cursor, menu, dialog box, bitmap, enhanced metafile, font, accelerator table, message-table entry, string-table entry, or version information. An *application-defined resource*, also called a *custom resource*, contains any data required by a specific application.

About Resources

This overview describes the functions that enable applications to find a resource in a module; load a resource into memory; add, delete or replace a resource in an executable file; and generate a list of the resources contained in a module. For specific information about the organization of resource data within executable files, refer to the resource formats documentation.

For information about how to create standard resources, refer to the following table:

| Resource | Topic |
|---------------------|-------------------------------------|
| Accelerator table | Keyboard Accelerators |
| Bitmap | Bitmaps |
| Cursor | Cursors |
| Dialog box | Dialog Boxes |
| Enhanced metafile | Metafiles |
| Font | Fonts and Text |
| Icon | Icons |
| Menu | Menus |
| Message-table entry | Your message-compiler documentation |
| String-table entry | Strings |
| Version information | Version Information |

For information about how to include resource data in a Win32-based executable file, refer to the documentation for your resource compiler.

Finding and Loading Resources

Before using a resource, an application must load it into memory. The **FindResource** and **FindResourceEx** functions find a resource in a module and return a handle to the binary resource data. **FindResource** locates a resource by type and name. **FindResourceEx** locates the resource by type, name, and language. Information about **FindResource** in this topic also applies to **FindResourceEx**.

The **LoadResource** function uses the resource handle returned by **FindResource** to load the resource into memory. After an application loads a resource by using **LoadResource**, the system automatically unloads and reloads the resource as memory conditions and application execution require. Thus, an application need not explicitly unload a resource it no longer needs.

An application can use **FindResource** and **LoadResource** to find and load any type of resource, but these functions should be used only if the application must access the binary resource data for subsequent function calls. To use a resource immediately, an application should use one of the following resource-specific functions to find and load resources in one call.

| Function | Action |
|-------------------------|---|
| FormatMessage | Loads and formats a message-table entry. |
| LoadAccelerators | Loads an accelerator table. |
| LoadBitmap | Loads a bitmap resource. |
| LoadCursor | Loads a cursor resource. |
| LoadIcon | Loads an icon resource. |
| LoadImage | Loads an icon, cursor, bitmap, or enhanced metafile resource. |
| LoadMenu | Loads a menu resource. |
| LoadString | Loads a string-table entry. |

Before terminating, an application should release the memory occupied by accelerator tables, bitmaps, cursors, icons, and menus by using one of the functions in the following table.

| Resource | Release function |
|-------------------|--------------------------------|
| Accelerator table | DestroyAcceleratorTable |
| Bitmap | DeleteObject |
| Cursor | DestroyCursor |
| Icon | DestroyIcon |
| Menu | DestroyMenu |

When the application terminates, the system automatically releases the memory occupied by the other types of resources.

Adding, Deleting, and Replacing Resources

Applications must frequently add, delete, or replace resources in executable files. Two methods can be used to accomplish these tasks. The first method is to edit the resource-definition file, recompile the resources, and add the recompiled resources to the application's executable file. The second method is to copy the resource data directly into the application's executable file.

For example, to localize an English-language application for use in Norway, it may be necessary to replace the English dialog box with one using Norwegian. A developer creates an appropriate dialog box by using a dialog-box editor or by writing a template in the resource-definition file. The developer then recompiles the resources and adds the new resources to the application's executable file.

If an appropriate dialog box exists in binary form, however, the developer can copy the data directly to the executable file being localized by using three Win32 functions. The **BeginUpdateResource** function creates an update handle for the executable file whose resources are to be changed. The **UpdateResource** function uses this handle to add, delete, or replace a resource in the executable file. The **EndUpdateResource** function closes the handle.

After an update handle to an executable file is created by **BeginUpdateResource**, an application can use **UpdateResource** repeatedly to make changes to the resource data. Each call to **UpdateResource** contributes to an internal list of additions, deletions, and replacements but does not actually write the data to the executable file. Immediately before closing the update handle, **EndUpdateResource** writes the accumulated changes to the executable file.

Sometimes, an application must copy resources or find resource sizes. The **LoadLibrary** function provides a module handle to an executable file whose resources are to be copied, and the **LockResource** function provides a pointer to the resource data in the specified module. The **SizeofResource** function returns the size, in bytes, of a specified resource.

Enumerating Resources

Three Win32 functions enable an application to obtain lists of resource types, names, and languages in a specified module. The **EnumResourceTypes** function provides a list of resource types found in the module, the **EnumResourceNames** function provides the name of each resource within a given type, and the **EnumResourceLanguages** function provides the language of each resource of a given name and type. These functions and their associated callback functions enable applications to create a list of all resources in a module. This process is described in *Creating a Resource List*.

Resource File Formats

This section describes the format of the *binary resource file* that the resource compiler creates based on the contents of the resource-definition file. This file usually has an `.res`

extension. The linker reformats the .res file into a resource object file and then links it to the executable file of a Win32-based application.

A binary resource file consists of a number of concatenated resource entries. Each entry consists of a resource header and the data for that resource. A resource header is **DWORD**-aligned in the file and consists of the following:

- A **DWORD** that contains the size of the resource header
- A **DWORD** that contains the size of the resource data
- The resource type
- The resource name
- Additional resource information

The **RESOURCEHEADER** structure describes the format of this header. The data for the resource follows the resource header and is specific to each type of resource. Some resources also employ a resource-specific group header structure to provide information about a group of resources. For a group list of the structures that describe the format of resources, see *Resource Structures*.

Accelerator Table Resources

An accelerator table is one resource entry in a resource file. It does not have a group header. An **ACCELTABLEENTRY** structure describes each entry in the accelerator table. Multiple accelerator tables are permitted.

Cursor and Icon Resources

The system handles each icon and cursor as a single file. However, these are stored in .res files and in executable files as a group of icon resources or a group of cursor resources. The file formats of icon and cursor resources are similar. In the .res file a resource group header follows all of the individual icon or cursor group components.

The format of each icon component closely resembles the format of the .ico file. Each icon image is stored in a **BITMAPINFO** structure followed by the color device-independent bitmap (DIB) bits of the icon's **XOR** mask. The monochrome DIB bits of the icon's **AND** mask follow the color DIB bits.

The format of each cursor component resembles the format of the .cur file. Each cursor image is stored in a **BITMAPINFO** structure followed by the monochrome device-independent bitmap (DIB) bits of the cursor's **XOR** mask, and then by the monochrome DIB bits of the cursor's **AND** mask. Note that there is a difference in the bitmaps of the two resources: unlike icons, cursor **XOR** masks do not have color DIB bits. Although the bitmaps of the cursor masks are monochrome and do not have DIB headers or color tables, the bits are still in DIB format with respect to alignment and direction. Another significant difference between cursors and icons is that cursors have a hotspot and icons do not.

The group header for both icon and cursor resources consists of a **NEWHEADER** structure plus one or more **RESDIR** structures. There is one **RESDIR** structure for each icon or cursor. The group header contains the information a Win32-based application needs to select the correct icon or cursor to display. Both the group header and the data that repeats for each icon or cursor in the group have a fixed length. This allows the application to access randomly the information.

Dialog-Box Resources

A dialog box is also one resource entry in the resource file. It consists of one **DLGTEMPLATE** dialog-box header structure plus one **DLGITEMTEMPLATE** structure for each control in the dialog box. The **DLGTEMPLATEEX** and **DLGITEMTEMPLATEEX** structures describe the format of extended dialog-box resources.

Font Resources

Fonts are stored in the resource file as a group of resources. Individual fonts make up a font group. A **FONT Statement** resource definition statement in the .rc file defines each font. Each individual font in the resource consists of the complete contents of the related .fnt file. A **FONTGROUPHDR** structure follows all the individual font components in the .res file.

Font resources are not added to the resources of a specific application. Instead, they are normally added to executable files that have a .fon extension. These files are usually resource-only dynamic-link libraries (DLLs) instead of applications.

Menu Resources

A menu resource consists of a **MENUHEADER** structure followed by one or more **NORMALMENUITEM** or **POPUPMENUITEM** structures, one for each menu item in the menu template. The **MENUEX_TEMPLATE_HEADER** and **MENUEX_TEMPLATE_ITEM** structures describe the format of extended menu resources.

Message-Table Resources

A message table is a resource that contains formatted text for display as an error message or in a message box. The main structure in a message-table resource is the **MESSAGE_RESOURCE_DATA** structure.

Version Resources

The main structure in a version resource is the **VS_FIXEDFILEINFO** structure. Additional structures include the **VarFileInfo** structure to store language information data, and **StringFileInfo** for user-defined string information. All strings in a version resource are in Unicode format for Win32-based applications. Each block of information is aligned on a **DWORD** boundary.


Getting More Information About Resources

The companion DVD that is bundled inside the Base Services volume of the *Microsoft Win32 Developer's Reference Library* has the complete set of reference information for Resources. Publishing constraints associated with volumes in the Windows Programming Reference Series—which are governed by the mission to provide concise, compact, and portable reference books—did not allow the reference section for Resources to be included in the printed version.

However, in order to provide you with the most complete and comprehensive guide to Win32 development, the Win32 Library includes all of its information in electronic form on the DVD. If you have not already, go through the installation process on the companion DVD, and the entire body of Resource programming reference (and much, much more) will be a click away.

Carets

A *caret* is a blinking line, block, or bitmap in the client area of a window. The caret typically indicates the place at which text or graphics will be inserted. The following illustration shows some common variations in the appearance of the caret:

Underline
Vertical Line |
Solid Block █
Gray Block ▒
Bitmap 

A Win32-based application can create a caret, change its blink time, and display, hide, or relocate the caret.

About Carets

The system provides one caret per message queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive. For more information on keyboard input, see *Keyboard Input*.

Use the **CreateCaret** function to specify the parameters for a caret. The system forms a caret by inverting the pixel color within the rectangle specified by the caret's position, width, and height. The width and height are specified in logical units; therefore, the appearance of a caret is subject to the window's mapping mode.

Caret Visibility

After the caret is defined, use the **ShowCaret** function to make the caret visible. When the caret appears, it automatically begins flashing. To display a solid caret, the system inverts every pixel in the rectangle; to display a gray caret, the system inverts every other pixel; to display a bitmap caret, the system inverts only the white bits of the bitmap.

Caret Blink Time

The elapsed time, in milliseconds, required to invert the caret is called the *blink time*. The caret will blink as long as the thread that owns the message queue has a message pump processing the messages.

The user can set the blink time of the caret using the Control Panel and applications should respect the settings that the user has chosen. An application can determine the caret's blink time by using the **GetCaretBlinkTime** function. If you are writing an application that allows the user to adjust the blink time, such as a Control Panel applet, use the **SetCaretBlinkTime** function to set the rate of the blink time to a specified number of milliseconds.

The *flash time* is the elapsed time, in milliseconds, required to display, invert, and restore the caret's display. The flash time of a caret is twice as much as the blink time.

Caret Position

You can determine the position of the caret using the **GetCaretPos** function. The position, in client coordinates, is copied to a **POINT** structure specified by a parameter in **GetCaretPos**. An application can move a caret in a window by using the **SetCaretPos** function. A window can move a caret only if it already owns the caret. **SetCaretPos** can move the caret whether it is visible or not.

Removing a Caret

You can temporarily remove a caret by hiding it, or you can permanently remove the caret by destroying it. To hide the caret, use the **HideCaret** function. This is useful when your application must redraw the screen while processing a message, but must keep the caret out of the way. When the application finishes drawing, it can display the caret again by using the **ShowCaret** function. Hiding the caret does not destroy its shape or invalidate the insertion point. Hiding the caret is cumulative; that is, if the application calls **HideCaret** five times, it must also call **ShowCaret** five times before the caret will reappear.

To remove the caret from the screen and destroy its shape, use the **DestroyCaret** function. **DestroyCaret** destroys the caret only if the window involved in the current task owns the caret.

Caret Reference

Caret Functions

CreateCaret

The **CreateCaret** function creates a new shape for the system caret and assigns ownership of the caret to the specified window. The caret shape can be a line, block, or bitmap.

```
BOOL CreateCaret(  
    HWND hWnd,           // handle to owner window  
    HBITMAP hBitmap,    // handle to bitmap for caret shape  
    int nWidth,         // caret width  
    int nHeight         // caret height  
);
```

Parameters

hWnd

[in] Handle to the window that owns the caret.

hBitmap

[in] Handle to the bitmap that defines the caret shape. If this parameter is NULL, the caret is solid. If this parameter is (HBITMAP) 1, the caret is gray. If this parameter is a bitmap handle, the caret is the specified bitmap. The bitmap handle must have been created by the **CreateBitmap**, **CreateDIBitmap**, or **LoadBitmap** function.

If *hBitmap* is a bitmap handle, **CreateCaret** ignores the *nWidth* and *nHeight* parameters; the bitmap defines its own width and height.

nWidth

[in] Specifies the width of the caret in logical units. If this parameter is zero, the width is set to the system-defined window border width. If *hBitmap* is a bitmap handle, **CreateCaret** ignores this parameter.

nHeight

[in] Specifies the height, in logical units, of the caret. If this parameter is zero, the height is set to the system-defined window border height. If *hBitmap* is a bitmap handle, **CreateCaret** ignores this parameter.

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 *nWidth* and *nHeight* parameters specify the caret's width and height, in logical units; the exact width and height, in pixels, depend on the window's mapping mode.

CreateCaret automatically destroys the previous caret shape, if any, regardless of the window that owns the caret. The caret is hidden until the application calls the **ShowCaret** function to make the caret visible.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

You can retrieve the width or height of the system's window border by using the **GetSystemMetrics** function, specifying the `SM_CXBORDER` and `SM_CYBORDER` values. Using the window border width or height guarantees that the caret will be visible on a high-resolution 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

Carets Overview, Caret Functions, **CreateBitmap**, **CreateDIBitmap**, **DestroyCaret**, **GetSystemMetrics**, **HideCaret**, **LoadBitmap**, **ShowCaret**

DestroyCaret

The **DestroyCaret** function destroys the caret's current shape, frees the caret from the window, and removes the caret from the screen.

If the caret shape is based on a bitmap, **DestroyCaret** does not free the bitmap.

BOOL DestroyCaret(VOID);

Parameters

This function has no parameters.

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

DestroyCaret destroys the caret only if a window in the current task owns the caret. If a window that is not in the current task owns the caret, **DestroyCaret** does nothing and returns FALSE.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

! 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

Carets Overview, Caret Functions, **CreateCaret**, **HideCaret**, **ShowCaret**

GetCaretBlinkTime

The **GetCaretBlinkTime** function returns the elapsed time, in milliseconds, required to invert the caret's pixels. The user can set this value using the Control Panel.

UINT GetCaretBlinkTime(VOID);

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the blink time, in milliseconds.

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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Carets Overview, Caret Functions, **SetCaretBlinkTime**

GetCaretPos

The **GetCaretPos** function copies the caret's position, in client coordinates, to the specified **POINT** structure.

```
BOOL GetCaretPos(  
    LPPPOINT lpPoint // client coordinates  
);
```

Parameters

lpPoint

[out] Pointer to the **POINT** structure that is to receive the client coordinates of the caret.

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 caret position is always given in the client coordinates of the window that contains the caret.

! 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

Carets Overview, Caret Functions, **POINT**, **SetCaretPos**

HideCaret

The **HideCaret** function removes the caret from the screen. Hiding a caret does not destroy its current shape or invalidate the insertion point.

```

BOOL HideCaret(
    HWND hWnd // handle to window with caret
);

```

Parameters

hWnd

[in] Handle to the window that owns the caret. If this parameter is NULL, **HideCaret** searches the current task for the window that owns the caret.

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

HideCaret hides the caret only if the specified window owns the caret. If the specified window does not own the caret, **HideCaret** does nothing and returns FALSE.

Hiding is cumulative. If your application calls **HideCaret** five times in a row, it must also call **ShowCaret** five times before the caret is displayed.

! 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

Carets Overview, Caret Functions, **CreateCaret**, **DestroyCaret**, **GetCaretPos**, **SetCaretPos**, **ShowCaret**

SetCaretBlinkTime

The **SetCaretBlinkTime** function sets the caret blink time to the specified number of milliseconds. The blink time is the elapsed time, in milliseconds, required to invert the caret's pixels.

```

BOOL SetCaretBlinkTime(
    UINT uMilliseconds // blink time
);

```

Parameters

uMSeconds

[in] Specifies the new blink time, in milliseconds.

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 user can set the blink time using the Control Panel. Applications should respect the setting that the user has chosen. The **SetCaretBlinkTime** function should only be used by application that allow the user to set the blink time, such as a Control Panel applet.

If you change the blink time, subsequently activated applications will use the modified blink time, even if you restore the previous blink time when you lose the keyboard focus or become inactive. This is due to the multithreaded environment, where deactivation of your application is not synchronized with the activation of another application. This feature allows the system to activate another application even if the current application has stopped responding.

! 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

Carets Overview, Caret Functions, **GetCaretBlinkTime**

SetCaretPos

The **SetCaretPos** function moves the caret to the specified coordinates. If the window that owns the caret was created with the `CS_OWNDC` class style, then the specified coordinates are subject to the mapping mode of the device context associated with that window.

```
BOOL SetCaretPos(  
    int X, // horizontal position  
    int Y  // vertical position  
);
```

Parameters

X

[in] Specifies the new x-coordinate of the caret.

Y

[in] Specifies the new y-coordinate of the caret.

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

SetCaretPos moves the caret whether or not the caret is hidden.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive. A window can set the caret position only if it owns the caret.

! 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

Carets Overview, **Caret Functions**, **GetCaretPos**, **HideCaret**, **ShowCaret**

ShowCaret

The **ShowCaret** function makes the caret visible on the screen at the caret's current position. When the caret becomes visible, it begins flashing automatically.

```
BOOL ShowCaret(  
    HWND hWnd // handle to window with caret  
);
```

Parameters

hWnd

[in] Handle to the window that owns the caret. If this parameter is **NULL**, **ShowCaret** searches the current task for the window that owns the caret.

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

ShowCaret shows the caret only if the specified window owns the caret, the caret has a shape, and the caret has not been hidden two or more times in a row. If one or more of these conditions is not met, **ShowCaret** does nothing and returns FALSE.

Hiding is cumulative. If your application calls **HideCaret** five times in a row, it must also call **ShowCaret** five times before the caret reappears.

The system provides one caret per queue. A window should create a caret only when it has the keyboard focus or is active. The window should destroy the caret before losing the keyboard focus or becoming inactive.

! 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

Carets Overview, Caret Functions, **CreateCaret**, **DestroyCaret**, **GetCaretPos**, **HideCaret**, **SetCaretPos**

Cursors

A *cursor* is a small picture whose location on the screen is controlled by a pointing device, such as a mouse, pen, or trackball. In the remainder of this overview, the term *mouse* refers to any pointing device.

When the user moves the mouse, the system moves the cursor accordingly. The cursor functions in the Microsoft Win32 application programming interface (API) enable applications to create, load, display, animate, move, confine, and destroy cursors.

About Cursors

The Microsoft Win32 API provides a set of standard cursors that are available for any application to use at any time. The SDK header files contain identifiers for the standard cursors—the identifiers have the “IDC_” prefix.

Each standard cursor has a corresponding default image associated with it. The user or an application can replace the default image associated with any standard cursor at any time. An application replaces a default image by using the **SetSystemCursor** function.

An application can use the **GetIconInfo** function to retrieve the current image for a cursor and can draw the cursor by using the **DrawIconEx** function. To draw the default image for a standard cursor, specify the `DI_COMPAT` flag in the call to **DrawIconEx**. If you do not specify the `DI_COMPAT` flag, **DrawIconEx** draws the standard cursor using the image that the user specified.

Custom cursors are designed for use in a specific application and can be any design the developer defines. The following illustration shows several custom cursors:



Cursors can be either monochrome or color, and either static or animated. The type of cursor used on a particular computer system depends on the system's display. Old displays such as VGA do not support color or animated cursors. New displays, whose display drivers use the device-independent bitmap (DIB) engine, support them.

Cursors and icons are similar and can be used interchangeably in many situations. The only difference between them is that an image specified as a cursor must be in the format that the display can support. For example, a cursor must be monochrome for a VGA display.

Cursor Reference

Cursor Functions

ClipCursor

The **ClipCursor** function confines the cursor to a rectangular area on the screen. If a subsequent cursor position (set by the **SetCursorPos** function or the mouse) lies outside the rectangle, the system automatically adjusts the position to keep the cursor inside the rectangular area.

```
BOOL ClipCursor(
    CONST RECT *lpRect // screen coordinates
);
```

Parameters

lpRect

[in] Pointer to the **RECT** structure that contains the screen coordinates of the upper-left and lower-right corners of the confining rectangle. If this parameter is `NULL`, the cursor is free to move anywhere on the screen.

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 cursor is a shared resource. If an application confines the cursor, it must release the cursor by using **ClipCursor** before relinquishing control to another application.

The calling process must have WINSTA_WRITEATTRIBUTES access to the window station.

! 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

Cursors Overview, Cursor Functions, **GetClipCursor**, **GetCursorPos**, **RECT**, **SetCursorPos**

CopyCursor

The **CopyCursor** function copies the specified cursor.

```
HCURSOR CopyCursor(  
    HCURSOR pcur // handle to cursor  
);
```

Parameters

pcur

[in] Handle to the cursor to be copied.

Return Values

If the function succeeds, the return value is the handle to the duplicate cursor.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **CopyCursor** function enables an application or DLL to obtain the handle to a cursor shape owned by another module. Then, if the other module is freed, the application is still able to use the cursor shape.

Before closing, an application must call the **DestroyCursor** function to free any system resources associated with the cursor.

Do not use the **CopyCursor** function for animated cursors. Instead, use the **CopyImage** 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Cursors Overview, Cursor Functions, **CopyIcon**, **CopyImage**, **DestroyCursor**, **GetCursor**, **SetCursor**, **ShowCursor**

CreateCursor

The **CreateCursor** function creates a cursor having the specified size, bit patterns, and hot spot.

```
HCURSOR CreateCursor(
    HINSTANCE hInst,           // handle to application instance
    int xHotSpot,             // x-coordinate of hot spot
    int yHotSpot,             // y-coordinate of hot spot
    int nWidth,               // cursor width
    int nHeight,              // cursor height
    CONST VOID *pvANDPlane,   // AND mask array
    CONST VOID *pvXORPlane    // XOR mask array
);
```

Parameters

hInst

[in] Handle to the current instance of the application creating the cursor.

xHotSpot

[in] Specifies the horizontal position of the cursor's hot spot.

yHotSpot

[in] Specifies the vertical position of the cursor's hot spot.

nWidth

[in] Specifies the width, in pixels, of the cursor.

nHeight

[in] Specifies the height, in pixels, of the cursor.

pvANDPlane

[in] Pointer to an array of bytes that contains the bit values for the **AND** mask of the cursor, as in a device-dependent monochrome bitmap.

pvXORPlane

[in] Pointer to an array of bytes that contains the bit values for the **XOR** mask of the cursor, as in a device-dependent monochrome bitmap.

Return Values

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

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The *nWidth* and *nHeight* parameters must specify a width and height that are supported by the current display driver, because the system cannot create cursors of other sizes. To determine the width and height supported by the display driver, use the **GetSystemMetrics** function, specifying the SM_CXCURSOR or SM_CYCURSOR value.

Before closing, an application must call the **DestroyCursor** function to free any system resources associated with the cursor.

! 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

Cursors Overview, Cursor Functions, **Createlcon**, **DestroyCursor**, **GetModuleHandle**, **GetSystemMetrics**, **SetCursor**

DestroyCursor

The **DestroyCursor** function destroys a cursor and frees any memory the cursor occupied. Do not use this function to destroy a shared cursor.

```
BOOL DestroyCursor(  
    HCURSOR hCursor // handle to cursor to destroy  
);
```

Parameters

hCursor

[in] Handle to the cursor to be destroyed. The cursor must not be in use.

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 **DestroyCursor** function destroys a nonshared cursor. Do not use this function to destroy a shared cursor. A shared cursor is valid as long as the module from which it was loaded remains in memory. The following functions obtain a shared cursor:

- **LoadCursor**
- **LoadCursorFromFile**
- **LoadImage** (if you use the LR_SHARED flag)
- **CopyImage** (if you use the LR_COPYRETURNORG flag and the *hImage* parameter is a shared cursor)

! 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

Cursors Overview, Cursor Functions, **CopyCursor**, **CopyImage**, **CreateCursor**, **LoadCursor**, **LoadCursorFromFile**, **LoadImage**

GetClipCursor

The **GetClipCursor** function retrieves the screen coordinates of the rectangular area to which the cursor is confined.

```
BOOL GetClipCursor(  
    LPRECT lpRect // screen coordinates  
);
```

Parameters

lpRect

[out] Pointer to a **RECT** structure that receives the screen coordinates of the confining rectangle. The structure receives the dimensions of the screen if the cursor is not confined to a rectangle.

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 cursor is a shared resource. If an application confines the cursor with the **ClipCursor** function, it must later release the cursor by using **ClipCursor** before relinquishing control to another application.

The calling process must have **WINSTA_READATTRIBUTES** access to the window station.

! 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

Cursors Overview, Cursor Functions, **ClipCursor**, **GetCursorPos**, **RECT**

GetCursor

The **GetCursor** function retrieves the handle to the current cursor.

```
HCURSOR GetCursor(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is the handle to the current cursor. If there is no cursor, the return value is NULL.

Windows 98, and Windows NT 4.0 SP3 and later: To get the cursor for another thread, use **GetGUIThreadInfo**.

! 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

Cursors Overview, Cursor Functions, **GetGUIThreadInfo**, **SetCursor**

GetCursorInfo

The **GetCursorInfo** function retrieves information about the global cursor.

```
BOOL GetCursorInfo(  
    PCURSORINFO pci // cursor information  
);
```

Parameters

pci

[out] Pointer to a **CURSORINFO** structure that receives the information.

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**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Cursors Overview, Cursor Functions, **CURSORINFO**

GetCursorPos

The **GetCursorPos** function retrieves the cursor's position, in screen coordinates.

```
BOOL GetCursorPos(  
    LPPPOINT lpPoint // cursor position  
);
```

Parameters

lpPoint

[out] Pointer to a **POINT** structure that receives the screen coordinates of the cursor.

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 cursor position is always specified in screen coordinates, and is not affected by the mapping mode of the window that contains the cursor.

The calling process must have **WINSTA_READATTRIBUTES** access to the window station.

! 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

Cursors Overview, Cursor Functions, **ClipCursor**, **POINT**, **SetCursor**, **SetCursorPos**, **ShowCursor**

LoadCursor

The **LoadCursor** function loads the specified cursor resource from the executable (.exe) file associated with an application instance.

Note This function has been superseded by the **LoadImage** function.

```
HCURSOR LoadCursor(
    HINSTANCE hInstance, // handle to application instance
    LPCTSTR lpCursorName // name or resource identifier
);
```

Parameters

hInstance

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

lpCursorName

[in] Pointer to a null-terminated string that contains the name of the cursor 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 also can be used to create this value.

To use one of the cursors predefined in the Microsoft Win32 API, the application must set the *hInstance* parameter to NULL and the *lpCursorName* parameter to one of the following values:

| Value | Meaning |
|-----------------|--|
| IDC_APPSTARTING | Standard arrow and small hourglass |
| IDC_ARROW | Standard arrow |
| IDC_CROSS | Crosshair |
| IDC_HAND | Windows 2000: Hand |
| IDC_HELP | Arrow and question mark |
| IDC_IBEAM | I-beam |
| IDC_ICON | Obsolete for applications marked version 4.0 or later |
| IDC_NO | Slashed circle |
| IDC_SIZE | Obsolete for applications marked version 4.0 or later; use IDC_SIZEALL |
| IDC_SIZEALL | Four-pointed arrow pointing north, south, east, and west |
| IDC_SIZENESW | Double-pointed arrow pointing northeast and southwest |
| IDC_SIZENS | Double-pointed arrow pointing north and south |
| IDC_SIZENWSE | Double-pointed arrow pointing northwest and southeast |
| IDC_SIZEWE | Double-pointed arrow pointing west and east |

| Value | Meaning |
|-------------|----------------|
| IDC_UPARROW | Vertical arrow |
| IDC_WAIT | Hourglass |

Return Values

If the function succeeds, the return value is the handle to the newly loaded cursor.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **LoadCursor** function loads the cursor resource only if it has not been loaded; otherwise, it retrieves the handle to the existing resource. This function returns a valid cursor handle only if the *lpCursorName* parameter is a pointer to a cursor resource. If *lpCursorName* is a pointer to any type of resource other than a cursor (such as an icon), the return value is not NULL, even though it is not a valid cursor handle.

The **LoadCursor** function searches the cursor resource most appropriate for the cursor for the current display device. The cursor resource can be a color or monochrome 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 *winuser.h*; include *windows.h*.

Library: Use *user32.lib*.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Cursors Overview, Cursor Functions, **LoadImage**, **MAKEINTRESOURCE**, **SetCursor**, **SetCursorPos**, **ShowCursor**

LoadCursorFromFile

The **LoadCursorFromFile** function creates a cursor based on data contained in a file. The file is specified by its name or by a system cursor identifier. The function returns a handle to the newly created cursor. Files containing cursor data may be in either cursor (.cur) or animated cursor (.ani) format.

```
HCURSOR LoadCursorFromFile(  
    LPCTSTR lpFileName // file or identifier  
);
```

Parameters

lpFileName

[in] Specifies the source of the file data to be used to create the cursor. The data in the file must be in either .cur or .ani format.

If the high-order word of *lpFileName* is nonzero, it is a pointer to a string that is a fully qualified name of a file containing cursor data.

If the high-order word of *lpFileName* is zero, the low-order word is a system cursor identifier. The function then searches the **[cursors]** section in the WIN.INI file for the file associated with the name of that system cursor. For a list of cursor identifiers, see Remarks.

Return Values

If the function is successful, the return value is a handle to the new cursor.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**. **GetLastError** may return the following value:

| Value | Meaning |
|----------------------|-------------------------------------|
| ERROR_FILE_NOT_FOUND | The specified file cannot be found. |

Remarks

The following is a list of system cursor names and identifiers:

| Cursor name | Cursor identifier |
|---------------|-------------------------------|
| "AppStarting" | OCR_APPSTARTING |
| "Arrow" | OCR_NORMAL |
| "Crosshair" | OCR_CROSS |
| "Hand" | Windows 2000: OCR_HAND |
| "Help" | OCR_HELP |
| "IBeam" | OCR_IBEAM |
| "Icon" | OCR_ICON |
| "No" | OCR_NO |
| "Size" | OCR_SIZE |
| "SizeAll" | OCR_SIZEALL |
| "SizeNESW" | OCR_SIZENESW |
| "SizeNS" | OCR_SIZENS |
| "SizeNWSE" | OCR_SIZENWSE |
| "SizeWE" | OCR_SIZEWE |
| "UpArrow" | OCR_UP |
| "Wait" | OCR_WAIT |

For example, if the Win.ini file contains the following:

```
[Cursors]
Arrow = "arrow.cur"
```

then the following call causes **LoadCursorFromFile** to obtain cursor data from the file Arrow.cur:

```
LoadCursorFromFile((LPWSTR)OCR_NORMAL)
```

If the Win.ini file does not contain an entry for the specified system cursor, the function fails and returns NULL.

! 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

Cursors Overview, Cursor Functions, **LoadCursor**, **SetCursor**, **SetSystemCursor**

SetCursor

The **SetCursor** function establishes the cursor shape.

```
HCURSOR SetCursor(
    HCURSOR hCursor // handle to cursor
);
```

Parameters

hCursor

[in] Handle to the cursor. The cursor must have been created by the **CreateCursor** function or loaded by the **LoadCursor** or **LoadImage** function. If this parameter is NULL, the cursor is removed from the screen.

Windows 95: The width and height of the cursor must be the values returned by the **GetSystemMetrics** function for SM_CXCURSOR and SM_CYCURSOR. In addition, either the cursor bit depth must match the bit depth of the display or the cursor must be monochrome.

Return Values

The return value is the handle to the previous cursor, if there was one.

If there was no previous cursor, the return value is `NULL`.

Remarks

The cursor is set only if the new cursor is different from the previous cursor; otherwise, the function returns immediately.

The cursor is a shared resource. A window should set the cursor shape only when the cursor is in its client area or when the window is capturing mouse input. In systems without a mouse, the window should restore the previous cursor before the cursor leaves the client area or before it relinquishes control to another window.

If your application must set the cursor while it is in a window, make sure the class cursor for the specified window's class is set to `NULL`. If the class cursor is not `NULL`, the system restores the class cursor each time the mouse is moved.

The cursor is not shown on the screen if the internal cursor display count is less than zero. This occurs if the application uses the **ShowCursor** function to hide the cursor more times than to show the cursor.

! 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

Cursors Overview, Cursor Functions, **CreateCursor**, **GetCursor**, **GetSystemMetrics**, **LoadCursor**, **LoadImage**, **SetCursorPos**, **ShowCursor**

SetCursorPos

The **SetCursorPos** function moves the cursor to the specified screen coordinates. If the new coordinates are not within the screen rectangle set by the most recent **ClipCursor** function call, the system automatically adjusts the coordinates, so that the cursor stays within the rectangle.

```
BOOL SetCursorPos(  
    int X, // horizontal position  
    int Y  // vertical position  
);
```

Parameters

X

[in] Specifies the new x-coordinate of the cursor, in screen coordinates.

Y

[in] Specifies the new y-coordinate of the cursor, in screen coordinates.

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 cursor is a shared resource. A window should move the cursor only when the cursor is in its client area.

The calling process must have `WINSTA_WRITEATTRIBUTES` access to the window station.

! 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

Cursors Overview, Cursor Functions, **ClipCursor**, **GetCursorPos**, **SetCaretPos**, **SetCursor**, **ShowCursor**

SetSystemCursor

The **SetSystemCursor** function enables an application to customize the system cursors. It replaces the contents of the system cursor specified by the *id* parameter with the contents of the cursor specified by the *hcur* parameter, and then destroys *hcur*.

```
BOOL SetSystemCursor (  
    HCURSOR hcur, // handle to cursor  
    DWORD id      // system cursor identifier  
);
```

Parameters

hcur

[in] Handle to a cursor. The function replaces the contents of the system cursor specified by *id* with the contents of the cursor handled by *hcur*.

The system destroys *hcur* by calling the **DestroyCursor** function. Therefore, *hcur* cannot be a cursor loaded using the **LoadCursor** function. To specify a cursor loaded from a resource, copy the cursor using the **CopyCursor** function, then pass the copy to **SetSystemCursor**.

id

[in] Specifies the system cursor to replace with the contents of *hcur*. This parameter can be one of the following values:

| Value | Meaning |
|-----------------|--|
| OCR_APPSTARTING | Standard arrow and small hourglass |
| OCR_CROSS | Crosshair |
| OCR_HAND | Windows 2000: Hand |
| OCR_HELP | Arrow and question mark |
| OCR_IBEAM | I-beam |
| OCR_NO | Slashed circle |
| OCR_NORMAL | Standard arrow |
| OCR_SIZEALL | Four-pointed arrow pointing north, south, east, and west |
| OCR_SIZENESW | Double-pointed arrow pointing northeast and southwest |
| OCR_SIZENS | Double-pointed arrow pointing north and south |
| OCR_SIZEWSE | Double-pointed arrow pointing northwest and southeast |
| OCR_SIZEWE | Double-pointed arrow pointing west and east |
| OCR_UP | Vertical arrow |
| OCR_WAIT | Hourglass |

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**.

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.

See Also

Cursors Overview, Cursor Functions, **DestroyCursor**, **LoadCursor**, **LoadCursorFromFile**, **SetCursor**

ShowCursor

The **ShowCursor** function displays or hides the cursor.

```
int ShowCursor(  
    BOOL bShow // cursor visibility  
);
```

Parameters

bShow

[in] Specifies whether the internal display counter is to be incremented or decremented. If *bShow* is TRUE, the display count is incremented by one. If *bShow* is FALSE, the display count is decremented by one.

Return Values

The return value specifies the new display counter.

Remarks

This function sets an internal display counter that determines whether the cursor should be displayed. The cursor is displayed only if the display count is greater than or equal to 0. If a mouse is installed, the initial display count is 0. If no mouse is installed, the display count is -1.

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

Cursors Overview, Cursor Functions, **ClipCursor**, **GetCursorPos**, **SetCursor**, **SetCursorPos**

Cursor Structures

CURSORINFO

The **CURSORINFO** structure contains global cursor information.

```
typedef struct tagCURSORINFO {
    DWORD   cbSize;
    DWORD   flags;
    HCURSOR hCursor;
    POINT   ptScreenPos;
} CURSORINFO, *PCURSORINFO, +LPCURSORINFO;
```

Members

cbSize

Specifies the size, in bytes, of the structure.

flags

Specifies the cursor state. This parameter can be one of the following values:

| Value | Meaning |
|----------------|------------------------|
| 0 | The cursor is hidden. |
| CURSOR_SHOWING | The cursor is showing. |

hCursor

Handle to the cursor.

ptScreenPos

A **POINT** structure that receives the screen coordinates of the cursor.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Cursors Overview, Cursor Structures, **GetCursorInfo**, **POINT**

Cursor Messages

The following message is used with cursors:

WM_SETCURSOR

WM_SETCURSOR

The **WM_SETCURSOR** message is sent to a window if the mouse causes the cursor to move within a window and mouse input is not captured.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_SETCURSOR  
    WPARAM wParam,      // handle to window (HWND)  
    LPARAM lParam       // hit-test code and mouse message  
);
```

Parameters

wParam

Handle to the window that contains the cursor.

lParam

The low-order word of *lParam* specifies the hit-test code.

The high-order word of *lParam* specifies the identifier of the mouse message.

Return Values

If an application processes this message, it should return TRUE to halt further processing or FALSE to continue.

Remarks

The high-order word of *lParam* is zero when the window enters menu mode.

The **DefWindowProc** function passes the **WM_SETCURSOR** message to a parent window before processing. If the parent window returns TRUE, further processing is halted. Passing the message to a window's parent window gives the parent window control over the cursor's setting in a child window. The **DefWindowProc** function also uses this message to set the cursor to an arrow if it is not in the client area, or to the registered class cursor if it is in the client area. If the low-order word of the *lParam* parameter is HTERROR and the high-order word of *lParam* specifies that one of the mouse buttons is pressed, **DefWindowProc** calls the **MessageBeep** 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 winuser.h; include windows.h.

+ See Also

Cursors Overview, Cursor Messages, **DefWindowProc**, **HIWORD**, **LOWORD**, **MessageBeep**

Icons

An *icon* is a picture that consists of a bitmap image combined with a mask to create transparent areas in the picture. The term icon can refer to either of the following:

- A single icon image. This is a resource of type `RT_ICON`.
- A group of images, from which the system or an application can choose the most appropriate icon based on size and color depth. This is a resource of type `RT_GROUP_ICON`.

About Icons

The system uses icons throughout the user interface to represent objects, such as files, folders, shortcuts, applications, and documents. The icon functions provided by the Win32 API enable applications to create, load, display, arrange, animate, and destroy icons.

Icon Reference

Icon Functions

CopyIcon

The **CopyIcon** function copies the specified icon from another module to the current module.

```
HICON CopyIcon(  
    HICON hIcon // handle to icon  
);
```

Parameters

hIcon
[in] Handle to the icon to be copied.

Return Values

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

If the function fails, the return value is `NULL`. To get extended error information, call **GetLastError**.

Remarks

The **CopyIcon** function enables an application or dynamic-link library (DLL) to get its own handle to an icon owned by another module. If the other module is freed, the application icon still will be able to use the icon.

Before closing, an application must call the **DestroyIcon** function to free any system resources associated with the icon.

! 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

Icons Overview, Icon Functions, **CopyCursor**, **DestroyIcon**, **DrawIcon**, **DrawIconEx**

Createlcon

The **Createlcon** function creates an icon that has the specified size, colors, and bit patterns.

```
HICON CreateIcon(
    HINSTANCE hInstance, // handle to application instance
    int nWidth, // icon width
    int nHeight, // icon height
    BYTE cPlanes, // number of planes in XOR bitmask
    BYTE cBitsPixel, // number of BPP in XOR bitmask
    CONST BYTE *lpbANDbits, // AND bitmask
    CONST BYTE *lpbXORbits // XOR bitmask
);
```

Parameters

hInstance

[in] Handle to the instance of the module creating the icon.

nWidth

[in] Specifies the width, in pixels, of the icon.

nHeight

[in] Specifies the height, in pixels, of the icon.

cPlanes

[in] Specifies the number of planes in the XOR bitmask of the icon.

cBitsPixel

[in] Specifies the number of bits-per-pixel in the XOR bitmask of the icon.

lpbANDbits

[in] Pointer to an array of bytes that contains the bit values for the AND bitmask of the icon. This bitmask describes a monochrome bitmap.

lpbXORbits

[in] Pointer to an array of bytes that contains the bit values for the XOR bitmask of the icon. This bitmask describes a monochrome or device-dependent color bitmap.

Return Values

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

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The *nWidth* and *nHeight* parameters must specify, respectively, a width and a height supported by the current display driver, because the system cannot create icons of other sizes. To determine the width and height supported by the display driver, use the **GetSystemMetrics** function, specifying the SM_CXICON or SM_CYICON value.

CreateIcon applies the following truth table to the AND and XOR bitmasks:

| AND bitmask | XOR bitmask | Display |
|-------------|-------------|----------------|
| 0 | 0 | Black |
| 0 | 1 | White |
| 1 | 0 | Screen |
| 1 | 1 | Reverse 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Icons Overview, Icon Functions, **GetSystemMetrics**

CreteIconFromResource

The **CreteIconFromResource** function creates an icon or cursor from resource bits describing the icon.

To specify a desired height or width, use the **CreteIconFromResourceEx** function.

```
HICON CreateIconFromResource(
    PBYTE presbits, // icon or cursor bits
    DWORD dwResSize, // number of bytes in bit buffer
    BOOL fIcon, // icon or cursor
    DWORD dwVer // format version
);
```

Parameters

presbits

[in] Pointer to a buffer containing the icon or cursor resource bits. These bits are typically loaded by calls to the **LookupIconIdFromDirectory** (in Windows 95, you also can call **LookupIconIdFromDirectoryEx**) and **LoadResource** functions.

dwResSize

[in] Specifies the size, in bytes, of the set of bits pointed to by the *presbits* parameter.

fIcon

[in] Specifies whether an icon or a cursor is to be created. If this parameter is TRUE, an icon is to be created. If it is FALSE, a cursor is to be created.

dwVer

[in] Specifies the version number of the icon or cursor format for the resource bits pointed to by the *presbits* parameter. This parameter can be one of the following values:

| Format | DwVer |
|-------------|------------|
| Windows 2.x | 0x00020000 |
| Windows 3.x | 0x00030000 |

All Win32-based applications use the Windows 3.x format for icons and cursors.

Return Values

If the function succeeds, the return value is a handle to the icon or cursor.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **CreteIconFromResource**, **CreteIconFromResourceEx**, **CreteIconIndirect**, **GetIconInfo**, **LookupIconIdFromDirectory**, and **LookupIconIdFromDirectoryEx**

functions allow shell applications and icon browsers to examine and use resources throughout the system.

! 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

Icons Overview, Icon Functions, **CreteIconFromResourceEx**, **CreteIconIndirect**, **GetIconInfo**, **LoadResource**, **LookupIconIdFromDirectory**, **LookupIconIdFromDirectoryEx**

CreteIconFromResourceEx

The **CreteIconFromResourceEx** function creates an icon or cursor from resource bits describing the icon.

```
HICON CreateIconFromResourceEx(
    PBYTE pIconBits, // icon or cursor bits
    DWORD cbIconBits, // number of bytes in bit buffer
    BOOL fIcon, // icon or cursor
    DWORD dwVersion, // format version
    int cxDesired, // desired width of icon or cursor
    int cyDesired, // desired height of icon or cursor
    UINT uFlags // load resource options
);
```

Parameters

pIconBits

[in] Pointer to a buffer containing the icon or cursor resource bits. These bits are typically loaded by calls to the **LookupIconIdFromDirectoryEx** and **LoadResource** functions.

cbIconBits

[in] Specifies the size, in bytes, of the set of bits pointed to by the *pIconBits* parameter.

fIcon

[in] Specifies whether an icon or a cursor is to be created. If this parameter is TRUE, an icon is to be created. If it is FALSE, a cursor is to be created.

dwVersion

[in] Specifies the version number of the icon or cursor format for the resource bits pointed to by the *pblconBits* parameter. This parameter can be one of the following values:

| Format | dwVersion |
|-------------|------------|
| Windows 2.x | 0x00020000 |
| Windows 3.x | 0x00030000 |

All Win32-based applications use the Windows 3.x format for icons and cursors.

cxDesired

[in] Specifies the desired width, in pixels, of the icon or cursor. If this parameter is zero, the function uses the SM_CXICON or SM_CXCURSOR system metric value to set the width.

cyDesired

[in] Specifies the desired height, in pixels, of the icon or cursor. If this parameter is zero, the function uses the SM_CYICON or SM_CYCURSOR system metric value to set the height.

uFlags

[in] Specifies a combination of the following values:

| Value | Meaning |
|-----------------|--------------------------------------|
| LR_DEFAULTCOLOR | Uses the default color format. |
| LR_MONOCHROME | Creates a monochrome icon or cursor. |

Return Values

If the function succeeds, the return value is a handle to the icon or cursor.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **CreateIconFromResource**, **CreateIconFromResourceEx**, **CreateIconIndirect**, **GetIconInfo**, and **LookupIconIdFromDirectoryEx** functions allow shell applications and icon browsers to examine and use resources throughout the system.

! 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

Icons Overview, Icon Functions, **BITMAPINFOHEADER**, **CreateIconFromResource**, **CreateIconIndirect**, **GetIconInfo**, **LoadResource**, **LookupIconIdFromDirectoryEx**

CreateIconIndirect

The **CreateIconIndirect** function creates an icon or cursor from an **ICONINFO** structure.

```
HICON CreateIconIndirect(  
    PICONINFO piconinfo // icon information  
);
```

Parameters

piconinfo

[in] Pointer to an **ICONINFO** structure the function uses to create the icon or cursor.

Return Values

If the function succeeds, the return value is a handle to the icon or cursor that is created.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The system copies the bitmaps in the **ICONINFO** structure before creating the icon or cursor. Because the system may select temporarily the bitmaps in a device context, the **hbmMask** and **hbmColor** members of the **ICONINFO** structure should not already be selected into a device context. The application must continue to manage the original bitmaps, and delete them when they are no longer necessary.

When you are finished using the icon, destroy it using the **DestroyIcon** 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

Icons Overview, Icon Functions, **DestroyIcon**, **ICONINFO**

DestroyIcon

The **DestroyIcon** function destroys an icon and frees any memory the icon occupied.

```
BOOL DestroyIcon(  
    HICON hIcon // handle to icon  
);
```

Parameters

hIcon

[in] Handle to the icon to be destroyed. The icon must not be in use.

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

It is only necessary to call **DestroyIcon** for icons and cursors created with the **CreateIconIndirect** and the **CopyIcon** functions. Do not use this function to destroy a shared icon. A shared icon is valid as long as the module from which it was loaded remains in memory. The following functions obtain a shared icon:

- **LoadIcon**
- **LoadImage** (if you use the LR_SHARED flag)
- **CopyImage** (if you use the LR_COPYRETURNORG flag and the *hImage* parameter is a shared icon)

! 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

Icons Overview, Icon Functions, **CopyIcon**, **CreateIconIndirect**

DrawIcon

The **DrawIcon** function draws an icon or cursor into the specified device context.

To specify additional drawing options, use the **DrawIconEx** function.

```
BOOL DrawIcon(  
    HDC hDC,           // handle to DC  
    int X,             // x-coordinate of upper-left corner  
    int Y,             // y-coordinate of upper-left corner  
    HICON hIcon       // handle to icon  
);
```

Parameters

hDC

[in] Handle to the device context into which the icon or cursor will be drawn.

X

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

Y

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

hIcon

[in] Handle to the icon to be drawn.

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

DrawIcon places the icon's upper-left corner at the location specified by the *X* and *Y* parameters. The location is subject to the current mapping mode of the 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.

Library: Use user32.lib.

+ See Also

Icons Overview, Icon Functions, **CreatelIcon**, **DrawIconEx**, **LoadIcon**

DrawIconEx

The **DrawIconEx** function draws an icon or cursor into the specified device context, performing the specified raster operations, and stretching or compressing the icon or cursor, as specified.

```
BOOL DrawIconEx(  
    HDC hdc,                // handle to device context  
    int xLeft,              // x-coordinate of upper-left corner  
    int yTop,               // y-coordinate of upper-left corner  
    HICON hIcon,            // handle to icon  
    int cxWidth,            // icon width  
    int cyWidth,            // icon height  
    UINT iStepIfAniCur,    // frame index, animated cursor  
    HBRUSH hbrFlickerFreeDraw, // handle to background brush  
    UINT diFlags             // icon-drawing flags  
);
```

Parameters

hdc

[in] Handle to the device context into which the icon or cursor will be drawn.

xLeft

[in] Specifies the logical x-coordinate of the upper-left corner of the icon or cursor.

yTop

[in] Specifies the logical y-coordinate of the upper-left corner of the icon or cursor.

hIcon

[in] Handle to the icon or cursor to be drawn. This parameter can identify an animated cursor.

cxWidth

[in] Specifies the logical width of the icon or cursor. If this parameter is zero and the *diFlags* parameter is `DI_DEFAULTSIZE`, the function uses the `SM_CXICON` or `SM_CXCURSOR` system metric value to set the width. If this parameter is zero and `DI_DEFAULTSIZE` is not used, the function uses the actual resource width.

cyWidth

[in] Specifies the logical height of the icon or cursor. If this parameter is zero and the *diFlags* parameter is `DI_DEFAULTSIZE`, the function uses the `SM_CYICON` or `SM_CYCURSOR` system metric value to set the width. If this parameter is zero and `DI_DEFAULTSIZE` is not used, the function uses the actual resource height.

iStepIfAniCur

[in] Specifies the index of the frame to draw, if *hIcon* identifies an animated cursor. This parameter is ignored if *hIcon* does not identify an animated cursor.

hbrFlickerFreeDraw

[in] Handle to a brush that the system uses for flicker-free drawing. If *hbrFlickerFreeDraw* is a valid brush handle, the system creates an offscreen bitmap

using the specified brush for the background color, draws the icon or cursor into the bitmap, and copies the bitmap into the device context identified by *hdc*. If *hbrFlickerFreeDraw* is NULL, the system draws the icon or cursor directly into the device context.

diFlags

[in] Specifies the drawing flags. This parameter can be one of the following values:

| Value | Meaning |
|----------------|---|
| DI_COMPAT | Draws the icon or cursor using the system default image, instead of the user-specified image. |
| DI_DEFAULTSIZE | Draws the icon or cursor using the width and height specified, by the system metric values for cursors or icons, if the <i>cxWidth</i> and <i>cyWidth</i> parameters are set to zero. If this flag is not specified, and <i>cxWidth</i> and <i>cyWidth</i> are set to zero, the function uses the actual resource size. |
| DI_IMAGE | Draws the icon or cursor using the image. |
| DI_MASK | Draws the icon or cursor using the mask. |
| DI_NORMAL | Combination of DI_IMAGE and DI_MASK. |

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 **DrawIconEx** function places the icon's upper-left corner at the location specified by the *xLeft* and *yTop* parameters. The location is subject to the current mapping mode of the device context.

! 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

Icons Overview, Icon Functions, **CopyImage**, **LoadImage**

Duplicatelcon

The **Duplicatelcon** function creates a duplicate of a specified icon.

```
HICON Duplicatelcon(  
    HINSTANCE hInst,    // not used  
    HICON hIcon,        // handle to icon  
);
```

Parameters

hInst

[in] This parameter is not used; it can be NULL.

hIcon

[in] Handle to the icon to be duplicated.

Return Values

If successful, the function returns the handle to the new icon that was created. If unsuccessful, it returns NULL.

Remarks

You must destroy the icon handle returned by **Duplicatelcon** by calling the **DestroyIcon** function.

! 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 shellapi.h.

Library: Use shell32.lib.

+ See Also

Icons Overview, Icon Functions, **DestroyIcon**

ExtractAssociatedIcon

The **ExtractAssociatedIcon** function returns a handle to an indexed icon found in a file or an icon found in an associated executable file.

```
HICON ExtractAssociatedIcon(  
    HINSTANCE hInst,    // application instance handle  
    LPTSTR lpIconPath,  // file name  
    LPWORD lpIcon       // icon index  
);
```

Parameters

hInst

[in] Specifies the instance of the application calling the function.

lpIconPath

[in] Pointer to a string that specifies the full path and file name of the file that contains the icon. The function extracts the icon handle from that file, or from an executable file associated with that file.

If the icon handle is obtained from an executable file, the function stores the full path and file name of that executable in the string pointed to by *lpIconPath*.

lpIcon

[in] Pointer to a **WORD** that specifies the index of the icon whose handle is to be obtained.

If the icon handle is obtained from an executable file, the function stores the icon's identifier in the **WORD** pointed to by *lpIcon*.

Return Values

If the function succeeds, the return value is an icon handle. If the icon is extracted from an associated executable file, the function stores the full path and file name of the executable file in the string pointed to by *lpIconPath*, and stores the icon's identifier in the **WORD** pointed to by *lpIcon*.

If the function fails, the return value is NULL.

Remarks

The **ExtractAssociatedIcon** function first looks for the indexed icon in the file specified by *lpIconPath*. If the function cannot obtain the icon handle from that file, and the file has an associated executable file, it looks in that executable file for an icon. Associations with executable files are based on file name extensions, are stored in the per-user part of the registry, and can be defined using File Manager's **Associate** command.

! 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 shellapi.h.

Library: Use shell32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Icons Overview, Icon Functions, **ExtractIcon**

ExtractIcon

The **ExtractIcon** function retrieves a handle to an icon from the specified executable file, dynamic-link library (DLL), or icon file.

To retrieve an array of handles to large or small icons, use the **ExtractIconEx** function.

```
HICON ExtractIcon(  
    HINSTANCE hInst,           // instance handle  
    LPCTSTR lpszExeFileName,  // file name  
    UINT nIconIndex          // icon index  
);
```

Parameters

hInst

[in] Handle to the instance of the application calling the function.

lpszExeFileName

[in] Pointer to a null-terminated string specifying the name of an executable file, DLL, or icon file.

nIconIndex

[in] Specifies the zero-based index of the icon to retrieve. For example, if this value is 0, the function returns a handle to the first icon in the specified file.

If this value is -1, the function returns the total number of icons in the specified file. If the file is an executable file or DLL, the return value is the number of RT_GROUP_ICON resources. If the file is an .ico file, the return value is 1.

Windows 95/98, Windows NT 4.0, and Windows 2000: If this value is a negative number not equal to -1, the function returns a handle to the icon in the specified file whose resource identifier is equal to the absolute value of *nIconIndex*. For example, use -3 to extract the icon whose resource identifier is 3. To extract the icon whose resource identifier is 1, use the **ExtractIconEx** function.

Return Values

The return value is a handle to an icon. If the file specified was not an executable file, DLL, or icon file, the return is 1. If no icons were found in the file, the return value is NULL.

Remarks

You must destroy the icon handle returned by **ExtractIcon** by calling the **DestroyIcon** 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 shellapi.h.

Library: Use shell32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

 See Also

Icons Overview, Icon Functions, **DestroyIcon**, **ExtractIconEx**

ExtractIconEx

The **ExtractIconEx** function creates an array of handles to large or small icons extracted from the specified executable file, dynamic-link library (DLL), or icon file.

```

UINT ExtractIconEx(
    LPCTSTR IpszFile,           // file name
    int nIconIndex,           // icon index
    HICON *phIconLarge,       // large icon array
    HICON *phIconSmall,      // small icon array
    UINT nIcons               // number of icons to extract
);

```

Parameters

IpszFile

[in] Pointer to a null-terminated string specifying the name of an executable file, DLL, or icon file from which icons will be extracted.

nIconIndex

[in] Specifies the zero-based index of the first icon to extract. For example, if this value is zero, the function extracts the first icon in the specified file.

If this value is -1 and *phIconLarge* and *phIconSmall* are both NULL, the function returns the total number of icons in the specified file. If the file is an executable file or DLL, the return value is the number of RT_GROUP_ICON resources. If the file is an .ico file, the return value is 1.

Windows 95/98, Windows NT 4.0, and Windows 2000: If this value is a negative number, and either *phIconLarge* or *phIconSmall* is not NULL, the function begins by extracting the icon whose resource identifier is equal to the absolute value of *nIconIndex*. For example, use -3 to extract the icon whose resource identifier is 3.

phIconLarge

[out] Pointer to an array of icon handles that receives handles to the large icons extracted from the file. If this parameter is NULL, no large icons are extracted from the file.

phIconSmall

[out] Pointer to an array of icon handles that receives handles to the small icons extracted from the file. If this parameter is NULL, no small icons are extracted from the file.

nIcons

[in] Specifies the number of icons to extract from the file.

Return Values

If the *nIconIndex* parameter is -1, the *phIconLarge* parameter is NULL, and the *phIconSmall* parameter is NULL, then the return value is the number of icons contained in the specified file. Otherwise, the return value is the number of icons successfully extracted from the file.

Remarks

You must destroy all icons extracted by **ExtractIconEx** by calling the **DestroyIcon** function.

To retrieve the dimensions of the large and small icons, use the **GetSystemMetrics** function with the SM_CXICON, SM_CYICON, SM_CXSMICON, and SM_CYSMICON flags.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

Windows CE: Requires version 1.0 or later.

Header: Declared in shellapi.h.

Library: Use shell32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Icons Overview, Icon Functions, **DestroyIcon**, **ExtractIcon**

GetIconInfo

The **GetIconInfo** function retrieves information about the specified icon or cursor.

```
BOOL GetIconInfo(  
    HICON hIcon,           // icon handle  
    PICONINFO pIconInfo  // icon structure  
);
```

Parameters

hIcon

[in] Handle to the icon or cursor. To retrieve information about a standard icon or cursor, specify one of the following values:

| Value | Meaning |
|-----------------|---|
| IDC_APPSTARTING | Standard arrow and small hourglass cursor |
| IDC_ARROW | Standard arrow cursor |
| IDC_CROSS | Crosshair cursor |
| IDC_HAND | Windows 2000: Hand cursor |
| IDC_HELP | Arrow and question-mark cursor |
| IDC_IBEAM | I-beam cursor |
| IDC_NO | Slashed circle cursor |
| IDC_SIZEALL | Four-pointed arrow cursor pointing north, south, east, and west |
| IDC_SIZENESW | Double-pointed arrow cursor pointing northeast and southwest |
| IDC_SIZENS | Double-pointed arrow cursor pointing north and south |
| IDC_SIZENWSE | Double-pointed arrow cursor pointing northwest and southeast |
| IDC_SIZEWE | Double-pointed arrow cursor pointing west and east |
| IDC_UPARROW | Vertical-arrow cursor |
| IDC_WAIT | Hourglass cursor |
| IDI_APPLICATION | Application icon |
| IDI_ASTERISK | Asterisk icon |
| IDI_EXCLAMATION | Exclamation-point icon |
| IDI_HAND | Stop-sign icon |
| IDI_QUESTION | Question-mark icon |
| IDI_WINLOGO | Windows logo icon |

piconinfo

[out] Pointer to an **ICONINFO** structure. The function fills in the structure's members.

Return Values

If the function succeeds, the return value is nonzero and the function fills in the members of the specified **ICONINFO** structure.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

GetIconInfo creates bitmaps for the **hbmMask** and **hbmColor** members of **ICONINFO**. The calling application must manage these bitmaps and delete them when they are no longer necessary.

! 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

Icons Overview, Icon Functions, **Createlcon**, **CreatelconFromResource**, **CreatelconIndirect**, **Destroylcon**, **Drawlcon**, **DrawlconEx**, **ICONINFO**, **Loadlcon**, **LookuplconIdFromDirectory**

Loadlcon

The **Loadlcon** function loads the specified icon resource from the executable (.exe) file associated with an application instance.

Note This function has been superseded by the **LoadImage** function.

```
HICON Loadlcon(
    HINSTANCE hInstance, // handle to application instance
    LPCTSTR lpIconName // name string or resource identifier
);
```

Parameters

hInstance

[in] Handle to an instance of the module whose executable file contains the icon to be loaded. This parameter must be NULL when a standard icon is being loaded.

lpIconName

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

To use one of the predefined icons, set the *hInstance* parameter to NULL and the *lpIconName* parameter to one of the following values:

| Value | Description |
|-----------------|--------------------------|
| IDI_APPLICATION | Default application icon |
| IDI_ASTERISK | Same as IDI_INFORMATION |
| IDI_ERROR | Hand-shaped icon |

(continued)

(continued)

| Value | Description |
|-----------------|------------------------|
| IDI_EXCLAMATION | Same as IDI_WARNING |
| IDI_HAND | Same as IDI_ERROR |
| IDI_INFORMATION | Asterisk icon |
| IDI_QUESTION | Question-mark icon |
| IDI_WARNING | Exclamation-point icon |
| IDI_WINLOGO | Windows logo icon |

Return Values

If the function succeeds, the return value is a handle to the newly loaded icon.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

LoadIcon loads the icon resource only if it has not been loaded; otherwise, it retrieves a handle to the existing resource. The function searches the icon resource for the icon most appropriate for the current display. The icon resource can be a color or monochrome bitmap.

LoadIcon can only load an icon whose size conforms to the SM_CXICON and SM_CYICON system metric values. Use the **LoadImage** function to load icons of other sizes.

! 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

Icons Overview, Icon Functions, **Createlcon**, **LoadImage**, **MAKEINTRESOURCE**

LookupIconIdFromDirectory

The **LookupIconIdFromDirectory** function searches through icon or cursor data for the icon or cursor that best fits the current display device.

To specify a desired height or width, use the **LookupIconIdFromDirectoryEx** function.

```
int LookupIconIdFromDirectory(  
    PBYTE presbits, // resource data  
    BOOL fIcon      // icon or cursor  
);
```

Parameters

presbits

[in] Pointer to the icon or cursor directory data. Because this function does not validate the resource data, it causes a general protection (GP) fault or returns an undefined value if *presbits* is not pointing to valid resource data.

fIcon

[in] Specifies whether an icon or a cursor is sought. If this parameter is TRUE, the function is searching for an icon; if the parameter is FALSE, the function is searching for a cursor.

Return Values

If the function succeeds, the return value is an integer resource identifier for the icon or cursor that best fits the current display device.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A resource file of type RT_GROUP_ICON (RT_GROUP_CURSOR indicates cursors) contains icon (or cursor) data in several device-dependent and device-independent formats. **LookupIconIdFromDirectory** searches the resource file for the icon (or cursor) that best fits the current display device and returns its integer identifier. The **FindResource** and **FindResourceEx** functions use the **MAKEINTRESOURCE** macro with this identifier to locate the resource in the module.

The icon directory is loaded from a resource file with resource type RT_GROUP_ICON (or RT_GROUP_CURSOR for cursors), and an integer resource name for the specific icon to be loaded. **LookupIconIdFromDirectory** returns an integer identifier that is the resource name of the icon that best fits the current display device.

The **LoadIcon**, **LoadCursor**, and **LoadImage** functions use this function to search the specified resource data for the icon or cursor that best fits the current display 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Icons Overview, Icon Functions, **CreateIconFromResource**, **CreateIconIndirect**, **FindResource**, **FindResourceEx**, **GetIconInfo**, **LoadCursor**, **LoadIcon**, **LoadImage**, **LookupIconIdFromDirectoryEx**, **MAKEINTRESOURCE**

LookupIconIdFromDirectoryEx

The **LookupIconIdFromDirectoryEx** function searches through icon or cursor data for the icon or cursor that best fits the current display device.

```
int LookupIconIdFromDirectoryEx(
    PBYTE presbits, // resource data
    BOOL fIcon,     // icon or cursor
    int cxDesired,  // width of icon or cursor
    int cyDesired,  // height of icon or cursor
    UINT Flags      // resource options
);
```

Parameters

presbits

[in] Pointer to the icon or cursor directory data. Because this function does not validate the resource data, it causes a general protection (GP) fault or returns an undefined value if *presbits* is not pointing to valid resource data.

fIcon

[in] Specifies whether an icon or a cursor is sought. If this parameter is TRUE, the function is searching for an icon; if the parameter is FALSE, the function is searching for a cursor.

cxDesired

[in] Specifies the desired width, in pixels, of the icon. If this parameter is zero, the function uses the SM_CXICON or SM_CXCURSOR system metric value.

cyDesired

[in] Specifies the desired height, in pixels, of the icon. If this parameter is zero, the function uses the SM_CYICON or SM_CYCURSOR system metric value.

Flags

[in] Specifies a combination of the following values:

| Value | Meaning |
|-----------------|--------------------------------------|
| LR_DEFAULTCOLOR | Uses the default color format. |
| LR_MONOCHROME | Creates a monochrome icon or cursor. |

Return Values

If the function succeeds, the return value is an integer resource identifier for the icon or cursor that best fits the current display device.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

A resource file of type `RT_GROUP_ICON` (`RT_GROUP_CURSOR` indicates cursors) contains icon (or cursor) data in several device-dependent and device-independent formats. **LookupIconIdFromDirectoryEx** searches the resource file for the icon (or cursor) that best fits the current display device and returns its integer identifier. The **FindResource** and **FindResourceEx** functions use the **MAKEINTRESOURCE** macro with this identifier to locate the resource in the module.

The icon directory is loaded from a resource file with resource type `RT_GROUP_ICON` (or `RT_GROUP_CURSOR` for cursors), and an integer resource name for the specific icon to be loaded. **LookupIconIdFromDirectoryEx** returns an integer identifier that is the resource name of the icon that best fits the current display device.

The **LoadIcon**, **LoadImage**, and **LoadCursor** functions use this function to search the specified resource data for the icon or cursor that best fits the current display device.

! 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

Icons Overview, Icon Functions, **CreateIconFromResourceEx**, **CreateIconIndirect**, **FindResource**, **FindResourceEx**, **GetIconInfo**, **LoadCursor**, **LoadIcon**, **LoadImage**, **LookupIconIdFromDirectory**, **MAKEINTRESOURCE**

Icon Structures

ICONINFO

The **ICONINFO** structure contains information about an icon or cursor.

```
typedef struct _ICONINFO {
    BOOL    fIcon;
    DWORD   xHotspot;
    DWORD   yHotspot;
    HBITMAP hbmMask;
    HBITMAP hbmColor;
} ICONINFO;
```


Members

fIcon

Specifies whether this structure defines an icon or a cursor. A value of TRUE specifies an icon; FALSE specifies a cursor.

xHotspot

Specifies the x-coordinate of a cursor's hot spot. If this structure defines an icon, the hot spot is always in the center of the icon, and this member is ignored.

yHotspot

Specifies the y-coordinate of the cursor's hot spot. If this structure defines an icon, the hot spot is always in the center of the icon, and this member is ignored.

hbmMask

Specifies the icon bitmask bitmap. If this structure defines a black and white icon, this bitmask is formatted so that the upper half is the icon AND bitmask and the lower half is the icon XOR bitmask. Under this condition, the height should be an even multiple of two. If this structure defines a color icon, this mask only defines the AND bitmask of the icon.

hbmColor

Handle to the icon color bitmap. This member can be optional if this structure defines a black and white icon. The AND bitmask of **hbmMask** is applied with the SRCAND flag to the destination; subsequently, the color bitmap is applied (using XOR) to the destination by using the SRCINVERT flag.

! 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

Icons Overview, Icon Structures, **CreatelconIndirect**, **GetIconInfo**

ICONMETRICS

The **ICONMETRICS** structure contains the scalable metrics associated with icons. This structure is used with the **SystemParametersInfo** function when the SPI_GETICONMETRICS or SPI_SETICONMETRICS action is specified.

```
typedef struct tagICONMETRICS {
    UINT    cbSize;
    int     iHorzSpacing;
```

```

    int    iVertSpacing;
    int    iTitleWrap;
    LOGFONT lfFont;
} ICONMETRICS, FAR *LPICONMETRICS;

```

Members

cbSize

Specifies the size of the structure, in bytes.

iHorzSpacing and iVertSpacing

Horizontal and vertical space, in pixels, for each arranged icon.

iTitleWrap

Title-wrapping flag. If this member is nonzero, icon titles wrap to a new line. If this member is zero, titles do not wrap.

lfFont

Specifies the font to use for icon titles.

! 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`.

Unicode: Declared as Unicode and ANSI structures.

+ See Also

Icons Overview, Icon Structures, **SystemParametersInfo**

WM_ERASEBKGD

The **WM_ERASEBKGD** message is sent when the window background must be erased (for example, when a window is resized). The message is sent to prepare an invalidated portion of a window for painting.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_ERASEBKGD
    WPARAM wParam,      // handle to device context (HDC)
    LPARAM lParam       // not used
);

```

Parameters

wParam

Handle to the device context.

lParam

This parameter is not used.

Return Values

An application should return nonzero if it erases the background; otherwise, it should return zero.

Remarks

The **DefWindowProc** function erases the background by using the class background brush specified by the **hbrBackground** member of the **WNDCLASS** structure. If **hbrBackground** is NULL, the application should process the **WM_ERASEBKGD** message and erase the background.

An application should return nonzero in response to **WM_ERASEBKGD** if it processes the message and erases the background; this indicates that no further erasing is required. If the application returns zero, the window will remain marked for erasing. (Typically, this indicates that the **fErase** member of the **PAINTSTRUCT** structure will be TRUE.)

! 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

Icons Overview, Icon Messages, **BeginPaint**, **DefWindowProc**, **PAINTSTRUCT**, **WM_ICONERASEBKGD**, **WNDCLASS**

WM_ICONERASEBKGD

The **WM_ICONERASEBKGD** message is sent to a minimized window when the background of the icon must be filled before painting the icon. A window receives this message only if a class icon is defined for the window; otherwise, **WM_ERASEBKGD** is sent.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
```

```

UINT uMsg, // WM_ICONERASEBKGD
WPARAM wParam, // handle to device context (HDC)
LPARAM lParam // not used
);

```

Parameters

wParam

Handle to the device context of the icon.

lParam

This parameter is not used.

Return Values

An application should return nonzero if it processes this message.

Remarks

The **DefWindowProc** function fills the icon background with the class background brush of the parent window.

! 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

Icons Overview, Icon Messages, **DefWindowProc**, **WM_ERASEBKGD**

WM_PAINTICON

The **WM_PAINTICON** message is sent to a minimized window when the icon is to be painted, but only if the application is written for 16-bit Windows. A window receives this message only if a class icon is defined for the window; otherwise, **WM_PAINT** is sent, instead.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd, // handle to window
    UINT uMsg, // WM_PAINTICON
    WPARAM wParam, // not used
    LPARAM lParam // not used
);

```

Parameters

This message has no parameters.

Return Values

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function draws the class icon. For compatibility with 16-bit Windows, *wParam* is TRUE. However, this value has no significance.

! 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

Icons Overview, Icon Messages, **DefWindowProc**, **WM_ICONERASEBKGD**, **WM_PAINT**

Menus

About Menus

A *menu* is a list of *menu items*. Clicking a menu item opens a submenu or causes the application to carry out a command.

Menu Bars and Menus

A menu is arranged in a hierarchy. At the top level of the hierarchy is the *menu bar*, *menus* drop down from the menu bar, and at the lower levels are *submenus*. A menu bar is sometimes called a *top-level menu*, and the menus and submenus are also known as *pop-up menus*.

A menu item can either carry out a command or open a submenu. An item that carries out a command is called a *command item* or a *command*.

An item on the menu bar almost always opens a menu. Menu bars rarely contain command items. A menu opened from the menu bar drops down from the menu bar and is sometimes called a *drop-down menu*. When a drop-down menu is displayed, it is attached to the menu bar. A menu item on the menu bar that opens a drop-down menu is also called a *menu name*.

The menu names on a menu bar represent the main categories of commands that an application provides. Selecting a menu name from the menu bar typically opens a menu whose menu items correspond to the commands in a category. For example, a menu bar might contain a **File** menu name that, when clicked by the user, activates a menu with menu items such as **New**, **Open**, and **Save**. To get information about a menu bar, call **GetMenuBarInfo**.

Only an overlapped or pop-up window can contain a menu bar; a child window cannot contain one. If the window has a title bar, the system positions the menu bar just below it. A menu bar is always visible. A submenu is not visible, however, until the user selects a menu item that activates it. For more information about overlapped and pop-up windows, see *Window Types*.

Each menu must have an owner window. The system sends messages to a menu's owner window when the user selects the menu or chooses an item from the menu. These messages are described in *Messages Used with Menus*.

Shortcut Menus

The system also provides *shortcut menus*. A shortcut menu is not attached to the menu bar; it can appear anywhere on the screen. An application typically associates a shortcut menu with a portion of a window, such as the client area, or with a specific object, such as an icon. For this reason, these menus are also called *context menus*.

A shortcut menu remains hidden until the user activates it, typically by right-clicking a selection, a toolbar, or a taskbar button. The menu is usually displayed at the position of the caret or mouse cursor.

The Window Menu

The **Window** menu (also known as the **System** menu or **Control** menu) is a pop-up menu defined and managed almost exclusively by the operating system. The user can open the window menu by either clicking the application icon on the title bar or right-clicking anywhere on the title bar.

The **Window** menu provides a standard set of menu items that the user can choose to change a window's size or position, or close the application. Items on the window menu can be added, deleted, and modified, but most applications just use the standard set of menu items. An overlapped, pop-up, or child window can have a window menu. It is uncommon for an overlapped or pop-up window not to include a window menu.

When the user chooses a command from the **Window** menu, the system sends a **WM_SYSCOMMAND** message to the menu's owner window. In most applications, the window procedure does not process messages from the window menu. Instead, it simply passes the messages to the **DefWindowProc** function for system-default processing of the message. If an application adds a command to the window menu, the window procedure must process the command.

An application can use the **GetSystemMenu** function to create a copy of the default window menu to modify. Any window that does not use the **GetSystemMenu** function to make its own copy of the window menu receives the standard window menu.

Help Identifier

Associated with each menu bar, menu, submenu, and shortcut menu is a help identifier. If the user presses the F1 key while the menu is active, this value is sent to the owner window as part of a **WM_HELP** message.

Menu Reference

Menu Functions

AppendMenu

The **AppendMenu** function appends a new item to the end of the specified menu bar, drop-down menu, submenu, or shortcut menu. You can use this function to specify the content, appearance, and behavior of the menu item.

Note The **AppendMenu** function has been superseded by the **InsertMenuItem** function. You still can use **AppendMenu**, however, if you do not need any of the extended features of **InsertMenuItem**.

```
BOOL AppendMenu(  
    HMENU hMenu,           // handle to menu  
    UINT uFlags,          // menu-item options  
    UINT_PTR uIDNewItem,  // identifier, menu, or submenu  
    LPCWSTR lpNewItem     // menu-item content  
);
```

Parameters

hMenu

[in] Handle to the menu bar, drop-down menu, submenu, or shortcut menu to be changed.

uFlags

[in] Specifies flags to control the appearance and behavior of the new menu item. This parameter can be a combination of the values listed in the following Remarks section.

uIDNewItem

[in] Specifies either the identifier of the new menu item or, if the *uFlags* parameter is set to MF_POPUP, a handle to the drop-down menu or submenu.

lpNewItem

[in] Specifies the content of the new menu item. The interpretation of *lpNewItem* depends on whether the *uFlags* parameter includes the MF_BITMAP, MF_OWNERDRAW, or MF_STRING flag, as shown in the following table:

| Value | Description |
|--------------|---|
| MF_BITMAP | Contains a bitmap handle. |
| MF_OWNERDRAW | Contains an application-supplied value that can be used to maintain additional data related to the menu item. The value is in the itemData member of the structure pointed to by the <i>lparam</i> parameter of either the WM_MEASURE or WM_DRAWITEM message that is sent when the menu is created or its appearance is updated. |
| MF_STRING | Contains a pointer to a null-terminated string. |

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 application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window.

To get keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the **WM_MENUCHAR** message. For more information, see *Owner-Drawn Menus* and the **WM_MENUCHAR** message.

The following flags can be set in the *uFlags* parameter:

| Value | Description |
|-----------------|---|
| MF_BITMAP | Uses a bitmap as the menu item. The <i>lpNewItem</i> parameter contains a handle to the bitmap. |
| MF_CHECKED | Places a check mark next to the menu item. If the application provides check-mark bitmaps (see SetMenuItemBitmaps), this flag displays the check-mark bitmap next to the menu item. |
| MF_DISABLED | Disables the menu item so that it cannot be selected, but the flag does not dim it. |
| MF_ENABLED | Enables the menu item so that it can be selected, and restores it from its dimmed state. |
| MF_GRAYED | Disables the menu item and dims it, so that it cannot be selected. |
| MF_MENUBARBREAK | Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line. |

(continued)

(continued)

| Value | Description |
|--------------|---|
| MF_MENUBREAK | Places the item on a new line (for a menu bar) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns. |
| MF_OWNERDRAW | Specifies that the item is an owner-drawn item. Before the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message to retrieve the width and height of the menu item. Then, the WM_DRAWITEM message is sent to the window procedure of the owner window whenever the appearance of the menu item must be updated. |
| MF_POPUP | Specifies that the menu item opens a drop-down menu or submenu. The <i>uidNewItem</i> parameter specifies a handle to the drop-down menu or submenu. This flag is used to add a menu name to a menu bar, or a menu item that opens a submenu to a drop-down menu, submenu, or shortcut menu. |
| MF_SEPARATOR | Draws a horizontal dividing line. This flag is used only in a drop-down menu, submenu, or shortcut menu. The line cannot be dimmed, disabled, or highlighted. The <i>lpNewItem</i> and <i>uidNewItem</i> parameters are ignored. |
| MF_STRING | Specifies that the menu item is a text string; the <i>lpNewItem</i> parameter is a pointer to the string. |
| MF_UNCHECKED | Does not place a check mark next to the item (default). If the application supplies check-mark bitmaps (see SetMenuItemBitmaps), this flag displays the clear bitmap next to the menu item. |

The following groups of flags cannot be used together:

- MF_BITMAP, MF_STRING, and MF_OWNERDRAW
- MF_CHECKED and MF_UNCHECKED
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_MENUBARBREAK and MF_MENUBREAK

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

Menus Overview, Menu Functions, **CreateMenu**, **DeleteMenu**, **DestroyMenu**, **DrawMenuBar**, **InsertMenu**, **InsertMenuItem**, **ModifyMenu**, **RemoveMenu**, **SetMenuItemBitmaps**

CheckMenuItem

The **CheckMenuItem** function sets the state of the specified menu item's check-mark attribute to either selected or clear.

Note The **CheckMenuItem** function has been superseded by the **SetMenuItemInfo** function. You still can use **CheckMenuItem**, however, if you do not need any of the extended features of **SetMenuItemInfo**.

```
DWORD CheckMenuItem(
    HMENU hmenu,           // handle to menu
    UINT uIDCheckItem,    // menu item to check or uncheck
    UINT uCheck            // menu item options
);
```

Parameters

hmenu

[in] Handle to the menu of interest.

uIDCheckItem

[in] Specifies the menu item whose check-mark attribute is to be set, as determined by the *uCheck* parameter.

uCheck

[in] Specifies flags that control the interpretation of the *uIDCheckItem* parameter and the state of the menu item's check-mark attribute. This parameter can be a combination of either MF_BYCOMMAND, or MF_BYPOSITION and MF_CHECKED or MF_UNCHECKED.

| Value | Meaning |
|---------------|--|
| MF_BYCOMMAND | Indicates that the <i>uIDCheckItem</i> parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default, if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified. |
| MF_BYPOSITION | Indicates that the <i>uIDCheckItem</i> parameter gives the zero-based relative position of the menu item. |
| MF_CHECKED | Sets the check-mark attribute to the selected state. |
| MF_UNCHECKED | Sets the check-mark attribute to the clear state. |

Return Values

The return value specifies the previous state of the menu item (either MF_CHECKED or MF_UNCHECKED). If the menu item does not exist, the return value is -1.

Remarks

An item in a menu bar cannot have a check mark.

The *uIDCheckItem* parameter identifies a item that opens a submenu or a command item. For an item that opens a submenu, the *uIDCheckItem* parameter must specify the position of the item. For a command item, the *uIDCheckItem* parameter can specify either the item's position or its identifier.

! 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

Menus Overview, Menu Functions, **EnableMenuItem**, **GetMenuItemID**, **SetMenuItemBitmaps**, **SetMenuItemInfo**

CheckMenuRadioItem

The **CheckMenuRadioItem** function checks a specified menu item and makes it a radio item. At the same time, the function clears all other menu items in the associated group, and clears the radio-item type flag for those items.

```

BOOL CheckMenuRadioItem(
    HMENU hmenu, // handle to menu
    UINT idFirst, // identifier or position of first item
    UINT idLast, // identifier or position of last item
    UINT idCheck, // identifier or position of menu item
    UINT uFlags // function options
);

```

Parameters

hmenu

[in] Handle to the menu that contains the group of menu items.

idFirst

[in] Identifier or position of the first menu item in the group.

idLast

[in] Identifier or position of the last menu item in the group.

idCheck

[in] Identifier or position of the menu item to check.

uFlags

[in] Value specifying the meaning of *idFirst*, *idLast*, and *idCheck*. If this parameter is MF_BYCOMMAND, the other parameters specify menu item identifiers. If it is MF_BYPOSITION, the other parameters specify the menu item positions.

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, use the **GetLastError** function.

Remarks

The **CheckMenuRadioItem** function sets the MFT_RADIOCHECK type flag and the MFS_CHECKED state for the item specified by *idCheck* and, at the same time, clears both flags for all other items in the group. The selected item is displayed using a bullet bitmap instead of a check-mark bitmap.

For more information about menu item type and state flags, see the **MENUITEMINFO** structure.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Menus Overview, Menu Functions, **MENUITEMINFO**

CreateMenu

The **CreateMenu** function creates a menu. The menu is initially empty, but it can be filled with menu items by using the **InsertMenuItem**, **AppendMenu**, and **InsertMenu** functions.

HMENU CreateMenu(VOID);

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the newly created menu.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

Resources associated with a menu that is assigned to a window are freed automatically. If the menu is not assigned to a window, an application must free system resources associated with the menu before closing. An application frees menu resources by calling the **DestroyMenu** function.

Windows 95: The system can support a maximum of 16,364 menu handles.

! 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

Menus Overview, Menu Functions, **AppendMenu**, **CreatePopupMenu**, **DestroyMenu**, **InsertMenu**, **InsertMenuItem**, **SetMenu**

CreatePopupMenu

The **CreatePopupMenu** function creates a drop-down menu, submenu, or shortcut menu. The menu is initially empty. You can insert or append menu items by using the **InsertMenuItem** function. You also can use the **InsertMenu** function to insert menu items, and the **AppendMenu** function to append menu items.

HMENU CreatePopupMenu(VOID);

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the newly created menu.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The application can add the new menu to an existing menu, or it can display a shortcut menu by calling the **TrackPopupMenuEx** or **TrackPopupMenu** function.

Resources associated with a menu that is assigned to a window are freed automatically. If the menu is not assigned to a window, an application must free system resources associated with the menu before closing. An application frees menu resources by calling the **DestroyMenu** function.

Windows 95: The system can support a maximum of 16,364 menu handles.

! 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

Menus Overview, Menu Functions, **AppendMenu**, **CreateMenu**, **DestroyMenu**, **InsertMenu**, **InsertMenuItem**, **SetMenu**, **TrackPopupMenu**, **TrackPopupMenuEx**

DeleteMenu

The **DeleteMenu** function deletes an item from the specified menu. If the menu item opens a menu or submenu, this function destroys the handle to the menu or submenu, and frees the memory used by the menu or submenu.

```
BOOL DeleteMenu(  
    HMENU hMenu,    // handle to menu  
    UINT uPosition, // menu item identifier or position  
    UINT uFlags     // option  
);
```

Parameters

hMenu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item to be deleted, as determined by the *uFlags* parameter.

uFlags

[in] Specifies how the *uPosition* parameter is interpreted. This parameter must be one of the following values:

| Value | Meaning |
|---------------|---|
| MF_BYCOMMAND | Indicates that <i>uPosition</i> gives the identifier of the menu item. The MF_BYCOMMAND flag is the default flag if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified. |
| MF_BYPOSITION | Indicates that <i>uPosition</i> gives the zero-based relative position of the menu item. |

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 application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window.

! 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

Menus Overview, Menu Functions, **DrawMenuBar**, **RemoveMenu**

DestroyMenu

The **DestroyMenu** function destroys the specified menu and frees any memory that the menu occupies.

```
BOOL DestroyMenu(
    HMENU hMenu // handle to menu
);
```

Parameters

hMenu

[in] Handle to the menu to be destroyed.

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

Before closing, an application must use the **DestroyMenu** function to destroy a menu that is not assigned to a window. A menu that is assigned to a window is automatically destroyed when the application closes.

DestroyMenu is recursive; that is, it will destroy the menu and all its submenus.

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

Menus Overview, Menu Functions, **CreateMenu**, **DeleteMenu**, **RemoveMenu**, **SetMenuItemInfo**

DrawMenuBar

The **DrawMenuBar** function redraws the menu bar of the specified window. If the menu bar changes after the system has created the window, this function must be called to draw the changed menu bar.

```
BOOL DrawMenuBar(  
    HWND hWnd // handle to window  
);
```

Parameters

hWnd

[in] Handle to the window whose menu bar needs redrawing.

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**.

! 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.

Library: Use user32.lib.

+ See Also

Menus Overview, Menu Functions, **DeleteMenu**, **InsertMenuItem**, **RemoveMenu**, **SetMenuItemInfo**

EnableMenuItem

The **EnableMenuItem** function enables, disables, or dims the specified menu item.

```

BOOL EnableMenuItem(
    HMENU hMenu,           // handle to menu
    UINT uIDEnableItem,   // menu item to update
    UINT uEnable          // options
);

```

Parameters

hMenu

[in] Handle to the menu.

uIDEnableItem

[in] Specifies the menu item to be enabled, disabled, or dimmed, as determined by the *uEnable* parameter. This parameter specifies an item in a menu bar, menu, or submenu.

uEnable

[in] Controls the interpretation of the *uIDEnableItem* parameter and indicate whether the menu item is enabled, disabled, or dimmed. This parameter must be a combination of either MF_BYCOMMAND or MF_BYPOSITION, and either MF_ENABLED, MF_DISABLED, or MF_GRAYED.

| Value | Meaning |
|---------------|--|
| MF_BYCOMMAND | Indicates that <i>uIDEnableItem</i> gives the identifier of the menu item. If neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified, the MF_BYCOMMAND flag is the default flag. |
| MF_BYPOSITION | Indicates that <i>uIDEnableItem</i> gives the zero-based relative position of the menu item. |

| Value | Meaning |
|-------------|---|
| MF_DISABLED | Indicates that the menu item is disabled, but not dimmed, so that it cannot be selected. |
| MF_ENABLED | Indicates that the menu item is enabled and restored from a dimmed state, so that it can be selected. |
| MF_GRAYED | Indicates that the menu item is disabled and dimmed, so that it cannot be selected. |

Return Values

The return value specifies the previous state of the menu item (it is either MF_DISABLED, MF_ENABLED, or MF_GRAYED). If the menu item does not exist, the return value is -1.

Remarks

An application must use the MF_BYPOSITION flag to specify the correct menu handle. If the menu handle to the menu bar is specified, the top-level menu item (an item in the menu bar) is affected. To set the state of an item in a drop-down menu or submenu by position, an application must specify a handle to the drop-down menu or submenu.

When an application specifies the MF_BYCOMMAND flag, the system checks all items that open submenus in the menu identified by the specified menu handle. Therefore, unless duplicate menu items are present, specifying the menu handle to the menu bar is sufficient.

The **InsertMenu**, **InsertMenuItem**, **LoadMenuIndirect**, **ModifyMenu**, and **SetMenuItemInfo** functions can also set the state (enabled, disabled, or dimmed) of a menu item.

When you change a window menu, the menu bar is not updated immediately. To force the update, call **DrawMenuBar**.

! 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

Menus Overview, Menu Functions, **DrawMenuBar**, **GetMenuItemID**, **InsertMenu**, **InsertMenuItem**, **LoadMenuIndirect**, **ModifyMenu**, **SetMenuItemInfo**, **WM_SYSCOMMAND**

EndMenu

The **EndMenu** function ends the calling thread's active menu.

```
BOOL EndMenu(VOID);
```

Parameters

This function has no parameters.

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

If a platform does not support **EndMenu**, send the owner of the active menu a **WM_CANCELMODE** message.

! 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.

+ See Also

Menus Overview, Menu Functions, **WM_CANCELMODE**

GetMenu

The **GetMenu** function retrieves a handle to the menu assigned to the specified window.

```
HMENU GetMenu(  
    HWND hwnd // handle to window  
);
```

Parameters

hwnd

[in] Handle to the window whose menu handle is to be retrieved.

Return Values

The return value is a handle to the menu. If the specified window has no menu, the return value is NULL. If the window is a child window, 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: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Menus Overview, Menu Functions, **GetSubMenu**, **SetMenu**

GetMenuBarInfo

The **GetMenuBarInfo** function retrieves information about the specified menu bar.

```

BOOL GetMenuBarInfo(
    HWND hwnd,           // handle to window
    LONG idObject,       // menu object
    LONG idItem,         // item identifier
    PMENUBARINFO pmbi // information
);

```

Parameters

hwnd

[in] Handle to the window (menu bar) whose information is to be retrieved.

idObject

[in] Specifies the menu object. This parameter can be one of the following values:

| Value | Meaning |
|---------------|---|
| OBJID_CLIENT | The pop-up menu associated with the window. |
| OBJID_MENU | The menu bar associated with the window (see the GetMenu function). |
| OBJID_SYSMENU | The system menu associated with the window (see the GetSystemMenu function). |

idItem

[in] Specifies the item for which to retrieve information. If this parameter is zero, the function retrieves information about the menu itself. If this parameter is 1, the function retrieves information about the first item on the menu, and so on.

pmbi

[out] Pointer to a **MENUBARINFO** structure that receives the information.

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**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Menus Overview, Menu Functions, **GetMenu**, **GetSystemMenu**, **MENUBARINFO**

GetMenuCheckMarkDimensions

The **GetMenuCheckMarkDimensions** function returns the dimensions of the default check-mark bitmap. The system displays this bitmap next to selected menu items. Before calling the **SetMenuItemBitmaps** function to replace the default check-mark bitmap for a menu item, an application must determine the correct bitmap size by calling **GetMenuCheckMarkDimensions**.

Note The **GetMenuCheckMarkDimensions** function is included only for compatibility with 16-bit versions of Windows. For Win32-based applications, use the **GetSystemMetrics** function with the `CXMENUCHECK` and `CYMENUCHECK` values to retrieve the bitmap dimensions.

LONG **GetMenuCheckMarkDimensions**(VOID)

Parameters

This function has no parameters.

Return Values

The return value specifies the height and width, in pixels, of the default check-mark bitmap. The high-order word contains the height; the low-order word contains the width.

! 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

Menus Overview, Menu Functions, **SetMenuItemBitmaps**

GetMenuDefaultItem

The **GetMenuDefaultItem** function determines the default menu item on the specified menu.

```
UINT GetMenuDefaultItem(
    HMENU hMenu,    // handle to menu
    UINT fByPos,    // retrieval option
    UINT gmdiFlags // function search option
);
```

Parameters

hMenu

[in] Handle to the menu for which to retrieve the default menu item.

fByPos

[in] Specifies whether to retrieve the menu item's identifier or its position. If this parameter is FALSE, the identifier is returned. Otherwise, the position is returned.

gmdiFlags

[in] Specifies how the function searches for menu items. This parameter can be zero or more of the following values:

| Value | Meaning |
|------------------|--|
| GMDI_GOINTOPOUPS | Specifies that if the default item is one that opens a submenu, the function is to search recursively in the corresponding submenu. If the submenu has no default item, the return value identifies the item that opens the submenu. By default, the function returns the first default item on the specified menu, regardless of whether it is an item that opens a submenu. |

(continued)

(continued)

| Value | Meaning |
|------------------|--|
| GMDI_USEDISABLED | Specifies that the function is to return a default item, even if it is disabled. By default, the function skips disabled or dimmed items. |

Return Values

If the function succeeds, the return value is the identifier or position of the menu item.

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

Menus Overview, Menu Functions, **SetMenuDefaultItem**

GetMenuInfo

The **GetMenuInfo** function gets information about a specified menu.

```

BOOL GetMenuInfo(
    HMENU hmenu,           // handle to menu
    LPCMENUINFO lpcmi,    // menu information
);

```

Parameters

hmenu

[in] Handle for a menu.

lpcmi

[out] Pointer to a **MENUINFO** structure containing information for the menu.

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**.

! 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`.

+ See Also

Menus Overview, Menu Functions

GetMenuItemCount

The **GetMenuItemCount** function determines the number of items in the specified menu.

```
int GetMenuItemCount(  
    HMENU hMenu // handle to menu  
);
```

Parameters

hMenu

[in] Handle to the menu to be examined.

Return Values

If the function succeeds, the return value specifies the number of items in the menu.

If the function fails, the return value is `-1`. 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Menus Overview, Menu Functions, **GetMenuItemID**

GetMenuItemID

The **GetMenuItemID** function retrieves the menu item identifier of a menu item located at the specified position in a menu.

```
UINT GetMenuItemID(  
    HMENU hMenu, // handle to menu  
    int nPos     // position of menu item  
);
```

Parameters

hMenu

[in] Handle to the menu that contains the item whose identifier is to be retrieved.

nPos

[in] Specifies the zero-based relative position of the menu item whose identifier is to be retrieved.

Return Values

The return value is the identifier of the specified menu item. If the menu item identifier is NULL, or if the specified item opens a submenu, the return value is -1 .

! 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

Menus Overview, Menu Functions, **GetMenuItemCount**

GetMenuItemInfo

The **GetMenuItemInfo** function retrieves information about a menu item.

```
BOOL GetMenuItemInfo(  
    HMENU hMenu, // handle to menu  
    UINT uItem,  // menu item  
    BOOL fByPosition, // meaning of uItem  
    LPMENUITEMINFO lpmii // menu item information  
);
```

Parameters

hMenu

[in] Handle to the menu that contains the menu item.

ulItem

[in] Identifier or position of the menu item to get information about. The meaning of this parameter depends on the value of *fByPosition*.

fByPosition

[in] Specifies the meaning of *ulItem*. If this parameter is FALSE, *ulItem* is a menu item identifier. Otherwise, it is a menu item position.

lpMii

[in/out] Pointer to a **MENUITEMINFO** structure that specifies the information to retrieve and receives information about the menu item.

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, use the **GetLastError** function.

Remarks

To retrieve a menu item of type MFT_STRING, first find the size of the string by setting the **dwTypeData** member of **MENUITEMINFO** to NULL and then calling **GetMenuItemInfo**. The value of **cch** is the size needed. Then, allocate a buffer of this size, place the pointer to the buffer in **dwTypeData**, and call **GetMenuItemInfo** once again to fill the buffer with the string.

If the retrieved menu item is of some other type, then **GetMenuItemInfo** sets the **dwTypeData** member to a value whose type is specified by the **fType** member and sets **cch** to 0.

Windows 2000 and Windows 98: Both **dwTypeData** and **cch** are used with MIIM_STRING.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Menus Overview, Menu Functions, **SetMenuItemInfo**, **MENUITEMINFO**

GetMenuItemRect

The **GetMenuItemRect** function retrieves the bounding rectangle for the specified menu item.

```
BOOL GetMenuItemRect(  
    HWND hWnd,           // handle to window  
    HMENU hMenu,        // handle to menu  
    UINT uItem,         // menu item position  
    LPRECT lprcItem    // bounding rectangle  
);
```

Parameters

hWnd

[in] Handle to the window containing the menu.

Windows 98 and Windows 2000: If this value is NULL and the *hMenu* parameter represents a pop-up menu, the function will find the menu window.

hMenu

[in] Handle to a menu.

uItem

[in] Zero-based position of the menu item.

lprcItem

[out] Pointer to a **RECT** structure that receives the bounding rectangle of the specified menu item expressed in screen coordinates.

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, use the **GetLastError** function.

! 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

Menus Overview, Menu Functions, **RECT**

GetMenuState

The **GetMenuState** function retrieves the menu flags associated with the specified menu item. If the menu item opens a submenu, this function also returns the number of items in the submenu.

Note The **GetMenuState** function has been superseded by the **GetMenuItemInfo** function. You still can use **GetMenuState**, however, if you do not need any of the extended features of **GetMenuItemInfo**.

```
UINT GetMenuState(
    HMENU hMenu, // handle to menu
    UINT uid,    // menu item to query
    UINT uFlags // options
);
```

Parameters

hMenu

[in] Handle to the menu that contains the menu item whose flags are to be retrieved.

uid

[in] Specifies the menu item for which the menu flags are to be retrieved, as determined by the *uFlags* parameter.

uFlags

[in] Specifies how the *uid* parameter is interpreted. This parameter can be one of the following values:

| Value | Description |
|---------------|--|
| MF_BYCOMMAND | Indicates that the <i>uid</i> parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified. |
| MF_BYPOSITION | Indicates that the <i>uid</i> parameter gives the zero-based relative position of the menu item. |

Return Values

If the specified item does not exist, the return value is -1 .

If the menu item opens a submenu, the low-order byte of the return value contains the menu flags associated with the item, and the high-order byte contains the number of items in the submenu opened by the item.

Otherwise, the return value is a mask (Boolean OR) of the menu flags. Following are the menu flags associated with the menu item:

| Value | Meaning |
|-----------------|--|
| MF_CHECKED | A check mark is placed next to the item (for drop-down menus, submenus, and shortcut menus only). |
| MF_DISABLED | The item is disabled. |
| MF_GRAYED | The item is disabled and dimmed. |
| MF_HILITE | The item is highlighted. |
| MF_MENUBARBREAK | This is the same as the MF_MENUBREAK flag, except for drop-down menus, submenus, and shortcut menus, where the new column is separated from the old column by a vertical line. |
| MF_MENUBREAK | The item is placed on a new line (for menu bars) or in a new column (for drop-down menus, submenus, and shortcut menus) without separating columns. |
| MF_OWNERDRAW | The item is owner-drawn. |
| MF_POPUP | The menu item is a submenu. |
| MF_SEPARATOR | There is a horizontal dividing line (only for drop-down menus, submenus, and shortcut menus). |

Remarks

In addition, it is possible to test an item for a flag value of MF_ENABLED, MF_STRING, MF_UNCHECKED, or MF_UNHILITE. However, since these values equate to zero, you must use an expression to test for them:

| Flag | Expression to test for the flag |
|--------------|-----------------------------------|
| MF_ENABLED | !(Flag~(MF_DISABLED MF_GRAYED |
| MF_STRING | !(Flag~(MF_BITMAP MF_OWNERDRAW) |
| MF_UNCHECKED | !(Flag~MF_CHECKED) |
| MF_UNHILITE | !(Flag~HILITE) |

! 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

Menus Overview, Menu Functions, **GetMenu**, **GetMenuItemCount**, **GetMenuItemID**, **GetMenuItemInfo**

GetMenuString

The **GetMenuString** function copies the text string of the specified menu item into the specified buffer.

Note The **GetMenuString** function has been superseded. Use the **GetMenuItemInfo** function to retrieve the menu item text.

```
int GetMenuString(
    HMENU hMenu, // handle to the menu
    UINT uIDItem, // menu item identifier
    LPTSTR lpString, // buffer for the string
    int nMaxCount, // maximum length of string
    UINT uFlag // options
);
```

Parameters

hMenu

[in] Handle to the menu.

uIDItem

[in] Specifies the menu item to be changed, as determined by the *uFlag* parameter.

lpString

[out] Pointer to the buffer that receives the null-terminated string.

If *lpString* is NULL, the function returns the length of the menu string.

nMaxCount

[in] Specifies the maximum length, in characters, of the string to be copied. If the string is longer than the maximum specified in the *nMaxCount* parameter, the extra characters are truncated.

If *nMaxCount* is 0, the function returns the length of the menu string.

uFlag

[in] Specifies how the *uIDItem* parameter is interpreted. This parameter must be one of the following values:

| Value | Meaning |
|---------------|--|
| MF_BYCOMMAND | Indicates that <i>uIDItem</i> gives the identifier of the menu item. If neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified, then MF_BYCOMMAND is the default flag. |
| MF_BYPOSITION | Indicates that <i>uIDItem</i> gives the zero-based relative position of the menu item. |

Return Values

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero.

If the specified item is not of type `MFT_STRING`, then the return value is zero.

Windows 98 and Windows 2000: `MIIM_STRING` supersedes `MFT_STRING`.

Remarks

The *nMaxCount* parameter must be one larger than the number of characters in the text string, to accommodate the terminating null character.

If *nMaxCount* is 0, the function returns the length of the menu string.

! 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

Menus Overview, Menu Functions, `GetMenuItemID`

GetSubMenu

The `GetSubMenu` function retrieves a handle to the drop-down menu or submenu activated by the specified menu item.

```
HMENU GetSubMenu(  
    HMENU hMenu, // handle to menu  
    int nPos     // menu item position  
);
```

Parameters

hMenu

[in] Handle to the menu.

nPos

[in] Specifies the zero-based relative position in the specified menu of an item that activates a drop-down menu or submenu.

Return Values

If the function succeeds, the return value is a handle to the drop-down menu or submenu activated by the menu item. If the menu item does not activate a drop-down menu or submenu, 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: Requires version 1.0 or later.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Menus Overview, Menu Functions, **CreatePopupMenu**, **GetMenu**

GetSystemMenu

The **GetSystemMenu** function allows the application to access the window menu (also known as the system menu or the control menu) for copying and modifying.

```
HMENU GetSystemMenu(  
    HWND hWnd, // handle to window  
    BOOL bRevert // reset option  
);
```

Parameters

hWnd

[in] Handle to the window that will own a copy of the window menu.

bRevert

[in] Specifies the action to be taken. If this parameter is FALSE, **GetSystemMenu** returns a handle to the copy of the window menu currently in use. The copy is initially identical to the window menu, but it can be modified.

If this parameter is TRUE, **GetSystemMenu** resets the window menu back to the default state. The previous window menu, if any, is destroyed.

Return Values

If the *bRevert* parameter is FALSE, the return value is a handle to a copy of the **window** menu. If the *bRevert* parameter is TRUE, the return value is NULL.

Remarks

Any window that does not use the **GetSystemMenu** function to make its own copy of the window menu receives the standard window menu.

The window menu initially contains items with various identifier values, such as `SC_CLOSE`, `SC_MOVE`, and `SC_SIZE`.

Menu items on the window menu send **WM_SYSCOMMAND** messages.

All predefined window menu items have identifier numbers greater than 0xF000. If an application adds commands to the window menu, it should use identifier numbers less than 0xF000.

The system automatically grays items on the standard window menu, depending on the situation. The application can perform its own checking or dimming by responding to the **WM_INITMENU** message that is sent before any menu is displayed.

! 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

Menus Overview, Menu Functions, **GetMenu**, **InsertMenuItem**, **SetMenuItemInfo**, **WM_INITMENU**, **WM_SYSCOMMAND**

HiliteMenuItem

The **HiliteMenuItem** function highlights or removes the highlighting from an item in a menu bar.

```

BOOL HiliteMenuItem(
    HWND hwnd,           // handle to window
    HMENU hmenu,        // handle to menu
    UINT uItemHilite,   // menu item
    UINT uHilite        // highlight options
);

```

Parameters

hwnd

[in] Handle to the window that contains the menu.

hmenu

[in] Handle to the menu bar that contains the item to be highlighted.

uItemHilite

[in] Specifies the menu item to be highlighted. This parameter is either the identifier of the menu item or the offset of the menu item in the menu bar, depending on the value of the *uHilite* parameter.

uHilite

[in] Controls the interpretation of the *ultemHilite* parameter, and indicates whether the menu item is highlighted. This parameter must be a combination of either MF_BYCOMMAND or MF_BYPOSITION and MF_HILITE or MF_UNHILITE.

| Value | Meaning |
|---------------|---|
| MF_BYCOMMAND | Indicates that <i>ultemHilite</i> gives the identifier of the menu item. |
| MF_BYPOSITION | Indicates that <i>ultemHilite</i> gives the zero-based relative position of the menu item. |
| MF_HILITE | Highlights the menu item. If this flag is not specified, the highlighting is removed from the item. |
| MF_UNHILITE | Removes highlighting from the menu item. |

Return Values

If the menu item is set to the specified highlight state, the return value is nonzero.

If the menu item is not set to the specified highlight state, the return value is zero.

Remarks

The MF_HILITE and MF_UNHILITE flags can be used only with the **HiliteMenuItem** function; they cannot be used with the **ModifyMenu** 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Menus Overview, Menu Functions, **ModifyMenu**

InsertMenu

The **InsertMenu** function inserts a new menu item into a menu, moving other items down the menu.

Note The **InsertMenu** function has been superseded by the **InsertMenuItem** function. You still can use **InsertMenu**, however, if you do not need any of the extended features of **InsertMenuItem**.

```

BOOL InsertMenu(
    HMENU hMenu,          // handle to menu
    UINT uPosition,      // item that new item precedes
    UINT uFlags,         // options
    UINT_PTR uIDNewItem, // identifier, menu, or submenu
    LPCTSTR lpNewItem    // menu item content
);

```

Parameters

hMenu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item before which the new menu item is to be inserted, as determined by the *uFlags* parameter.

uFlags

[in] Specifies flags that control the interpretation of the *uPosition* parameter and the content, appearance, and behavior of the new menu item. This parameter must be a combination of one of the following required values, and at least one of the values listed in the following Remarks section:

| Value | Description |
|---------------|--|
| MF_BYCOMMAND | Indicates that the <i>uPosition</i> parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified. |
| MF_BYPOSITION | Indicates that the <i>uPosition</i> parameter gives the zero-based relative position of the new menu item. If <i>uPosition</i> is -1 , the new menu item is appended to the end of the menu. |

uIDNewItem

[in] Specifies either the identifier of the new menu item or, if the *uFlags* parameter has the MF_POPUP flag set, a handle to the drop-down menu or submenu.

lpNewItem

[in] Specifies the content of the new menu item. The interpretation of *lpNewItem* depends on whether the *uFlags* parameter includes the MF_BITMAP, MF_OWNERDRAW, or MF_STRING flag, as follows:

| Value | Description |
|--------------|---|
| MF_BITMAP | Contains a bitmap handle. |
| MF_OWNERDRAW | Contains an application-supplied value that can be used to maintain additional data related to the menu item. The value is in the itemData member of the structure pointed to by the <i>lparam</i> parameter of the WM_MEASUREITEM or WM_DRAWITEM message sent when the menu item is created or its appearance is updated. |
| MF_STRING | Contains a pointer to a null-terminated string (the default). |

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 application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window.

The following list describes the flags that can be set in the *uFlags* parameter:

| Value | Description |
|-----------------|---|
| MF_BITMAP | Uses a bitmap as the menu item. The <i>lpNewItem</i> parameter contains a handle to the bitmap. |
| MF_CHECKED | Places a check mark next to the menu item. If the application provides check-mark bitmaps (see SetMenuItemBitmaps), this flag displays the check-mark bitmap next to the menu item. |
| MF_DISABLED | Disables the menu item so that it cannot be selected, but does not dim it. |
| MF_ENABLED | Enables the menu item so that it can be selected, and restores it from its dimmed state. |
| MF_GRAYED | Disables the menu item and dims it, so that it cannot be selected. |
| MF_MENUBARBREAK | Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line. |
| MF_MENUBREAK | Places the item on a new line (for menu bars) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns. |
| MF_OWNERDRAW | Specifies that the item is an owner-drawn item. Before the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message to retrieve the width and height of the menu item. Then, the WM_DRAWITEM message is sent to the window procedure of the owner window whenever the appearance of the menu item must be updated. |
| MF_POPUP | Specifies that the menu item opens a drop-down menu or submenu. The <i>uidNewItem</i> parameter specifies a handle to the drop-down menu or submenu. This flag is used to add a menu name to a menu bar or a menu item that opens a submenu to a drop-down menu, submenu, or shortcut menu. |

continued

(continued)

| Value | Description |
|--------------|--|
| MF_SEPARATOR | Draws a horizontal dividing line. This flag is used only in a drop-down menu, submenu, or shortcut menu. The line cannot be dimmed, disabled, or highlighted. The <i>lpNewItem</i> and <i>uIDNewItem</i> parameters are ignored. |
| MF_STRING | Specifies that the menu item is a text string; the <i>lpNewItem</i> parameter is a pointer to the string. |
| MF_UNCHECKED | Does not place a check mark next to the menu item (default). If the application supplies check-mark bitmaps (see the SetMenuItemBitmaps function), this flag displays the clear bitmap next to the menu item. |

The following groups of flags cannot be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, MF_OWNERDRAW, and MF_SEPARATOR
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

! 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

Menus Overview, Menu Functions, **AppendMenu**, **DeleteMenu**, **DrawMenuBar**, **InsertMenuItem**, **ModifyMenu**, **RemoveMenu**, **SetMenuItemBitmaps**, **WM_DRAWITEM**, **WM_MEASUREITEM**

InsertMenuItem

The **InsertMenuItem** function inserts a new menu item at the specified position in a menu.

```

BOOL InsertMenuItem(
    HMENU hMenu,           // handle to menu
    UINT uItem,           // identifier or position
    BOOL fByPosition,     // meaning of uItem
    LPCMENUIITEMINFO lpmi // menu item information
);

```

Parameters

hMenu

[in] Handle to the menu in which the new menu item is inserted.

uItem

[in] Identifier or position of the menu item before which to insert the new item. The meaning of this parameter depends on the value of *fByPosition*.

fByPosition

[in] Value specifying the meaning of *uItem*. If this parameter is FALSE, *uItem* is a menu item identifier. Otherwise, it is a menu item position.

lpmi

[in] Pointer to a **MENUIITEMINFO** structure that contains information about the new menu item.

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, use the **GetLastError** function.

Remarks

The application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window.

In order for keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the **WM_MENUCHAR** message. See *Owner-Drawn Menus* and the **WM_MENUCHAR Message** for more information.

! 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

Menus Overview, Menu Functions, **DrawMenuBar**, **MENUIITEMINFO**

IsMenu

The **IsMenu** function determines whether a handle is a menu handle.

```
BOOL IsMenu(  
    HMENU hMenu // handle to test  
);
```

Parameters

hMenu

[in] Handle to be tested.

Return Values

If *hMenu* is a menu handle, the return value is nonzero.

If *hMenu* is not a menu handle, 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Menus Overview, Menu Functions

LoadMenu

The **LoadMenu** function loads the specified menu resource from the executable (.exe) file associated with an application instance.

```
HMENU LoadMenu(  
    HINSTANCE hInstance, // handle to module  
    LPCTSTR lpMenuName // menu name or resource identifier  
);
```

Parameters

hInstance

[in] Handle to the module containing the menu resource to be loaded.

lpMenuName

[in] Pointer to a null-terminated string that contains the name of the menu resource. Alternatively, this parameter can consist of the resource identifier in the low-order

word and zero in the high-order word. To create this value, use the **MAKEINTRESOURCE** macro.

Return Values

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

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **DestroyMenu** function is used, before an application closes, to destroy the menu and free memory that the loaded menu occupied.

! 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

Menus Overview, Menu Functions **LoadMenuIndirect**, **MAKEINTRESOURCE**

LoadMenuIndirect

The **LoadMenuIndirect** function loads the specified menu template in memory.

```
HMENU LoadMenuIndirect(  
    CONST MENUTEMPLATE *lpMenuTemplate // menu template  
);
```

Parameters

lpMenuTemplate

[in] Pointer to a menu template or an extended menu template.

A menu template consists of a **MENITEMTEMPLATEHEADER** structure followed by one or more contiguous **MENITEMTEMPLATE** structures. An extended menu template consists of a **MENUEX_TEMPLATE_HEADER** structure followed by one or more contiguous **MENUEX_TEMPLATE_ITEM** structures.

Return Values

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

If the function fails, the return value is `NULL`. To get extended error information, call **GetLastError**.

Remarks

For both the ANSI and Unicode versions of this function, the strings in the **MENUITEMTEMPLATE** structure must be Unicode strings.

! 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

Menus Overview, Menu Functions, **LoadMenu**, **MENUEX_TEMPLATE_HEADER**, **MENUEX_TEMPLATE_ITEM**, **MENUITEMTEMPLATE**, **MENUITEMTEMPLATEHEADER**

MenuItemFromPoint

The **MenuItemFromPoint** function determines which menu item, if any, is at the specified location.

```
int MenuItemFromPoint(
    HWND hwnd,        // handle to window
    HMENU hMenu,     // handle to menu
    POINT ptScreen   // location to test
);
```

Parameters

hwnd

[in] Handle to the window containing the menu.

Windows 98 and Windows 2000: If this value is `NULL` and the *hMenu* parameter represents a pop-up menu, the function will find the menu window.

hMenu

[in] Handle to the menu containing the menu items to hit test.

ptScreen

[in] A **POINT** structure that specifies the location to test. If *hMenu* specifies a menu bar, this parameter is in window coordinates. Otherwise, it is in client coordinates.

Return Values

Returns the zero-based position of the menu item at the specified location or -1 if no menu item is at the specified location.

! 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

Menus Overview, Menu Functions, **POINT**

ModifyMenu

The **ModifyMenu** function changes an existing menu item. This function is used to specify the content, appearance, and behavior of the menu item.

Note The **ModifyMenu** function has been superseded by the **SetMenuItemInfo** function. You still can use **ModifyMenu**, however, if you do not need any of the extended features of **SetMenuItemInfo**.

```
BOOL ModifyMenu(  
    HMENU hMnu,           // handle to menu  
    UINT uPosition,      // menu item to modify  
    UINT uFlags,         // options  
    UINT_PTR uIDNewItem, // identifier, menu, or submenu  
    LPCTSTR lpNewItem    // menu item content  
);
```

Parameters

hMnu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item to be changed, as determined by the *uFlags* parameter.

uFlags

[in] Specifies flags that control the interpretation of the *uPosition* parameter and the content, appearance, and behavior of the menu item. This parameter must be a combination of one of the following required values and at least one of the values listed in the following Remarks section:

| Value | Meaning |
|---------------|--|
| MF_BYCOMMAND | Indicates that the <i>uPosition</i> parameter gives the identifier of the menu item. The MF_BYCOMMAND flag is the default if neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified. |
| MF_BYPOSITION | Indicates that the <i>uPosition</i> parameter gives the zero-based relative position of the menu item. |

ulDNewItem

[in] Specifies either the identifier of the modified menu item or, if the *uFlags* parameter has the MF_POPUP flag set, a handle to the drop-down menu or submenu.

lpNewItem

[in] Pointer to the content of the changed menu item. The interpretation of this parameter depends on whether the *uFlags* parameter includes the MF_BITMAP, MF_OWNERDRAW, or MF_STRING flag:

| Value | Meaning |
|--------------|--|
| MF_BITMAP | Contains a bitmap handle. |
| MF_OWNERDRAW | Contains a value supplied by an application that is used to maintain additional data related to the menu item. The value is in the <i>itemData</i> member of the structure pointed to by the <i>lparam</i> parameter of the WM_MEASUREITEM or WM_DRAWITEM message sent when the menu item is created or its appearance is updated. |
| MF_STRING | Contains a pointer to a null-terminated string (the default). |

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

If **ModifyMenu** replaces a menu item that opens a drop-down menu or submenu, the function destroys the old drop-down menu or submenu and frees the memory used by it.

In order for keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the **WM_MENUCHAR** message. See *Owner-Drawn Menus* and **WM_MENUCHAR** for more information.

The application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window. To change the attributes of existing menu items, it is much faster to use the **CheckMenuItem** and **EnableMenuItem** functions.

The following list describes the flags that may be set in the *uFlags* parameter:

| Value | Meaning |
|-----------------|---|
| MF_BITMAP | Uses a bitmap as the menu item. The <i>lpNewItem</i> parameter contains a handle to the bitmap. |
| MF_BYCOMMAND | Indicates that the <i>uPosition</i> parameter specifies the identifier of the menu item (the default). |
| MF_BYPOSITION | Indicates that the <i>uPosition</i> parameter specifies the zero-based relative position of the new menu item. |
| MF_CHECKED | Places a check mark next to the item. If your application provides check-mark bitmaps (see the SetMenuItemBitmaps function), this flag displays a selected bitmap next to the menu item. |
| MF_DISABLED | Disables the menu item so that it cannot be selected, but this flag does not dim it. |
| MF_ENABLED | Enables the menu item so that it can be selected, and restores it from its dimmed state. |
| MF_GRAYED | Disables the menu item and dims it, so that it cannot be selected. |
| MF_MENUBARBREAK | Functions the same as the MF_MENUBREAK flag for a menu bar. For a drop-down menu, submenu, or shortcut menu, the new column is separated from the old column by a vertical line. |
| MF_MENUBREAK | Places the item on a new line (for menu bars) or in a new column (for a drop-down menu, submenu, or shortcut menu) without separating columns. |
| MF_OWNERDRAW | Specifies that the item is an owner-drawn item. Before the menu is displayed for the first time, the window that owns the menu receives a WM_MEASUREITEM message to retrieve the width and height of the menu item. Then, the WM_DRAWITEM message is sent to the window procedure of the owner window whenever the appearance of the menu item must be updated. |
| MF_POPUP | Specifies that the menu item opens a drop-down menu or submenu. The <i>uIDNewItem</i> parameter specifies a handle to the drop-down menu or submenu. This flag is used to add a menu name to a menu bar or a menu item that opens a submenu to a drop-down menu, submenu, or shortcut menu. |
| MF_SEPARATOR | Draws a horizontal dividing line. This flag is used only in a drop-down menu, submenu, or shortcut menu. The line cannot be dimmed, disabled, or highlighted. The <i>lpNewItem</i> and <i>uIDNewItem</i> parameters are ignored. |

(continued)

(continued)

| Value | Meaning |
|--------------|--|
| MF_STRING | Specifies that the menu item is a text string; the <i>lpNewItem</i> parameter is a pointer to the string. |
| MF_UNCHECKED | Does not place a check mark next to the item (the default). If your application supplies check-mark bitmaps (see the SetMenuItemBitmaps function), this flag displays a clear bitmap next to the menu item. |

The following groups of flags cannot be used together:

- MF_BYCOMMAND and MF_BYPOSITION
- MF_DISABLED, MF_ENABLED, and MF_GRAYED
- MF_BITMAP, MF_STRING, MF_OWNERDRAW, and MF_SEPARATOR
- MF_MENUBARBREAK and MF_MENUBREAK
- MF_CHECKED and MF_UNCHECKED

! 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

Menus Overview, Menu Functions, **AppendMenu**, **CheckMenuItem**, **DrawMenuBar**, **EnableMenuItem**, **SetMenuItemBitmaps**, **SetMenuItemInfo**, **WM_DRAWITEM**, **WM_MEASUREITEM**

RemoveMenu

The **RemoveMenu** function deletes a menu item or detaches a submenu from the specified menu. If the menu item opens a drop-down menu or submenu, **RemoveMenu** does not destroy the menu or its handle, allowing the menu to be reused. Before this function is called, the **GetSubMenu** function should retrieve a handle to the drop-down menu or submenu.

```

BOOL RemoveMenu(
    HMENU hMenu,    // handle to menu
    UINT uPosition, // menu item identifier or position
    UINT uFlags     // options
);

```

Parameters

hMenu

[in] Handle to the menu to be changed.

uPosition

[in] Specifies the menu item to be deleted, as determined by the *uFlags* parameter.

uFlags

[in] Specifies how the *uPosition* parameter is interpreted. This parameter must be one of the following values:

| Value | Meaning |
|---------------|--|
| MF_BYCOMMAND | Indicates that <i>uPosition</i> gives the identifier of the menu item. If neither the MF_BYCOMMAND nor MF_BYPOSITION flag is specified, then MF_BYCOMMAND is the default flag. |
| MF_BYPOSITION | Indicates that <i>uPosition</i> gives the zero-based relative position of the menu item. |

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 application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window.

! 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

Menus Overview, Menu Functions, **CreatePopupMenu**, **DeleteMenu**, **DrawMenuBar**, **GetSubMenu**

SetMenu

The **SetMenu** function assigns a new menu to the specified window.

```

BOOL SetMenu(
    HWND hWnd, // handle to window
    HMENU hMenu // handle to menu
);

```

Parameters

hWnd

[in] Handle to the window to which the menu is to be assigned.

hMenu

[in] Handle to the new menu. If this parameter is NULL, the window's current menu is removed.

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 window is redrawn to reflect the menu change. A menu can be assigned to any window that is not a child window.

The **SetMenu** function replaces the previous menu, if any, but it does not destroy it. An application should call the **DestroyMenu** function to accomplish this task.

! 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

Menus Overview, Menu Functions, **DestroyMenu**, **GetMenu**

SetMenuDefaultItem

The **SetMenuDefaultItem** function sets the default menu item for the specified menu.

```

BOOL SetMenuDefaultItem(
    HMENU hMenu, // handle to menu
    UINT uItem, // identifier or position
    UINT fByPos // meaning of uItem
);

```

Parameters

hMenu

[in] Handle to the menu to set the default item for.

ultem

[in] Identifier or position of the new default menu item, or -1 for no default item. The meaning of this parameter depends on the value of *fByPos*.

fByPos

[in] Value specifying the meaning of *ultem*. If this parameter is FALSE, *ultem* is a menu item identifier. Otherwise, it is a menu item position.

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, use the **GetLastError** function.

! 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

Menus Overview, Menu Functions, **GetMenuDefaultItem**

SetMenuInfo

The **SetMenuInfo** function sets information for a specified menu.

```
BOOL SetMenuInfo(  
    HMENU hmenu,           // handle to menu  
    LPCMENUINFO lpcmi     // menu information  
);
```

Parameters

hmenu

[in] Handle to a menu.

lpcmi

[in] Pointer to a **MENUINFO** structure for the menu.

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**.

! 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.

+ See Also

Menus Overview, Menu Functions

SetMenuItemBitmaps

The **SetMenuItemBitmaps** function associates the specified bitmap with a menu item. Whether the menu item is selected or clear, the system displays the appropriate bitmap next to the menu item.

```

BOOL SetMenuItemBitmaps(
    HMENU hMenu,           // handle to menu
    UINT uPosition,       // menu item
    UINT uFlags,           // options
    HBITMAP hBitmapUnchecked, // handle to unchecked bitmap
    HBITMAP hBitmapChecked  // handle to checked bitmap
);

```

Parameters

hMenu

[in] Handle to the menu containing the item to receive new check-mark bitmaps.

uPosition

[in] Specifies the menu item to be changed, as determined by the *uFlags* parameter.

uFlags

[in] Specifies how the *uPosition* parameter is interpreted. The *uFlags* parameter must be one of the following values:

| Value | Meaning |
|--------------|--|
| MF_BYCOMMAND | Indicates that <i>uPosition</i> gives the identifier of the menu item. If neither MF_BYCOMMAND nor MF_BYPOSITION is specified, MF_BYCOMMAND is the default flag. |

| Value | Meaning |
|---------------|--|
| MF_BYPOSITION | Indicates that <i>uPosition</i> gives the zero-based relative position of the menu item. |

hBitmapUnchecked

[in] Handle to the bitmap displayed when the menu item is not selected.

hBitmapChecked

[in] Handle to the bitmap displayed when the menu item is selected.

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

If either the *hBitmapUnchecked* or *hBitmapChecked* parameter is NULL, the system displays nothing next to the menu item for the corresponding check state. If both parameters are NULL, the system displays the default check-mark bitmap when the item is selected, and removes the bitmap when the item is not selected.

When the menu is destroyed, these bitmaps are not destroyed; it is up to the application to destroy them.

The selected and clear bitmaps should be monochrome. The system uses the Boolean AND operator to combine bitmaps with the menu, so that the white part becomes transparent and the black part becomes the menu-item color. If you use color bitmaps, the results might be undesirable.

Use the **GetSystemMetrics** function with the CXMENUCHECK and CYMENUCHECK values to retrieve the bitmap dimensions.

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

Menus Overview, Menu Functions, **GetSystemMetrics**

SetMenuItemInfo

The **SetMenuItemInfo** function changes information about a menu item.

```
BOOL SetMenuItemInfo(  
    HMENU hMenu,           // handle to menu  
    UINT uItem,           // identifier or position  
    BOOL fByPosition,     // meaning of uItem  
    LPMENITEMINFO lpmi) // menu item information  
);
```

Parameters

hMenu

[in] Handle to the menu that contains the menu item.

uItem

[in] Identifier or position of the menu item to change. The meaning of this parameter depends on the value of *fByPosition*.

fByPosition

[in] Value specifying the meaning of *uItem*. If this parameter is FALSE, *uItem* is a menu item identifier. Otherwise, it is a menu item position.

lpmi

[in] Pointer to a **MENITEMINFO** structure that contains information about the menu item and specifies which menu item attributes to change.

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, use the **GetLastError** function.

Remarks

The application must call the **DrawMenuBar** function whenever a menu changes, whether or not the menu is in a displayed window.

In order for keyboard accelerators to work with bitmap or owner-drawn menu items, the owner of the menu must process the **WM_MENUCHAR** message. See Owner-Drawn Menus and the **WM_MENUCHAR** Message for more information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Menus Overview, Menu Functions, **DrawMenuBar**, **GetMenuItemInfo**, **MENUITEMINFO**

TrackPopupMenu

The **TrackPopupMenu** function displays a shortcut menu at the specified location and tracks the selection of items on the menu. The shortcut menu can appear anywhere on the screen.

To specify an area of the screen the menu should not overlap, use the **TrackPopupMenuEx** function.

```

BOOL TrackPopupMenu(
    HMENU hMenu,           // handle to shortcut menu
    UINT uFlags,          // options
    int x,                // horizontal position
    int y,                // vertical position
    int nReserved,       // reserved, must be zero
    HWND hWnd,           // handle to owner window
    CONST RECT *prcRect // ignored
);

```

Parameters

hMenu

[in] Handle to the shortcut menu to be displayed. The handle can be obtained by calling **CreatePopupMenu** to create a new shortcut menu, or by calling **GetSubMenu** to retrieve a handle to a submenu associated with an existing menu item.

uFlags

[in] Use zero or more of these flags to specify function options.

Use one of the following flags to specify how the function positions the shortcut menu horizontally:

| Value | Meaning |
|-----------------|---|
| TPM_CENTERALIGN | If this flag is set, the function centers the shortcut menu horizontally relative to the coordinate specified by the <i>x</i> parameter. |
| TPM_LEFTALIGN | If this flag is set, the function positions the shortcut menu so that its left side is aligned with the coordinate specified by the <i>x</i> parameter. |
| TPM_RIGHTALIGN | Positions the shortcut menu so that its right side is aligned with the coordinate specified by the <i>x</i> parameter. |

Use one of the following flags to specify how the function positions the shortcut menu vertically:

| Value | Meaning |
|------------------|---|
| TPM_BOTTOMALIGN | If this flag is set, the function positions the shortcut menu so that its bottom side is aligned with the coordinate specified by the <i>y</i> parameter. |
| TPM_TOPALIGN | If this flag is set, the function positions the shortcut menu so that its top side is aligned with the coordinate specified by the <i>y</i> parameter. |
| TPM_VCENTERALIGN | If this flag is set, the function centers the shortcut menu vertically relative to the coordinate specified by the <i>y</i> parameter. |

Use the following flags to determine the user selection without having to set up a parent window for the menu:

| Value | Meaning |
|---------------|---|
| TPM_NONOTIFY | If this flag is set, the function does not send notification messages when the user clicks on a menu item. |
| TPM_RETURNCMD | If this flag is set, the function returns the menu item identifier of the user's selection in the return value. |

Use one of the following flags to specify which mouse button the shortcut menu tracks:

| Value | Meaning |
|-----------------|--|
| TPM_LEFTBUTTON | If this flag is set, the user can select menu items with only the left mouse button. |
| TPM_RIGHTBUTTON | If this flag is set, the user can select menu items with <i>both</i> the left and right mouse buttons. |

Windows 98, Windows 2000: Use one of the following flags to modify the animation of a menu:

| Value | Meaning |
|---------------------|---------------------------------------|
| TPM_HORNEGANIMATION | Animates the menu from left to right. |
| TPM_HORPOSANIMATION | Animates the menu from right to left. |
| TPM_NOANIMATION | Displays menu without animation. |
| TPM_VERNEGANIMATION | Animates the menu from bottom to top. |
| TPM_VERPOSANIMATION | Animates the menu from top to bottom. |

For any animation to occur, the **SystemParametersInfo** function must set `SPI_SETMENUANIMATION`. Also, all the `TPM_*ANIMATION` flags, except for `TPM_NOANIMATION`, are ignored if menu fade animation is on. See the `SPI_GETMENUFADE` flag in **SystemParametersInfo**.

Windows 98, Windows 2000: Use the `TPM_RECURSE` flag to display a menu when another menu is already displayed. This is intended to support context menus within a menu.

x

[in] Specifies the horizontal location of the shortcut menu, in screen coordinates.

y

[in] Specifies the vertical location of the shortcut menu, in screen coordinates.

nReserved

Reserved; must be zero.

hWnd

[in] Handle to the window that owns the shortcut menu. This window receives all messages from the menu. The window does not receive a **WM_COMMAND** message from the menu until the function returns.

If you specify `TPM_NONOTIFY` in the *uFlags* parameter, the function does not send messages to the window identified by *hWnd*. However, you still must pass a window handle in *hWnd*. It can be any window handle from your application.

prcRect

Ignored.

Return Values

If you specify `TPM_RETURNCMD` in the *uFlags* parameter, the return value is the menu-item identifier of the item that the user selected. If the user cancels the menu without making a selection, or if an error occurs, then the return value is zero.

If you do not specify `TPM_RETURNCMD` in the *uFlags* parameter, the return value is nonzero if the function succeeds and zero if it fails. 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

Menus Overview, Menu Functions, **CreatePopupMenu**, **GetSubMenu**, **RECT**, **SystemParametersInfo**, **TrackPopupMenuEx**, **WM_COMMAND**

TrackPopupMenuEx

The **TrackPopupMenuEx** function displays a shortcut menu at the specified location and tracks the selection of items on the shortcut menu. The shortcut menu can appear anywhere on the screen.

```

BOOL TrackPopupMenuEx(
    HMENU hmenu,           // handle to shortcut menu
    UINT fuFlags,         // options
    int x,                // horizontal position
    int y,                // vertical position
    HWND hwnd,           // handle to window
    LPTMPARAMS lptpm // area not to overlap
);

```

Parameters

hmenu

[in] Handle to the shortcut menu to be displayed. This handle can be obtained by calling the **CreatePopupMenu** function to create a new shortcut menu, or by calling the **GetSubMenu** function to retrieve a handle to a submenu associated with an existing menu item.

fuFlags

[in] Specifies function options.

Use one of the following flags to specify how the function positions the shortcut menu horizontally:

| Value | Meaning |
|-----------------|---|
| TPM_CENTERALIGN | If this flag is set, the function centers the shortcut menu horizontally relative to the coordinate specified by the <i>x</i> parameter. |
| TPM_LEFTALIGN | If this flag is set, the function positions the shortcut menu so that its left side is aligned with the coordinate specified by the <i>x</i> parameter. |
| TPM_RIGHTALIGN | Positions the shortcut menu so that its right side is aligned with the coordinate specified by the <i>x</i> parameter. |

Use one of the following flags to specify how the function positions the shortcut menu vertically:

| Value | Meaning |
|-----------------|---|
| TPM_BOTTOMALIGN | If this flag is set, the function positions the shortcut menu so that its bottom side is aligned with the coordinate specified by the <i>y</i> parameter. |

| Value | Meaning |
|------------------|--|
| TPM_TOPALIGN | If this flag is set, the function positions the shortcut menu so that its top side is aligned with the coordinate specified by the <i>y</i> parameter. |
| TPM_VCENTERALIGN | If this flag is set, the function centers the shortcut menu vertically relative to the coordinate specified by the <i>y</i> parameter. |

Use the following flags to determine the user selection without having to set up a parent window for the menu:

| Value | Meaning |
|---------------|---|
| TPM_NONOTIFY | If this flag is set, the function does not send notification messages when the user clicks on a menu item. |
| TPM_RETURNCMD | If this flag is set, the function returns the menu item identifier of the user's selection in the return value. |

Use one of the following flags to specify which mouse button the shortcut menu tracks:

| Value | Meaning |
|-----------------|--|
| TPM_LEFTBUTTON | If this flag is set, the user can select menu items with only the left mouse button. |
| TPM_RIGHTBUTTON | If this flag is set, the user can select menu items with <i>both</i> the left and right mouse buttons. |

Windows 98, Windows 2000: Use one of the following flags to modify the animation of a menu:

| Value | Meaning |
|---------------------|---------------------------------------|
| TPM_HORNEGANIMATION | Animates the menu from left to right. |
| TPM_HORPOSANIMATION | Animates the menu from right to left. |
| TPM_NOANIMATION | Displays menu without animation. |
| TPM_VERNEGANIMATION | Animates the menu from bottom to top. |
| TPM_VERPOSANIMATION | Animates the menu from top to bottom. |

For any animation to occur, the **SystemParametersInfo** function must set SPI_SETMENUANIMATION. Also, all the TPM_*ANIMATION flags, except for TPM_NOANIMATION, are ignored if menu fade animation is on. See the SPI_GETMENUFADE flag in **SystemParametersInfo**.

Windows 98, Windows 2000: Use the TPM_RECURSE flag to display a menu when another menu is already displayed. This is intended to support context menus within a menu.

Use one of the following flags to specify whether to accommodate horizontal or vertical alignment:

| Value | Meaning |
|----------------|---|
| TPM_HORIZONTAL | If the menu cannot be shown at the specified location without overlapping the excluded rectangle, the system tries to accommodate the requested horizontal alignment before the requested vertical alignment. |
| TPM_VERTICAL | If the menu cannot be shown at the specified location without overlapping the excluded rectangle, the system tries to accommodate the requested vertical alignment before the requested horizontal alignment. |

The excluded rectangle is a portion of the screen that the menu should not overlap; it is specified by *lptpm*.

x

[in] Horizontal location of the shortcut menu, in screen coordinates.

y

[in] Vertical location of the shortcut menu, in screen coordinates.

hwnd

[in] Handle to the window that owns the shortcut menu. This window receives all messages from the menu. The window does not receive a **WM_COMMAND** message from the menu until the function returns.

If you specify TPM_NONOTIFY in the *fuFlags* parameter, the function does not send messages to the window identified by *hwnd*. However, you still have to pass a window handle in *hwnd*. It can be any window handle from your application.

lptpm

[in] Pointer to a **TPMPARAMS** structure that specifies an area of the screen the menu should not overlap. This parameter can be NULL.

Return Values

If you specify TPM_RETURNCMD in the *fuFlags* parameter, the return value is the menu-item identifier of the item that the user selected. If the user cancels the menu without making a selection, or if an error occurs, then the return value is zero.

If you do not specify TPM_RETURNCMD in the *fuFlags* parameter, the return value is nonzero if the function succeeds and zero if it fails. 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: Requires version 1.0 or later.

Header: Declared in winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Menus Overview, Menu Functions, **CreatePopupMenu**, **GetSubMenu**, **SystemParametersInfo**, **TPMPARAMS**, **WM_COMMAND**

Menu Structures

MDINEXTMENU

The **MDINEXTMENU** structure contains information about the menu to be activated.

```
typedef struct tagMDINEXTMENU {
    HMENU hmenuIn;
    HMENU hmenuNext;
    HWND hwndNext;
} MDINEXTMENU, *PMDINEXTMENU;
```

Members

hmenuIn

Receives a handle to the current menu.

hmenuNext

Specifies a handle to the menu to be activated.

hwndNext

Specifies a handle to the window to receive the menu notification messages.

! 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

Menus Overview, Menu Structures, **WM_NEXTMENU**

MENUBARINFO

The **MENUBARINFO** structure contains menu bar information.

```
typedef struct tagMENUBARINFO {
    DWORD cbSize;
    RECT rcBar;
```

```

HMENU hMenu;
HWND hwndMenu;
BOOL fBarFocused;
BOOL fFocused;
} MENUBARINFO, *PMENUBARINFO;

```

Members

cbSize

Specifies the size, in bytes, of the structure.

rcBar

Pointer to a **RECT** structure that specifies the coordinates of the menu bar, pop-up menu, or menu item.

hMenu

Handle to the menu bar or pop-up menu.

hwndMenu

Handle to the submenu.

fBarFocused

If the menu bar or pop-up menu has the focus, this parameter is TRUE. Otherwise, the parameter is FALSE.

fFocused

If the menu item has the focus, this parameter is TRUE. Otherwise, the parameter is FALSE.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Menus Overview, Menu Structures, **GetMenuBarInfo**, **RECT**

MENUEX_TEMPLATE_HEADER

The **MENUEX_TEMPLATE_HEADER** structure defines the header for an extended menu template. This structure definition is for explanation only; it is not present in any standard header file.

```

typedef struct {
    WORD wVersion;
    WORD wOffset;
    DWORD dwHelpId;
} MENUEX_TEMPLATE_HEADER;

```

Members

wVersion

Template version number. This member must be 1 for extended menu templates.

wOffset

Offset of the first **MENUEX_TEMPLATE_ITEM** structure, relative to the end of this structure member. If the first item definition immediately follows the **dwHelpId** member, this member should be 4.

dwHelpId

Help identifier of menu bar.

Remarks

An extended menu template consists of a **MENUEX_TEMPLATE_HEADER** structure followed by one or more contiguous **MENUEX_TEMPLATE_ITEM** structures. The **MENUEX_TEMPLATE_ITEM** structures, which are variable in length, are aligned on **DWORD** boundaries. To create a menu from an extended menu template in memory, use the **LoadMenuIndirect** function.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.

+ See Also

Menus Overview, Menu Structures, **LoadMenuIndirect**, **MENUEX_TEMPLATE_ITEM**

MENUEX_TEMPLATE_ITEM

The **MENUEX_TEMPLATE_ITEM** structure defines a menu item in an extended menu template. This structure definition is for explanation only; it is not present in any standard header file.

```
typedef struct {
    DWORD dwType;
    DWORD dwState;
    UINT  uId;
    WORD  bResInfo;
    WCHAR szText[1];
    DWORD dwHelpId;
} MENUEX_TEMPLATE_ITEM;
```

Members

dwType

Menu item type. This member can be a combination of the type (beginning with MFT) values listed with the **MENUITEMINFO** structure.

dwState

Menu item state. This member can be a combination of the state (beginning with MFS) values listed with the **MENUITEMINFO** structure.

uld

Menu item identifier. This is an application-defined 16-bit value used as a handle to the menu item. In an extended menu resource, items that open drop-down menus or submenus, as well as command items, can have identifiers.

bResInfo

Value specifying whether the menu item is the last item in the menu bar, drop-down menu, submenu, or shortcut menu and whether it is an item that opens a drop-down menu or submenu. This member can be zero or more of these values:

| Value | Meaning |
|-------|---|
| 0x01 | The structure defines an item that opens a drop-down menu or submenu. Subsequent structures define menu items in the corresponding drop-down menu or submenu. |
| 0x80 | The structure defines the last menu item in the menu bar, drop-down menu, submenu, or shortcut menu. |

For 32-bit applications, this member is a word; for 16-bit applications, it is a byte.

szText

Menu item text. This member, which is a null-terminated Unicode string, is aligned on a word boundary. The size of the menu item definition varies depending on the length of this string.

dwHelpId

Help identifier for a drop-down menu or submenu. This member, which is included only for items that open drop-down menus or submenus, is located at the first **DWORD** boundary following the variable-length **szText** member.

Remarks

An extended menu template consists of a **MENUEX_TEMPLATE_HEADER** structure followed by one or more contiguous **MENUEX_TEMPLATE_ITEM** structures. The **MENUEX_TEMPLATE_ITEM** structures, which are variable in length, are aligned on **DWORD** boundaries. To create a menu from an extended menu template in memory, use the **LoadMenuIndirect** function.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.

+ See Also

Menus Overview, Menu Structures, **LoadMenuIndirect**, **MENUEX_TEMPLATE_HEADER**, **MENUITEMINFO**

MENUGETOBJECTINFO

The **MENUGETOBJECTINFO** structure contains information about the menu that the mouse cursor is on.

```
typedef struct tagMENUGETOBJECTINFO {
    DWORD   dwFlags;
    UINT    uPos;
    HMENU   hmenu;
    PVOID   riid;
    PVOID   pvObj;
} MENUGETOBJECTINFO, *PMENUGETOBJECTINFO;
```

Members

dwFlags

Position of the mouse cursor with respect to the item indicated by **uPos**. It can be one of the following values:

| Value | Meaning |
|-----------------|---|
| MNGOF_BOTTOMGAP | Mouse is on the bottom of the item indicated by uPos . |
| MNGOF_TOPGAP | Mouse is on the top of the item indicated by uPos . |

uPos

Position of the item the mouse cursor is on.

hmenu

Handle to the menu the mouse cursor is on.

riid

Identifier of the requested interface. Currently, it can be only **IDropTarget**.

pvObj

Pointer to the interface corresponding to the **riid** member. This pointer is to be returned by the application when processing the message.

Remarks

The **MENUGETOBJECTINFO** structure is used only in drag-and-drop menus. When the **WM_MENUGETOBJECT** message is sent, *lParam* is a pointer to this structure.

To create a drag-and-drop menu, call **SetMenuInfo** with **MNS_DRAGDROP** set.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`.

+ See Also

Menus Overview, Menu Structures, **SetMenuInfo**

MENUINFO

The **MENUINFO** structure contains information about a menu.

```
typedef struct tagMENUINFO {
    DWORD    cbSize;
    DWORD    fMask;
    DWORD    dwStyle;
    UINT     cyMax;
    HBRUSH   hbrBack;
    DWORD    dwContextHelpID;
    ULONG_PTR dwMenuData;
} MENUINFO, FAR *LPMENUINFO;
typedef MENUINFO CONST FAR *LPCMENUINFO;
```

Members

cbSize

Size of the structure, in bytes

fMask

Members to retrieve or set (except for **MIM_APPLYTOSUBMENUS**). This member can be one or more of the following values:

| Value | Meaning |
|----------------------------|---|
| MIM_APPLYTOSUBMENUS | Settings apply to the menu and all of its submenus. SetMenuInfo uses this flag and GetMenuInfo ignores this flag. |
| MIM_BACKGROUND | Retrieves or sets the hbrBack member. |
| MIM_HELPID | Retrieves or sets the dwContextHelpID member. |

| Value | Meaning |
|---------------|---|
| MIM_MAXHEIGHT | Retrieves or sets the cyMax member. |
| MIM_MENUDATA | Retrieves or sets the dwMenuData member. |
| MIM_STYLE | Retrieves or sets the dwStyle member. |

dwStyle

Style of the menu. It can be one or more of the following values:

| Value | Meaning |
|-----------------|---|
| MNS_AUTODISMISS | Menu automatically ends when mouse is outside the menu for approximately 10 seconds. |
| MNS_CHECKORBMP | The same space is reserved for the check mark and the bitmap. If the check mark is drawn, the bitmap is not. All check marks and bitmaps are aligned. Used for menus where some items use check marks and some use bitmaps. |
| MNS_DRAGDROP | Menu items are OLE drop targets or drag sources. Menu owner receives WM_MENUDRAG and WM_MENUGETOBJECT messages. |
| MNS_MODELESS | Menu is modeless; that is, there is no menu modal message loop while the menu is active. |
| MNS_NOCHECK | No space is reserved to the left of an item for a check mark. The item still can be selected, but the check mark will not appear next to the item. |
| MNS_NOTIFYBYPOS | Menu owner receives a WM_MENUCOMMAND message, instead of a WM_COMMAND message, when the user makes a selection. |

cyMax

Maximum height of the menu in pixels. When the menu items exceed the space available, scrollbars are automatically used. The default (0) is the screen height.

hbrBack

Brush to use for the menu's background.

dwContextHelpID

The context help identifier. This is the same value used in **GetMenuContextHelpId** and **SetMenuContextHelpId**.

dwMenuData

An application-defined value.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h.

 See Also

Menus Overview, Menu Structures

MENUITEMINFO

The **MENUITEMINFO** structure contains information about a menu item.

```
typedef struct tagMENUITEMINFO {
    UINT    cbSize;
    UINT    fMask;
    UINT    fType;
    UINT    fState;
    UINT    wID;
    HMENU   hSubMenu;
    HBITMAP hbmpChecked;
    HBITMAP hbmpUnchecked;
    ULONG_PTR dwItemData;
    LPTSTR  dwTypeData;
    UINT    cch;
    HBITMAP hbmpItem;
} MENUITEMINFO, *LPMENUITEMINFO;
```

Members

cbSize

Size of structure, in bytes.

fMask

Members to retrieve or set. This member can be one or more of these values:

| Value | Meaning |
|-----------------|--|
| MIIM_BITMAP | Windows 98, Windows 2000: Retrieves or sets the hbmpItem member. |
| MIIM_CHECKMARKS | Retrieves or sets the hbmpChecked and hbmpUnchecked members. |
| MIIM_DATA | Retrieves or sets the dwItemData member. |
| MIIM_FTYPE | Windows 98, Windows 2000: Retrieves or sets the fType member. |
| MIIM_ID | Retrieves or sets the wID member. |
| MIIM_STATE | Retrieves or sets the fState member. |
| MIIM_STRING | Windows 98, Windows 2000: Retrieves or sets the dwTypeData member. |
| MIIM_SUBMENU | Retrieves or sets the hSubMenu member. |

| Value | Meaning |
|-----------|--|
| MIIM_TYPE | Retrieves or sets the fType and dwTypeData members. Windows 98, Windows 2000: MIIM_TYPE is replaced by MIIM_BITMAP, MIIM_FTYPE, and MIIM_STRING. |

fType

Menu item type. This member can be one or more of these values:

| Value | Meaning |
|------------------|--|
| MFT_BITMAP | Displays the menu item using a bitmap. The low-order word of the dwTypeData member is the bitmap handle, and the cch member is ignored. Windows 98, Windows 2000: MFT_BITMAP is replaced by MIIM_BITMAP and hbmpltem . |
| MFT_MENUBARBREAK | Places the menu item on a new line (for a menu bar) or in a new column (for a drop-down menu, submenu, or shortcut menu). For a drop-down menu, submenu, or shortcut menu, a vertical line separates the new column from the old. |
| MFT_MENUBREAK | Places the menu item on a new line (for a menu bar) or in a new column (for a drop-down menu, submenu, or shortcut menu). For a drop-down menu, submenu, or shortcut menu, the columns are not separated by a vertical line. |
| MFT_OWNERDRAW | Assigns responsibility for drawing the menu item to the window that owns the menu. The window receives a WM_MEASUREITEM message before the menu is displayed for the first time, and a WM_DRAWITEM message whenever the appearance of the menu item must be updated. If this value is specified, the dwTypeData member contains an application-defined value. |
| MFT_RADIOCHECK | Displays selected menu items using a radio-button mark instead of a check mark if the hbmpChecked member is NULL. |
| MFT_RIGHTJUSTIFY | Right-justifies the menu item and any subsequent items. This value is valid only if the menu item is in a menu bar. |
| MFT_RIGHTORDER | Windows 95/98, Windows 2000: Specifies that menus cascade right-to-left (the default is left-to-right). This is used to support right-to-left languages, such as Arabic and Hebrew. |

(continued)

(continued)

| Value | Meaning |
|---------------|--|
| MFT_SEPARATOR | Specifies that the menu item is a separator. A menu item separator appears as a horizontal dividing line. The dwTypeData and cch members are ignored. This value is valid only in a drop-down menu, submenu, or shortcut menu. |
| MFT_STRING | Displays the menu item using a text string. The dwTypeData member is the pointer to a null-terminated string, and the cch member is the length of the string. Windows 98, Windows 2000: MFT_STRING is replaced by MIIM_STRING. |

The MFT_BITMAP, MFT_SEPARATOR, and MFT_STRING values cannot be combined with one another. Set **fMask** to MIIM_TYPE to use **fType**.

Windows 98 and Windows 2000: **fType** is used only if **fMask** has a value of MIIM_FTYPE.

fState

Menu item state. This member can be one or more of these values:

| Value | Meaning |
|---------------|---|
| MFS_CHECKED | Checks the menu item. For more information about selected menu items, see the hbmpChecked member. |
| MFS_DEFAULT | Specifies that the menu item is the default. A menu can contain only one default menu item, which is displayed in bold. |
| MFS_DISABLED | Disables the menu item and dims it, so that it cannot be selected. This is equivalent to MFS_GRAYED. |
| MFS_ENABLED | Enables the menu item, so that it can be selected. This is the default state. |
| MFS_GRAYED | Disables the menu item and dims it, so that it cannot be selected. This is equivalent to MFS_DISABLED. |
| MFS_HILITE | Highlights the menu item. |
| MFS_UNCHECKED | Unchecks the menu item. For more information about clear menu items, see hbmpUnchecked . |
| MFS_UNHILITE | Removes the highlight from the menu item. This is the default state. |

Set **fMask** to MIIM_STATE to use **fState**.

wID

Application-defined 16-bit value that identifies the menu item. Set **fMask** to MIIM_ID to use **wID**.

hSubMenu

Handle to the drop-down menu or submenu associated with the menu item. If the menu item is not an item that opens a drop-down menu or submenu, this member is NULL. Set **fMask** to MIIM_SUBMENU to use **hSubMenu**.

hbmpChecked

Handle to the bitmap to display next to the item, if it is selected. If this member is NULL, a default bitmap is used. If the MFT_RADIOCHECK type value is specified, the default bitmap is a bullet. Otherwise, it is a check mark. Set **fMask** to MIIM_CHECKMARKS to use **hbmpChecked**.

hbmpUnchecked

Handle to the bitmap to display next to the item if it is not selected. If this member is NULL, no bitmap is used. Set **fMask** to MIIM_CHECKMARKS to use **hbmpUnchecked**.

dwItemData

Application-defined value associated with the menu item. Set **fMask** to MIIM_DATA to use **dwItemData**.

dwTypeData

Content of the menu item. The meaning of this member depends on the value of **fType** and is used only if the MIIM_TYPE flag is set in the **fMask** member.

To retrieve a menu item of type MFT_STRING, first find the size of the string by setting the **dwTypeData** member of **MENUITEMINFO** to NULL and then calling **GetMenuItemInfo**. The value of **cch** is the size needed. Then, allocate a buffer of this size, place the pointer to the buffer in **dwTypeData**, and call **GetMenuItemInfo** once again to fill the buffer with the string. If the retrieved menu item is of some other type, then **GetMenuItemInfo** sets the **dwTypeData** member to a value whose type is specified by the **fType** member.

When using with the **SetMenuItemInfo** function, this member should contain a value whose type is specified by the **fType** member.

Windows 98 and Windows 2000: **dwTypeData** is used only if the MIIM_STRING flag is set in the **fMask** member.

cch

Length of the menu item text when information is received about a menu item of the MFT_STRING type. This member is used only if the MIIM_TYPE flag is set in the **fMask** member and is zero, otherwise. This member is ignored when the content of a menu item is set by calling **SetMenuItemInfo**.

Before calling **GetMenuItemInfo**, the application must set this member to the length of the buffer pointed to by the **dwTypeData** member. If the retrieved menu item is of type MFT_STRING (as indicated by the **fType** member), then **GetMenuItemInfo** sets **cch** to the length of the retrieved string. If the retrieved menu item is of some other type, then **GetMenuItemInfo** sets the **cch** field to zero.

Windows 98, Windows 2000: **cch** is used when the MIIM_STRING flag is set in the **fMask** member.

hbmpltem

Windows 98, Windows 2000: Handle to the bitmap to be displayed, or it can be one of the values in the following table. It is used when the `MIIM_BITMAP` flag is set in the `fMask` member:

| Value | Bitmap to be displayed |
|--------------------------------------|--|
| <code>HBMMENU_CALLBACK</code> | A bitmap that is drawn by the window that owns the menu. The application must process the <code>WM_MEASUREITEM</code> and <code>WM_DRAWITEM</code> messages. |
| <code>HBMMENU_MBAR_CLOSE</code> | Close button for the menu bar. |
| <code>HBMMENU_MBAR_CLOSE_D</code> | Disabled close button for the menu bar. |
| <code>HBMMENU_MBAR_MINIMIZE</code> | Minimize button for the menu bar. |
| <code>HBMMENU_MBAR_MINIMIZE_D</code> | Disabled minimize button for the menu bar. |
| <code>HBMMENU_MBAR_RESTORE</code> | Restore button for the menu bar. |
| <code>HBMMENU_POPUP_CLOSE</code> | Close button for the submenu. |
| <code>HBMMENU_POPUP_MAXIMIZE</code> | Maximize button for the submenu. |
| <code>HBMMENU_POPUP_MINIMIZE</code> | Minimize button for the submenu. |
| <code>HBMMENU_POPUP_RESTORE</code> | Restore button for the submenu. |
| <code>HBMMENU_SYSTEM</code> | Windows icon, or the icon of the window specified in <code>dwItemData</code> . |

Remarks

A menu can display items using either text or bitmaps, but not both.

The `MENUITEMINFO` structure is used with the `GetMenuItemInfo`, `InsertMenuItem`, and `SetMenuItemInfo` functions.

Windows 98 and Windows 2000: The menu can display items using text, bitmaps, or both.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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`.

Unicode: Declared as Unicode and ANSI structures.

+ See Also

Menus Overview, Menu Structures, `GetMenuItemInfo`, `InsertMenuItem`, `SetMenuItemInfo`, `WM_DRAWITEM`, `WM_MEASUREITEM`

MENUITEMTEMPLATE

The **MENUITEMTEMPLATE** structure defines a menu item in a menu template.

```
typedef struct {
    WORD mtOption;
    WORD mtID;
    WCHAR mtString[1];
} MENUITEMTEMPLATE, *PMENUITEMTEMPLATE;
```

Members

mtOption

Specifies one or more of the following predefined menu options that control the appearance of the menu item as shown in the following table:

| Value | Meaning |
|-----------------|---|
| MF_CHECKED | Indicates that the menu item has a check mark next to it. |
| MF_GRAYED | Indicates that the menu item is initially inactive and drawn with a shaded effect. |
| MF_HELP | Indicates that the menu item has a vertical separator to its left. |
| MF_MENUBARBREAK | Indicates that the menu item is placed in a new column. The old and new columns are separated by a bar. |
| MF_MENUBREAK | Indicates that the menu item is placed in a new column. |
| MF_OWNERDRAW | Indicates that the owner window of the menu is responsible for drawing all visual aspects of the menu item, including highlighted, selected, and inactive states. This option is not valid for an item in a menu bar. |
| MF_POPUP | Indicates that the item is one that opens a drop-down menu or submenu. |

mtID

Specifies the menu item identifier of a command item; a command item sends a command message to its owner window. The **MENUITEMTEMPLATE** structure for an item that opens a drop-down menu or submenu does not contain the **mtID** member.

mtString

Specifies the null-terminated string for the menu item.

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

Menus Overview, Menu Structures, **LoadMenuIndirect**, **MENUITEMTEMPLATEHEADER**

MENUITEMTEMPLATEHEADER

The **MENUITEMTEMPLATEHEADER** structure defines the header for a menu template. A complete menu template consists of a header and one or more menu item lists.

```
typedef struct {  
    WORD versionNumber;  
    WORD offset;  
} MENUITEMTEMPLATEHEADER, *PMENUITEMTEMPLATEHEADER;
```

Members

versionNumber

Specifies the version number. This member must be zero.

offset

Specifies the offset, in bytes, from the end of the header. The menu item list begins at this offset. Usually, this member is zero, and the menu item list follows immediately after the header.

Remarks

One or more **MENUITEMTEMPLATE** structures are combined to form the menu item list.

! 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

Menus Overview, Menu Structures, **LoadMenuIndirect**, **MENUITEMTEMPLATE**

TPMPARAMS

The **TPMPARAMS** structure contains extended parameters for the **TrackPopupMenuEx** function.

```
typedef struct tagTPMPARAMS {  
    UINT cbSize;
```

```
    RECT rcExclude;
} TPMPARAMS, *LTPMPARAMS;
```

Members

cbSize

Size of structure, in bytes.

rcExclude

Rectangle to exclude when positioning the window, in screen 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`.

+ See Also

Menus Overview, Menu Structures, **Rectangle**, **TrackPopupMenuEx**

Menu Messages

WM_COMMAND

The **WM_COMMAND** message is sent when the user selects a command item from a menu; a control sends a notification message to its parent window; or an accelerator keystroke is translated.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_COMMAND
    WPARAM wParam,     // notification code and identifier
    LPARAM lParam       // handle to control (HWND)
);
```

Parameters

wParam

The high-order word specifies the notification code if the message is from a control. If the message is from an accelerator, this value is 1. If the message is from a menu, this value is zero.

The low-order word specifies the identifier of the menu item, control, or accelerator.

lParam

Handle to the control sending the message if the message is from a control. Otherwise, this parameter is NULL.

Return Values

If an application processes this message, it should return zero.

Remarks

Accelerator keystrokes that select items from the window menu are translated into **WM_SYSCOMMAND** messages.

If an accelerator keystroke occurs that corresponds to a menu item when the window that owns the menu is minimized, no **WM_COMMAND** message is sent. However, if an accelerator keystroke occurs that does not match any of the items in the window's menu or in the window menu, a **WM_COMMAND** message is sent, even if the window is minimized.

If an application enables a menu separator, the system sends a **WM_COMMAND** message with the low-word of the *wParam* parameter set to zero when the user selects the separator.

Windows 98, Windows 2000: If a menu is defined with a **MENUINFO.dwStyle** value of **MNS_NOTIFYBYPOS**, **WM_MENUCOMMAND** is sent, instead of **WM_COMMAND**.

! 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

Menus Overview, Menu Messages, **HIWORD**, **LOWORD**, **MENUINFO**, **WM_MENUCOMMAND**, **WM_SYSCOMMAND**

WM_CONTEXTMENU

The **WM_CONTEXTMENU** message notifies a window that the user clicked the right mouse button (*right-clicked*) in the window.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_CONTEXTMENU
```

```

    WPARAM wParam, // handle to window (HWND)
    LPARAM lParam // horizontal and vertical position
);

```

Parameters

wParam

Handle to the window in which the user right-clicked the mouse. This can be a child window of the window receiving the message. For more information about processing this message, see the Remarks section.

lParam

The low-order word specifies the horizontal position of the cursor, in screen coordinates, at the time of the mouse click.

The high-order word specifies the vertical position of the cursor, in screen coordinates, at the time of the mouse click.

Return Values

No return value.

Remarks

A window can process this message by displaying a shortcut menu using the **TrackPopupMenu** or **TrackPopupMenuEx** function. To obtain the horizontal and vertical positions, use the following code:

```

xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);

```

If a window does not display a shortcut menu, it should pass this message to the **DefWindowProc** function. If a window is a child window, **DefWindowProc** sends the message to the parent. Otherwise, **DefWindowProc** displays a default shortcut menu if the specified position is in the window's caption.

DefWindowProc generates the **WM_CONTEXTMENU** message when it processes the **WM_RBUTTONDOWN** or **WM_NCRBUTTONDOWN** message or when the user types SHIFT+F10. The **WM_CONTEXTMENU** message is generated also when the user presses and releases the VK_APPS key.

If the context menu is generated from the keyboard—for example, if the user types SHIFT+F10—then the x-coordinates and y-coordinates are -1, and the application should display the context menu at the location of the current selection instead of at (xPos, yPos).

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.

See Also

Menus Overview, Menu Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **TrackPopupMenu**, **TrackPopupMenuEx**, **WM_NCRBUTTONUP**, **WM_RBUTTONUP**

WM_ENTERMENULOOP

The **WM_ENTERMENULOOP** message informs an application's main window procedure that a menu modal loop has been entered.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_ENTERMENULOOP  
    WPARAM wParam,       // track pop-up menu indicator  
    LPARAM lParam        // not used  
);
```

Parameters

wParam

Specifies whether the window menu was entered using the **TrackPopupMenu** function. This parameter has a value of TRUE if the window menu was entered using **TrackPopupMenu**, and FALSE if it was not.

lParam

This parameter is not used.

Return Values

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function returns 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 winuser.h; include windows.h.

+ See Also

Menus Overview, Menu Messages, **DefWindowProc**, **WM_EXITMENULOOP**

WM_EXITMENULOOP

The **WM_EXITMENULOOP** message informs an application's main window procedure that a menu modal loop has been exited.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,          // handle to window  
    UINT uMsg,          // WM_EXITLOOP  
    WPARAM wParam,     // menu status  
    LPARAM lParam       // not used  
);
```

Parameters

wParam

Specifies whether the menu is a shortcut menu. This parameter has a value of **TRUE** if it is a shortcut menu, and **FALSE** if it is not.

lParam

This parameter is not used.

Return Values

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function returns 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 `winuser.h`; include `windows.h`.

+ See Also

Menus Overview, Menu Messages, **DefWindowProc**, **WM_ENTERMENULOOP**

WM_MENUCOMMAND

The **WM_MENUCOMMAND** message is sent when the user makes a selection from a menu.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_MENUCOMMAND  
    WPARAM wParam,      // item index  
    LPARAM lParam       // handle to menu (HMENU)  
);
```

Parameters

wParam

Specifies the zero-based index of the item selected.

lParam

Handle to the menu for the item selected.

Remarks

The **WM_MENUCOMMAND** message gives you a handle to the menu—so that you can access the menu data in the **MENUINFO** structure—and also gives you the index of the selected item, which is typically what applications need. In contrast, the **WM_COMMAND** message gives you the menu item identifier.

The **WM_MENUCOMMAND** message is sent only for menus that are defined with the **MNS_NOTIFYBYPOS** flag set in the **dwStyle** member of the **MENUINFO** structure.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`.

See Also

Menus Overview, Menu Messages

WM_MENUDRAG

The **WM_MENUDRAG** message is sent to the owner of a drag-and-drop menu when the user drags a menu item.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_MENUDRAG
    WPARAM wParam,     // starting position
    LPARAM lParam      // handle to menu (HMENU)
);

```

Parameters

wParam

Specifies the position of the item where the drag operation began.

lParam

Handle to the menu containing the item.

Return Values

The application should return one of the following values:

| Value | Meaning |
|--------------|--|
| MND_CONTINUE | Menu should remain active. If the mouse is released, it should be ignored. |
| MND_ENDMENU | Menu should be ended. |

Remarks

The application can call the **DoDragDrop** function in response to this message.

To create a drag-and-drop menu, call **SetMenuInfo** with MNS_DRAGDROP.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h.

+ See Also

Menus Overview, Menu Messages, **SetMenuInfo**

WM_MENUGETOBJECT

The **WM_MENUGETOBJECT** message is sent to the owner of a drag-and-drop menu when the mouse cursor enters a menu item or moves from the center of the item to the top or bottom of the item.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MENUGETOBJECT
    WPARAM wParam,      // not used
    LPARAM lParam        // menu information (PMENUGETOBJECTINFO)
);

```

Parameters

wParam

This parameter is not used.

lParam

Pointer to a **MENUGETOBJECTINFO** structure.

Return Values

The application should return one of the following values:

| Values | Meaning |
|------------------|--|
| MNGO_NOERROR | An interface pointer was returned in the <i>pvObj</i> member of MENUGETOBJECTINFO . |
| MNGO_NOINTERFACE | The interface is not supported. |

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in *winuser.h*.

+ See Also

Menus Overview, Menu Messages

WM_MENURBUTTONUP

The **WM_MENURBUTTONUP** message is sent when the user releases the right mouse button while the cursor is on a menu item.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MENURBUTTONUP
    WPARAM wParam,      // mouse release position
    LPARAM lParam        // handle to menu (HMENU)
);

```

Parameters

wParam

Specifies the position of the item when the mouse was released.

lParam

Handle to the menu containing the item.

Remarks

The **WM_MENURBUTTONUP** message allows applications to provide a context-sensitive menu—also known as a shortcut menu—for the menu item specified in this message. To display a context-sensitive menu for a menu item, call the **TrackPopupMenuEx** function with **TPM_RECURSE**.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`.

+ See Also

Menus Overview, Menu Messages

WM_NEXTMENU

The **WM_NEXTMENU** message is sent to an application when the right or left arrow key is used to switch between the menu bar and the system menu.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_NEXTMENU  
    WPARAM wParam,       // virtual-key code  
    LPARAM lParam        // menu information (LPMDINEXTMENU)  
);
```

Parameters

wParam

Specifies the virtual-key code of the key.

lParam

Pointer to a **MDINEXTMENU** structure that contains information about the menu to be activated.

Remarks

In responding to this message, the application can specify both the menu to switch to in the **hmenuNext** member of **MDINEXTMENU** and the window to receive the menu notification messages in the **hwndNext** member of the **MDINEXTMENU** structure. You must set both members for the changes to take effect (they are initially NULL).

! 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

Menus Overview, Menu Messages, **MDINEXTMENU**

WM_UNINITMENUPOPUP

The **WM_UNINITMENUPOPUP** message is sent when a drop-down menu or submenu has been destroyed.

A window receives this message through its **WindowProc** function.

```
HRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_UNINITMENUPOPUP  
    WPARAM wParam,     // handle to menu (HMENU)  
    LPARAM lParam       // menu identifier  
);
```

Parameters

wParam

Handle to the menu.

lParam

The high-order word identifies the menu that was destroyed. Currently, it can only be MF_SYSMENU (the window menu).

Remarks

If an application receives a **WM_INITMENUPOPUP** message, it will receive a **WM_UNINITMENUPOPUP** message.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h.

+ See Also

Menus Overview, Menu Messages, **HIWORD**

Strings

About Strings

The string functions give Microsoft Win32-based applications the means to copy, compare, sort, format, and convert character strings, as well as the means to determine the character type of each character in a string. All the string functions support the single-byte, double-byte, and Unicode character sets, if these character sets are supported by the operating system on which the application is run.

Win32 String Functions

Many Win32 string functions duplicate or enhance familiar string functions from the standard C run-time (CRT) library. Many of the enhancements enable Win32 functions to work with Unicode or extended character sets. For example, the functions in the following table are identical to standard C functions, except that the Win32 functions are enhanced for use with Unicode:

| Win32 function | Standard CRT function |
|-----------------|-----------------------|
| Lstrcat | strcat |
| Lstrcmp | strcmp |
| Lstrcmpi | strcmpi |
| Lstrcpy | strcpy |
| Lstrlen | strlen |

The **strlen** function, for example, always returns the number of bytes in a string, but the **lstrlen** function returns the number of characters, regardless of whether the characters are 1 or 2 bytes wide.

The following Win32 functions differ from standard C functions, such as **tolower** and **toupper**, in that they operate on any character in a character set. By using the **CharLower** function, for example, a Win32-based application can convert an uppercase U with an umlaut (Ü) to lowercase (ü). For more information about character sets, see *Single-Byte Character Sets*:

| Function | Description |
|----------------------|---|
| CharLower | Converts a character or string to lowercase. |
| CharLowerBuff | Converts a character string to lowercase. |
| CharNext | Moves to the next character in a string. |
| CharPrev | Moves to the preceding character in a string. |
| CharUpper | Converts a character or string to uppercase. |
| CharUpperBuff | Converts a string to uppercase. |

The following Win32 functions make determinations about a character, based on the semantics of the language selected by the user. These functions are Unicode-enabled:

| Function | Description |
|---------------------------|---|
| IsCharAlpha | Determines whether a character is alphabetic. |
| IsCharAlphaNumeric | Determines whether a character is alphanumeric. |
| IsCharLower | Determines whether a character is lowercase. |
| IsCharUpper | Determines whether a character is uppercase. |

The **wsprintf** and **wvsprintf** functions are extensions to the standard C functions **sprintf** and **vsprintf**. The Win32 versions support format specifications unique to Unicode.

String Resources

An application that maintains character strings in resources can be translated into new languages with minimum effort. Instead of searching for strings in the source modules, you can translate the strings in the resource file and relink the application. In addition, using string resources simplifies creation of Unicode and non-Unicode versions of the application from the same source files.

The **LoadString** function loads a string resource from an application's executable file. The **FormatMessage** function loads a string resource and interprets formatting options that might be embedded in the string.

Win32-based resources in binary form are stored in Unicode format. When loading resources, applications can use the Unicode version of the resource functions (**LoadStringW**, for example) to obtain resources as Unicode data.

String Reference

String Functions

CharLower

The **CharLower** function converts a character string or a single character to lowercase. If the operand is a character string, the function converts the characters in place.

```
LPTSTR CharLower(  
    LPTSTR psz // single character or string  
);
```

Parameters

psz

[in/out] Pointer to a null-terminated string, or specifies a single character. If the high-order word of this parameter is zero, the low-order word must contain a single character to be converted.

Return Values

If the operand is a character string, the function returns a pointer to the converted string. Since the string is converted in place, the return value is equal to *psz*.

If the operand is a single character, the return value is a 32-bit value whose high-order word is zero and whose low-order word contains the converted character.

There is no indication of success or failure; failure is rare. There is no extended error information for this function; do not call **GetLastError**.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup, or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup, or by using Control Panel. Windows 95 does not have language drivers.

! 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

Strings Overview, String Functions, **CharLowerBuff**, **CharUpper**, **CharUpperBuff**

CharLowerBuff

The **CharLowerBuff** function converts uppercase characters in a buffer to lowercase characters. The function converts the characters in place.

```
DWORD CharLowerBuff(  
    LPTSTR lpsz,           // characters  
    DWORD  cchLength      // number of characters to process  
);
```

Parameters

lpsz

[in/out] Pointer to a buffer containing one or more characters to process.

cchLength

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by *lpsz*.

The function examines each character, and converts uppercase characters to lowercase characters. The function examines the number of characters indicated by *cchLength*, even if one or more characters are null characters.

Return Values

The return value is the number of **TCHARs** processed. For example, if **CharLowerBuff**("Acme of Operating Systems", 10) succeeds, then the return value is 10.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup, or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup, or by using Control Panel. Windows 95 does not have language drivers.

! 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

Strings Overview, String Functions, **CharLower**, **CharUpper**, **CharUpperBuff**

CharNext

The **CharNext** function returns a pointer to the next character in a string.

```
LPTSTR CharNext(  
    LPCWSTR lpsz // current character  
);
```

Parameters

lpsz

[in] Pointer to a character in a null-terminated string.

Return Values

The return value is a pointer to the next character in the string, or to the terminating null character if at the end of the string.

If *lpsz* points to the terminating null character, the return value is equal to *lpsz*.

! 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

Strings Overview, String Functions, **CharNextExA**, **CharPrev**

CharNextExA

The **CharNextExA** function retrieves the pointer to the next character in a string. This function can handle strings consisting of either single-byte or multi-byte characters.

```
LPSTR CharNextExA(
    WORD CodePage,          // code page identifier
    LPCSTR lpCurrentChar,  // current character
    DWORD dwFlags          // reserved; must be zero
);
```

Parameters

CodePage

[in] Identifier of the code page to use to check lead-byte ranges. Can be one of the code-page values provided in the “Code-Page Identifiers” table in Unicode and Character Set Constants, or one of the following predefined values:

| Value | Meaning |
|----------|------------------------------------|
| 0 | Use system default ANSI code page. |
| CP_ACP | Use system default ANSI code page. |
| CP_OEMCP | Use system default OEM code page. |

lpCurrentChar

[in] Pointer to a character in a null-terminated string.

dwFlags

Reserved; must be zero.

Return Values

The return value is a pointer to the next character in the string, or to the terminating null character if at the end of the string.

If *lpCurrentChar* points to the terminating null character, the return value is equal to *lpCurrentChar*.

! 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

Strings Overview, String Functions, **CharNext**, **CharPrevExA**

CharPrev

The **CharPrev** function returns a pointer to the preceding character in a string.

```
LPTSTR CharPrev(
    LPCTSTR lpszStart, // first character
    LPCTSTR lpszCurrent // current character
);
```

Parameters

lpszStart

[in] Pointer to the beginning of the string.

lpszCurrent

[in] Pointer to a character in a null-terminated string.

Return Values

The return value is a pointer to the preceding character in the string, or to the first character in the string if the *lpszCurrent* parameter equals the *lpszStart* parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 95 or later.

Windows CE: Requires version 1.01 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

Strings Overview, String Functions, **CharNext**, **CharPrevExA**

CharPrevExA

The **CharPrevExA** function retrieves the pointer to the preceding character in a string. This function can handle strings consisting of either single-byte or multi-byte characters.

```
LPWSTR CharPrevExA(
    WORD CodePage, // identifier of code page
    LPCWSTR lpStart, // first character
    LPCWSTR lpCurrentChar, // current character
    DWORD dwFlags // reserved: must be zero
);
```


Parameters

CodePage

[in] Identifier of the code page to use to check lead-byte ranges. Can be one of the code-page values provided in the “Code-Page Identifiers” table in Unicode and Character Set Constants, or one of the following predefined values:

| Value | Meaning |
|----------|------------------------------------|
| 0 | Use system default ANSI code page. |
| CP_ACP | Use system default ANSI code page. |
| CP_OEMCP | Use system default OEM code page. |

lpStart

[in] Pointer to the beginning of the string.

lpCurrentChar

[in] Pointer to a character in a null-terminated string.

dwFlags

Reserved; must be zero.

Return Values

The return value is a pointer to the preceding character in the string, or to the first character in the string if the *lpCurrentChar* parameter equals the *lpStart* parameter.



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

Strings Overview, String Functions, **CharNextExA**, **CharPrev**

CharToOem

The **CharToOem** function translates a string into the OEM-defined character set. (OEM stands for original equipment manufacturer.)

```

BOOL CharToOem(
    LPCTSTR lpszSrc, // string to translate
    LPSTR lpszDst, // translated string
);

```

Parameters

lpzSrc

[in] Pointer to the null-terminated string to translate.

lpzDst

[out] Pointer to the buffer for the translated string. If the **CharToOem** function is being used as an ANSI function, the string can be translated in place by setting the *lpzDst* parameter to the same address as the *lpzSrc* parameter. This cannot be done if **CharToOem** is being used as a wide-character function.

Return Values

The return value is always nonzero.

! 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

Strings Overview, String Functions, **CharToOemBuff**, **OemToChar**, **OemToCharBuff**

CharToOemBuff

The **CharToOemBuff** function translates a specified number of characters in a string into the OEM-defined character set. (OEM stands for original equipment manufacturer.)

```
BOOL CharToOemBuff(  
    LPCTSTR lpzSrc,    // string to translate  
    LPSTR lpzDst,      // translated string  
    DWORD cchDstLength // length of string to translate  
);
```

Parameters

lpzSrc

[in] Pointer to the null-terminated string to translate.

lpzDst

[out] Pointer to the buffer for the translated string. If the **CharToOemBuff** function is being used as an ANSI function, the string can be translated in place by setting the *lpzDst* parameter to the same address as the *lpzSrc* parameter. This cannot be done if **CharToOemBuff** is being used as a wide-character function.

cchDstLength

[in] Specifies the number of characters to translate in the string identified by the *lpzSrc* parameter.

Return Values

The return value is always nonzero.

Remarks

Unlike the **CharToOem** function, the **CharToOemBuff** function does not stop converting characters when it encounters a null character in the buffer pointed to by *lpzSrc*. The **CharToOemBuff** function converts all *cchDstLength* characters.

! 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

Strings Overview, String Functions, **CharToOem**, **OemToChar**, **OemToCharBuff**

CharUpper

The **CharUpper** function converts a character string or a single character to uppercase. If the operand is a character string, the function converts the characters in place.

```
LPTSTR CharUpper(  
    LPTSTR lpz // single character or string  
);
```

Parameters

lpz

[in/out] Pointer to a null-terminated string or specifies a single character. If the high-order word of this parameter is zero, the low-order word must contain a single character to be converted.

Return Values

If the operand is a character string, the function returns a pointer to the converted string. Since the string is converted in place, the return value is equal to *lpz*.

If the operand is a single character, the return value is a 32-bit value whose high-order word is zero and whose low-order word contains the converted character.

There is no indication of success or failure; failure is rare. There is no extended error information for this function; do not call **GetLastError**.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup, or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup, or by using Control Panel. Windows 95 does not have language drivers.

! 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

Strings Overview, String Functions, **CharLower**, **CharLowerBuff**, **CharUpperBuff**

CharUpperBuff

The **CharUpperBuff** function converts lowercase characters in a buffer to uppercase characters. The function converts the characters in place.

```
DWORD CharUpperBuff(  
    LPTSTR lpsz,           // characters  
    DWORD  cchLength      // number of characters to process  
);
```

Parameters

lpsz

[in] Pointer to a buffer containing one or more characters to process.

cchLength

[in] Specifies the size, in **TCHARs**, of the buffer pointed to by *lpsz*.

The function examines each character, and converts lowercase characters to uppercase characters. The function examines the number of characters indicated by *cchLength*, even if one or more characters are null characters.

Return Values

The return value is the number of **TCHARs** processed.

For example, if **CharUpperBuff**("Zenith of API Sets", 10) succeeds, then the return value is 10.

Remarks

Windows NT/ 2000: To make the conversion, the function uses the language driver for the current language selected by the user at setup, or by using Control Panel. If no language has been selected, the system completes the conversion by using internal default mapping. The conversion is made based on the code page associated with the process locale.

Windows 95: The function makes the conversion based on the information associated with the user's default locale, which is the locale selected by the user at setup, or by using Control Panel. Windows 95 does not have language drivers.

! 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

Strings Overview, String Functions, **CharLower**, **CharLowerBuff**, **CharUpper**

CompareString

The **CompareString** function compares two character strings, using the locale specified by the given identifier as the basis for the comparison.

```
int CompareString(  
    LCID Locale,           // locale identifier  
    DWORD dwCmpFlags,     // comparison-style options  
    LPCTSTR lpString1,    // first string  
    int cchCount1,        // size of first string  
    LPCTSTR lpString2,    // second string
```

```
    int cchCount2    // size of second string
);
```

Parameters

Locale

[in] Specifies the locale used for the comparison. This parameter can be one of the following predefined locale identifiers:

| Value | Meaning |
|-----------------------|------------------------------------|
| LOCALE_SYSTEM_DEFAULT | The system's default locale. |
| LOCALE_USER_DEFAULT | The current user's default locale. |

This parameter also can be a locale identifier created by the **MAKELCID** macro.

dwCmpFlags

[in] A set of flags that indicate how the function compares the two strings. By default, these flags are not set. This parameter can specify zero to get the default behavior, or it can be any combination of the following values:

| Value | Meaning |
|---------------------|---|
| NORM_IGNORECASE | Ignore case. |
| NORM_IGNOREKANATYPE | Do not differentiate between Hiragana and Katakana characters. Corresponding Hiragana and Katakana characters compare as equal. |
| NORM_IGNORENONSPACE | Ignore nonspacing characters. |
| NORM_IGNORESYMBOLS | Ignore symbols. |
| NORM_IGNOREWIDTH | Do not differentiate between a single-byte character and the same character as a double-byte character. |
| SORT_STRINGSORT | Treat punctuation the same as symbols. |

lpString1

[in] Pointer to the first string to be compared.

cchCount1

[in] Specifies the size, in **TCHARs**, of the string pointed to by the *lpString1* parameter. The count does not include the null-terminator. If this parameter is -1 , the string is assumed to be null-terminated, and the length is calculated automatically.

lpString2

[in] Pointer to the second string to be compared.

cchCount2

[in] Specifies the size, in **TCHARs**, of the string pointed to by the *lpString2* parameter. The count does not include the null-terminator. If this parameter is -1 , the string is assumed to be null-terminated, and the length is calculated automatically.

Return Values

If the function succeeds, the return value is one of the following values:

| Value | Meaning |
|-------------------|---|
| CSTR_EQUAL | The string pointed to by the <i>lpString1</i> parameter is equal in lexical value to the string pointed to by the <i>lpString2</i> parameter. |
| CSTR_GREATER_THAN | The string pointed to by the <i>lpString1</i> parameter is greater in lexical value than the string pointed to by the <i>lpString2</i> parameter. |
| CSTR_LESS_THAN | The string pointed to by the <i>lpString1</i> parameter is less in lexical value than the string pointed to by the <i>lpString2</i> parameter. |

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_INVALID_FLAGS
ERROR_INVALID_PARAMETER

Remarks

Notice that, if the return value is CSTR_EQUAL, the two strings are “equal” in the collation sense, although not necessarily identical.

To maintain the C run-time convention of comparing strings, the value 2 can be subtracted from a nonzero return value. The meaning of < 0 , $== 0$ and > 0 is consistent, then, with the C run times.

If the two strings are of different lengths, they are compared up to the length of the shortest one. If they are equal to that point, the return value will indicate that the longer string is greater. For more information about locale identifiers, see *Locales*.

Typically, strings are compared using what is called a “word sort” technique. In a word sort, all punctuation marks and other nonalphanumeric characters, except for the hyphen and the apostrophe, come before any alphanumeric character. The hyphen and the apostrophe are treated differently than the other nonalphanumeric symbols, in order to ensure that words such as “coop” and “co-op” stay together within a sorted list.

If the SORT_STRINGSORT flag is specified, strings are compared using what is called a “string sort” technique. In a string sort, the hyphen and apostrophe are treated like any other nonalphanumeric symbols: they come before the alphanumeric symbols.

The following table shows a list of words sorted both ways:

| Word sort | String sort | Word sort | String sort |
|-----------|-------------|-----------|-------------|
| billet | bill's | t-ant | t-ant |
| bills | billet | tanya | t-aria |

| Word sort | String sort | Word sort | String sort |
|-----------|-------------|-----------|-------------|
| bill's | bills | t-aria | tanya |
| cannot | can't | sued | sue's |
| cant | cannot | sues | sued |
| can't | cant | sue's | sues |
| con | co-op | went | we're |
| coop | con | were | went |
| co-op | coop | we're | were |

The **Istrcmp** and **Istrcmpi** functions use a word sort. The **CompareString** and **LCMapString** functions default to using a word sort, but use a string sort if their caller sets the `SORT_STRINGSORT` flag.

The **CompareString** function is optimized to run at the highest speed when *dwCmpFlags* is set to either 0 or `NORM_IGNORECASE`, and *cchCount1* and *cchCount2* have the value `-1`.

The **CompareString** function ignores Arabic Kashidas during the comparison. Thus, if two strings are identical, save for the presence of Kashidas, **CompareString** returns a value of 2; the strings are considered “equal” in the collation sense, although they are not necessarily identical.

For DBCS locales, the flag `NORM_IGNORECASE` has an effect on all the wide (two-byte) characters, as well as on the narrow (one-byte) characters. This includes the wide Greek and Cyrillic characters.

In Chinese Simplified, the sorting order used to compare the strings is based on the following sequence: symbols, digit numbers, English letters, and Chinese Simplified characters. The characters within each group sort in character-code order.

In Chinese Traditional, the sorting order used to compare strings is based on the number of strokes in the characters. Symbols, digit numbers, and English characters are considered to have zero strokes. The sort sequence is symbols, digit numbers, English letters, and Chinese Traditional characters. The characters within each stroke-number group sort in character-code order.

In Japanese, the sorting order used to compare the strings is based on the Japanese 50-on sorting sequence. The Kanji ideographic characters sort in character-code order.

In Japanese, the flag `NORM_IGNORENONSPACE` has an effect on the daku-on, handaku-on, chou-on, you-on, and soku-on modifiers, and on the repeat kana/kanji characters.

In Korean, the sort order is based on the sequence: symbols, digit numbers, Jaso and Hangeul, Hanja, and English. Within the Jaso-Hangeul group, each Jaso character is followed by the Hangeuls that start with that Jaso. Hanja characters are sorted in Hangeul pronunciation order. Where multiple Hanja have the same Hangeul pronunciation, they are sorted in character-code order.

The flag `NORM_IGNORENONSPACE` only has an effect for the locales in which accented characters are sorted in a second pass from main characters. All characters in the string first are compared without regard to accents, and (if the strings are equal) a second pass over the strings is performed to compare accents. In this case, this flag causes the second pass to not be performed. For locales that sort accented characters in the first pass, this flag has no effect.

! 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 `winnls.h`; include `windows.h`.

Library: Use `kernel32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **FoldString**, **GetSystemDefaultLCID**, **GetUserDefaultLCID**, **LCMapString**, **Istrcmp**, **Istrcmpi**, **MAKELCID**

FoldString

The **FoldString** function maps one string to another, performing a specified transformation option.

```
int FoldString(
    DWORD dwMapFlags, // mapping transformation options
    LPCTSTR lpSrcStr, // source string
    int cchSrc,       // size of source string
    LPTSTR lpDestStr, // destination buffer
    int cchDest       // size of destination buffer
);
```

Parameters

dwMapFlags

[in] A set of bit flags that indicate the type of transformation to be used during mapping. This value can be a combination of the following values:

| Value | Meaning |
|----------------------------|--|
| <code>MAP_COMPOSITE</code> | Map accented characters to composite characters, in which the accent and base character are represented by two character values. This value cannot be combined with <code>MAP_PRECOMPOSED</code> . |

| Value | Meaning |
|----------------------|---|
| MAP_EXPAND_LIGATURES | Expand all ligature characters, so that they are represented by their two-character equivalent. For example, the ligature “æ” expands to the two characters “a” and “e.” This value cannot be combined with MAP_PRECOMPOSED or MAP_COMPOSITE. |
| MAP_FOLDZONE | Fold compatibility zone characters into standard Unicode equivalents. For information about compatibility zone characters, see the following Remarks section. |
| MAP_FOLDDIGITS | Map all digits to Unicode characters 0 through 9. |
| MAP_PRECOMPOSED | Map accented characters to precomposed characters, in which the accent and base character are combined into a single character value. This value cannot be combined with MAP_COMPOSITE. |

lpSrcStr

[in] Pointer to the string to be mapped.

cchSrc

[in] Specifies the size, in **TCHARs**, of the *lpSrcStr* buffer. If *cchSrc* is -1 , *lpSrcStr* is assumed to be null-terminated, and the length is calculated automatically.

lpDestStr

[out] Pointer to the buffer to store the mapped string.

cchDest

[in] Specifies the size, in **TCHARs**, of the *lpDestStr* buffer. If *cchDest* is zero, the function returns the number of characters required to hold the mapped string, and the buffer pointed to by *lpDestStr* is not used.

Return Values

If the function succeeds, the return value is the number of **TCHARs** written to the destination buffer or, if the *cchDest* parameter is zero, the number of characters required to hold the mapped string.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. **GetLastError** may return one of the following error codes:

ERROR_INSUFFICIENT_BUFFER
 ERROR_INVALID_FLAGS
 ERROR_INVALID_PARAMETER

Remarks

The mapped string is null-terminated if the source string is null-terminated.

The *lpSrcStr* and *lpDestStr* pointers must not be the same. If they are the same, the function fails and **GetLastError** returns ERROR_INVALID_PARAMETER.

The compatibility zone in Unicode consists of characters, in the range 0xF900 through 0xFFEF, that are assigned to characters from other character-encoding standards, but are actually variants of characters that are already in Unicode. The compatibility zone is used to support round-trip mapping to these standards. Applications can use the `MAP_FOLDZONE` flag to avoid supporting the duplication of characters in the compatibility zone.

! 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 `winnls.h`; include `windows.h`.

Library: Use `kernel32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **CompareString**, **LCMapString**

GetStringTypeA

The **GetStringTypeA** function returns character-type information for the characters in the specified source string. For each character in the string, the function sets one or more bits in the corresponding 16-bit element of the output array. Each bit identifies a given character type, such as whether the character is a letter, a digit, or neither.

```

BOOL GetStringTypeA(
    LCID Locale,           // locale identifier
    DWORD dwInfoType,     // information-type options
    LPCSTR lpSrcStr,      // source string
    int cchSrc,           // size of source string
    LPWORD lpCharType     // output buffer
);

```

Parameters

Locale

[in] Specifies the locale identifier. This value uniquely defines the ANSI code page to use to translate the string pointed to by *lpSrcStr* from ANSI to Unicode. The function, then, analyzes each Unicode character for character-type information.

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values:

| Value | Meaning |
|------------------------------------|-----------------------|
| <code>LOCALE_SYSTEM_DEFAULT</code> | Default system locale |
| <code>LOCALE_USER_DEFAULT</code> | Default user locale |

Note that the *Locale* parameter does not exist in the **GetStringTypeW** function. Because of that parameter difference, an application cannot automatically invoke the proper **A** or **W** version of **GetStringType*** through the use of the **#define UNICODE** switch. An application can circumvent this limitation by using **GetStringTypeEx**, which is the recommended function.

dwInfoType

[in] Specifies the type of character information the user wants to retrieve. The various types are divided into different levels (see the following Remarks section for a list of the information included in each type). This parameter can specify one of the following character-type flags:

| Flag | Meaning |
|-----------|---|
| CT_CTYPE1 | Retrieve character-type information |
| CT_CTYPE2 | Retrieve bidirectional layout information |
| CT_CTYPE3 | Retrieve text processing information |

lpSrcStr

[in] Pointer to the string for which character types are requested. If *cchSrc* is -1 , the string is assumed to be null-terminated. This must be an ANSI string. Note that this can be a double-byte character set (DBCS) string if the locale is appropriate for DBCS.

cchSrc

[in] Specifies the size, in characters, of the string pointed to by the *lpSrcStr* parameter. If this count includes a null terminator, the function returns character-type information for the null terminator. If this value is -1 , the string is assumed to be null-terminated, and the length is calculated automatically.

lpCharType

[out] Pointer to an array of 16-bit values. The length of this array must be large enough to receive one 16-bit value for each character in the source string. When the function returns, this array contains one word corresponding to each character in the source string.

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**. **GetLastError** may return one of the following error codes:

ERROR_INVALID_FLAGS
ERROR_INVALID_PARAMETER

Remarks

The *lpSrcStr* and *lpCharType* pointers must not be the same. If they are the same, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

The *Locale* parameter is only used to perform string conversion to Unicode. It has nothing to do with the CTYPEs the function returns. The CTYPEs are solely determined by Unicode code points, and do not vary on a locale basis. For example, Greek letters are C1_ALPHA for any *Locale* value.

The character-type bits are divided into several levels. The information for one level can be retrieved by a single call to this function. Each level is limited to 16 bits of information, so that the other mapping routines, which are limited to 16 bits of representation per character, also can return character-type information.

The character types supported by this function include the following:

Ctype 1

These types support ANSI C and POSIX (LC_CTYPE) character-typing functions. A bitwise-OR of these values is returned in the array pointed to by the *lpCharType* parameter, when the *dwInfoType* parameter is set to CT_CTYPE1. For DBCS locales, the Ctype 1 attributes apply to both narrow characters and wide characters. The Japanese hiragana and katakana characters, and the kanji ideograph characters, all have the C1_ALPHA attribute.

| Name | Value | Meaning |
|-----------|--------|---|
| C1_UPPER | 0x0001 | Uppercase |
| C1_LOWER | 0x0002 | Lowercase |
| C1_DIGIT | 0x0004 | Decimal digits |
| C1_SPACE | 0x0008 | Space characters |
| C1_PUNCT | 0x0010 | Punctuation |
| C1_CNTRL | 0x0020 | Control characters |
| C1_BLANK | 0x0040 | Blank characters |
| C1_XDIGIT | 0x0080 | Hexadecimal digits |
| C1_ALPHA | 0x0100 | Any linguistic character: alphabetic, syllabary, or ideographic |

The following character types are either constant or computable from basic types and do not need to be supported by this function:

| Type | Description |
|--------------|--|
| Alphanumeric | Alphabetic characters and digits (C1_ALPHA and C1_DIGIT) |
| Printable | Graphic characters and blanks (all C1_* types except C1_CNTRL) |

The Windows version 3.1 functions **IsCharUpper** and **IsCharLower** do not always produce correct results for characters in the range 0x80–0x9f, so they may produce different results than this function for characters in that range. (For example, the German Windows version 3.1 language driver incorrectly reports 0x9a, lowercase s hacek, as uppercase.)

Ctype 2

These types support proper layout of Unicode text. For DBCS locales, Ctype 2 applies to both narrow and wide characters. The direction attributes are assigned so that the bidirectional layout algorithm standardized by Unicode produces accurate results. These types are mutually exclusive. For more information about the use of these attributes, see *The Unicode Standard: Worldwide Character Encoding, Volumes 1 and 2*, Addison Wesley Publishing Company: 1991, 1992, ISBN 0201567881.

| Name | Value | Meaning |
|-----------------------|--------------|---|
| Strong | | |
| C2_LEFTTORIGHT | 0x0001 | Left to right |
| C2_RIGHTTOLEFT | 0x0002 | Right to left |
| Weak | | |
| C2_EUROPENUMBER | 0x0003 | European number, European digit |
| C2_EUROPESEPARATOR | 0x0004 | European numeric separator |
| C2_EUROPETERMINATOR | 0x0005 | European numeric terminator |
| C2_ARABICNUMBER | 0x0006 | Arabic number |
| C2_COMMONSEPARATOR | 0x0007 | Common numeric separator |
| Neutral | | |
| C2_BLOCKSEPARATOR | 0x0008 | Block separator |
| C2_SEGMENTSEPARATOR | 0x0009 | Segment separator |
| C2_WHITESPACE | 0x000A | White space |
| C2_OTHERNEUTRAL | 0x000B | Other neutrals |
| Not applicable | | |
| C2_NOTAPPLICABLE | 0x0000 | No implicit directionality (for example, control codes) |

Ctype 3

These types are intended to be placeholders for extensions to the POSIX types required for general text processing or for the standard C library functions. A bitwise-OR of these values is returned when *dwInfoType* is set to CT_CTYPE3. For DBCS locales, the Ctype 3 attributes apply to both narrow characters and wide characters. The Japanese hiragana and katakana characters, and the kanji ideograph characters, all have the C3_ALPHA attribute.

| Name | Value | Meaning |
|---------------|--------------|---------------------------|
| C3_NONSPACING | 0x0001 | Nonspacing mark |
| C3_DIACRITIC | 0x0002 | Diacritic nonspacing mark |
| C3_VOWELMARK | 0x0004 | Vowel nonspacing mark |
| C3_SYMBOL | 0x0008 | Symbol |
| C3_KATAKANA | 0x0010 | Katakana character |

(continued)

(continued)

| Name | Value | Meaning |
|-----------------------|--------|--|
| C3_HIRAGANA | 0x0020 | Hiragana character |
| C3_HALFWIDTH | 0x0040 | Half-width (narrow) character |
| C3_FULLWIDTH | 0x0080 | Full-width (wide) character |
| C3_IDEOGRAPH | 0x0100 | Ideographic character |
| C3_KASHIDA | 0x0200 | Arabic Kashida character |
| C3_LEXICAL | 0x0400 | Punctuation that is counted as part of the word (Kashida, hyphen, feminine/masculine ordinal indicators, equal sign, and so forth) |
| C3_ALPHA | 0x8000 | All linguistic characters (alphabetic, syllabary, and ideographic) |
| Not applicable | | |
| C3_NOTAPPLICABLE | 0x0000 | Not applicable |

! 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 winnls.h; include windows.h.

Library: Use kernel32.lib.

+ See Also

Strings Overview, String Functions, **GetLocaleInfo**, **GetStringTypeEx**, **GetStringTypeW**

GetStringTypeEx

The **GetStringTypeEx** function returns character-type information for the characters in the specified source string. For each character in the string, the function sets one or more bits in the corresponding 16-bit element of the output array. Each bit identifies a given character type, such as whether the character is a letter, a digit, or neither.

Unlike its close relatives **GetStringTypeA** and **GetStringTypeW**, **GetStringTypeEx** exhibits appropriate **A** or **W** behavior through the use of the **#define UNICODE** switch. It is the recommended function.

```

BOOL GetStringTypeEx(
    LCID Locale,           // locale identifier
    DWORD dwInfoType,     // information-type options
    LPCTSTR lpSrcStr,     // source string
    int cchSrc,           // size of source string
    LPWORD lpCharType     // output buffer
);

```

Parameters

Locale

[in] Specifies the locale identifier. This value uniquely defines the ANSI code page to use to translate the string pointed to by *lpSrcStr* from ANSI to Unicode. The function, then, analyzes each Unicode character for character-type information. Note that the **W** version of this function ignores this parameter.

This parameter can be a locale identifier created by the **MAKELCID** macro, or one of the following predefined values:

| Value | Meaning |
|-----------------------|-----------------------|
| LOCALE_SYSTEM_DEFAULT | Default system locale |
| LOCALE_USER_DEFAULT | Default user locale |

dwInfoType

[in] Specifies the type of character information the user wants to retrieve. The various types are divided into different levels (see the following Remarks section for a list of the information included in each type). This parameter can specify one of the following character-type values:

| Value | Meaning |
|-----------|---|
| CT_CTYPE1 | Retrieve character-type information |
| CT_CTYPE2 | Retrieve bidirectional layout information |
| CT_CTYPE3 | Retrieve text processing information |

lpSrcStr

[in] Pointer to the string for which character types are requested. If *cchSrc* is -1 , the string is assumed to be null-terminated.

cchSrc

[in] Specifies the size, in **TCHARs**, of the string pointed to by the *lpSrcStr* parameter. If this count includes a null terminator, the function returns character-type information for the null terminator. If this value is -1 , the string is assumed to be null-terminated and the length is calculated automatically.

lpCharType

[out] Pointer to an array of 16-bit values. The length of this array must be large enough to receive one 16-bit value for each character in the source string. When the function returns, this array contains one word corresponding to each character in the source string.

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**. **GetLastError** may return one of the following error codes:

ERROR_INVALID_FLAGS
 ERROR_INVALID_PARAMETER

Remarks

The **GetStringTypeEx** function exists to circumvent a limitation caused by the difference in parameters of **GetStringTypeA** and **GetStringTypeW**. That parameter difference prevents an application from automatically invoking the proper **A** or **W** version of **GetStringType*** through the use of the **#define UNICODE** switch. **GetStringTypeEx**, on the other hand, behaves properly as regards that switch. Thus, it is the recommended function.

The *Locale* parameter is only used to perform string conversion to Unicode. It has nothing to do with the CTYPEs the function returns. The CTYPEs are determined solely by Unicode code points, and do not vary on a locale basis. For example, Greek letters are C1_ALPHA for any *Locale* value.

The *lpSrcStr* and *lpCharType* pointers must not be the same. If they are the same, then, the function fails and **GetLastError** returns ERROR_INVALID_PARAMETER.

The character-type bits are divided into several levels. The information for one level can be retrieved by a single call to this function. Each level is limited to 16 bits of information, so that the other mapping routines, which are limited to 16 bits of representation per character, also can return character-type information.

The character types supported by this function include the following:

Ctype 1

These types support ANSI C and POSIX (LC_CTYPE) character-typing functions. A combination of these values is returned in the array pointed to by the *lpCharType* parameter when the *dwInfoType* parameter is set to CT_CTYPE1:

| Name | Value | Meaning |
|-----------|--------|---|
| C1_UPPER | 0x0001 | Uppercase |
| C1_LOWER | 0x0002 | Lowercase |
| C1_DIGIT | 0x0004 | Decimal digits |
| C1_SPACE | 0x0008 | Space characters |
| C1_PUNCT | 0x0010 | Punctuation |
| C1_CNTRL | 0x0020 | Control characters |
| C1_BLANK | 0x0040 | Blank characters |
| C1_XDIGIT | 0x0080 | Hexadecimal digits |
| C1_ALPHA | 0x0100 | Any linguistic character: alphabetic, syllabary, or ideographic |

The following character types are either constant or computable from basic types, and do not need to be supported by this function:

| Type | Description |
|--------------|---|
| Alphanumeric | Alphabetical characters and digits (C1_ALPHA and C1_DIGIT) |
| Printable | Graphical characters and blank (all C1_* types except for C1_CNTRL) |

Ctype 2

These types support proper layout of Unicode text. The direction attributes are assigned so that the bidirectional layout algorithm standardized by Unicode produces accurate results. These types are mutually exclusive. For more information about the use of these attributes, see *The Unicode Standard: Worldwide Character Encoding, Volumes 1 and 2*, Addison Wesley Publishing Company: 1991, 1992, ISBN 0201567881.

| Name | Value | Meaning |
|-----------------------|--------|---|
| Strong | | |
| C2_LEFTTORIGHT | 0x0001 | Left to right |
| C2_RIGHTTOLEFT | 0x0002 | Right to left |
| Weak | | |
| C2_EUROPENUMBER | 0x0003 | European number, European digit |
| C2_EUROPESEPARATOR | 0x0004 | European numeric separator |
| C2_EUROPETERMINATOR | 0x0005 | European numeric terminator |
| C2_ARABICNUMBER | 0x0006 | Arabic number |
| C2_COMMONSEPARATOR | 0x0007 | Common numeric separator |
| Neutral | | |
| C2_BLOCKSEPARATOR | 0x0008 | Block separator |
| C2_SEGMENTSEPARATOR | 0x0009 | Segment separator |
| C2_WHITESPACE | 0x000A | White space |
| C2_OTHERNEUTRAL | 0x000B | Other neutrals |
| Not applicable | | |
| C2_NOTAPPLICABLE | 0x0000 | No implicit directionality (for example, control codes) |

Ctype 3

These types are intended to be placeholders for extensions to the POSIX types required for general text processing or for the standard C library functions. A combination of these values is returned when *dwInfoType* is set to CT_CTYPE3:

| Name | Value | Meaning |
|---------------|--------|---------------------------|
| C3_NONSPACING | 0x0001 | Nonspacing mark |
| C3_DIACRITIC | 0x0002 | Diacritic nonspacing mark |
| C3_VOWELMARK | 0x0004 | Vowel nonspacing mark |

(continued)

(continued)

| Name | Value | Meaning |
|-----------------------|--------|--|
| C3_SYMBOL | 0x0008 | Symbol |
| C3_KATAKANA | 0x0010 | Katakana character |
| C3_HIRAGANA | 0x0020 | Hiragana character |
| C3_HALFWIDTH | 0x0040 | Half-width character |
| C3_FULLWIDTH | 0x0080 | Full-width character |
| C3_IDEOGRAPH | 0x0100 | Ideographic character |
| C3_KASHIDA | 0x0200 | Arabic Kashida character |
| C3_LEXICAL | 0x0400 | Punctuation that is counted as part of the word (Kashida, hyphen, feminine/masculine ordinal indicators, equal sign, and so forth) |
| C3_ALPHA | 0x8000 | All linguistic characters (alphabetic, syllabary, and ideographic) |
| Not applicable | | |
| C3_NOTAPPLICABLE | 0x0000 | Not applicable |

! 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 winnls.h; include windows.h.

Library: Use kernel32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **GetLocaleInfo**, **GetStringTypeA**, **GetStringTypeW**

GetStringTypeW

The **GetStringTypeW** function returns character-type information for the characters in the specified source string. For each character in the string, the function sets one or more bits in the corresponding 16-bit element of the output array. Each bit identifies a given character type, such as whether the character is a letter, a digit, or neither.

```

BOOL GetStringTypeW(
    DWORD dwInfoType, // information-type options
    LPCWSTR lpSrcStr, // source string
    int cchSrc,       // number of characters in string
    LPWORD lpCharType // output buffer
);

```

Parameters

dwInfoType

[in] Specifies the type of character information the user wants to retrieve. The various types are divided into different levels (see the following Remarks section for a list of the information included in each type). This parameter can specify one of the following character-type values.

| Value | Meaning |
|------------|---|
| CT_CTTYPE1 | Retrieve character-type information |
| CT_CTTYPE2 | Retrieve bidirectional layout information |
| CT_CTTYPE3 | Retrieve text processing information |

lpSrcStr

[in] Pointer to the string for which character types are requested. If *cchSrc* is -1 , the string is assumed to be null-terminated. This must be a Unicode string.

cchSrc

[in] Specifies the size, in wide characters, of the string pointed to by the *lpSrcStr* parameter. If this count includes a null terminator, the function returns character-type information for the null terminator. If this value is -1 , the string is assumed to be null-terminated, and the length is calculated automatically.

lpCharType

[out] Pointer to an array of 16-bit values. The length of this array must be large enough to receive one 16-bit value for the number of characters specified in the *cchSrc* parameter. When the function returns, this array contains one word corresponding to each Unicode character in the source string.

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**. **GetLastError** may return one of the following error codes:

ERROR_INVALID_FLAGS
ERROR_INVALID_PARAMETER

Remarks

Note that the **GetStringTypeA** function has one more parameter than the **GetStringTypeW** function: **GetStringTypeA** has a first parameter that is an **LCID** named *Locale*. This parameter does not exist in the **GetStringTypeW** function. Because of that parameter difference, an application cannot invoke automatically the proper **A** or **W** version of **GetStringType*** through the use of the **#define UNICODE** switch. An application can circumvent this limitation by using **GetStringTypeEx**; it is the recommended function.

The *lpSrcStr* and *lpCharType* pointers must not be the same. If they are the same, the function fails and **GetLastError** returns `ERROR_INVALID_PARAMETER`.

The character-type bits are divided into several levels. The information for one level can be retrieved by a single call to this function. Each level is limited to 16 bits of information, so that the other mapping routines, which are limited to 16 bits of representation per character, can also return character-type information.

The character types supported by this function include the following:

Ctype 1

These types support ANSI C and POSIX (`LC_CTYPE`) character-typing functions. A combination of these values is returned in the array pointed to by the *lpCharType* parameter when the *dwInfoType* parameter is set to `CT_CTYPE1`:

| Name | Value | Meaning |
|------------------------|---------------------|---|
| <code>C1_UPPER</code> | <code>0x0001</code> | Uppercase |
| <code>C1_LOWER</code> | <code>0x0002</code> | Lowercase |
| <code>C1_DIGIT</code> | <code>0x0004</code> | Decimal digits |
| <code>C1_SPACE</code> | <code>0x0008</code> | Space characters |
| <code>C1_PUNCT</code> | <code>0x0010</code> | Punctuation |
| <code>C1_CNTRL</code> | <code>0x0020</code> | Control characters |
| <code>C1_BLANK</code> | <code>0x0040</code> | Blank characters |
| <code>C1_XDIGIT</code> | <code>0x0080</code> | Hexadecimal digits |
| <code>C1_ALPHA</code> | <code>0x0100</code> | Any linguistic character: alphabetic, syllabary, or ideographic |

The following character types are either constant or computable from basic types, and do not need to be supported by this function:

| Type | Description |
|--------------|---|
| Alphanumeric | Alphabetical characters and digits (<code>C1_ALPHA</code> and <code>C1_DIGIT</code>) |
| Printable | Graphical characters and blanks (all <code>C1_*</code> types except <code>C1_CNTRL</code>) |

Ctype 2

These types support proper layout of Unicode text. The direction attributes are assigned so that the bidirectional layout algorithm standardized by Unicode produces accurate results. These types are mutually exclusive. For more information about the use of these attributes, see *The Unicode Standard: Worldwide Character Encoding, Volumes 1 and 2*, Addison Wesley Publishing Company: 1991, 1992, ISBN 0201567881.

| Name | Value | Meaning |
|-----------------------|--------|---|
| Strong | | |
| C2_LEFTTORIGHT | 0x0001 | Left to right |
| C2_RIGHTTOLEFT | 0x0002 | Right to left |
| Weak | | |
| C2_EUROPENUMBER | 0x0003 | European number, European digit |
| C2_EUROPESEPARATOR | 0x0004 | European numeric separator |
| C2_EUROPETERMINATOR | 0x0005 | European numeric terminator |
| C2_ARABICNUMBER | 0x0006 | Arabic number |
| C2_COMMONSEPARATOR | 0x0007 | Common numeric separator |
| Neutral | | |
| C2_BLOCKSEPARATOR | 0x0008 | Block separator |
| C2_SEGMENTSEPARATOR | 0x0009 | Segment separator |
| C2_WHITESPACE | 0x000A | White space |
| C2_OTHERNEUTRAL | 0x000B | Other neutrals |
| Not applicable | | |
| C2_NOTAPPLICABLE | 0x0000 | No implicit directionality (for example, control codes) |

Ctype 3

These types are intended to be placeholders for extensions to the POSIX types required for general text processing or for the standard C library functions. A combination of these values is returned when *dwInfoType* is set to CT_CTYPE3:

| Name | Value | Meaning |
|---------------|--------|--|
| C3_NONSPACING | 0x0001 | Nonspacing mark |
| C3_DIACRITIC | 0x0002 | Diacritic nonspacing mark |
| C3_VOWELMARK | 0x0004 | Vowel nonspacing mark |
| C3_SYMBOL | 0x0008 | Symbol |
| C3_KATAKANA | 0x0010 | Katakana character |
| C3_HIRAGANA | 0x0020 | Hiragana character |
| C3_HALFWIDTH | 0x0040 | Half-width character |
| C3_FULLWIDTH | 0x0080 | Full-width character |
| C3_IDEOGRAPH | 0x0100 | Ideographic character |
| C3_KASHIDA | 0x0200 | Arabic Kashida character |
| C3_LEXICAL | 0x0400 | Punctuation that is counted as part of the word (Kashida, hyphen, feminine/masculine ordinal indicators, equal sign, and so forth) |

(continued)

(continued)

| Name | Value | Meaning |
|-----------------------|--------|--|
| C3_ALPHA | 0x8000 | All linguistic characters (alphabetic, syllabary, and ideographic) |
| Not applicable | | |
| C3_NOTAPPLICABLE | 0x0000 | Not applicable |

! Requirements**Windows NT/2000:** Requires Windows NT 3.1 or later.**Windows 95/98:** Unsupported.**Windows CE:** Unsupported.**Header:** Declared in winnls.h; include windows.h.**Library:** Use kernel32.lib.**+** See AlsoStrings Overview, String Functions, **GetLocaleInfo**, **GetStringTypeA**, **GetStringTypeEx**

IsCharAlpha

The **IsCharAlpha** function determines whether a character is an alphabetic character. This determination is based on the semantics of the language selected by the user during setup, or through Control Panel.

```

BOOL IsCharAlpha(
    TCHAR ch // character to test
);

```

Parameters*ch*

[in] Specifies the character to be tested.

Return Values

If the character is alphabetic, the return value is nonzero.

If the character is not alphabetic, 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 winuser.h; include windows.h.

Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Strings Overview, String Functions, **IsCharAlphaNumeric**

IsCharAlphaNumeric

The **IsCharAlphaNumeric** function determines whether a character is either an alphabetic, or a numeric character. This determination is based on the semantics of the language selected by the user during setup, or through Control Panel.

```
BOOL IsCharAlphaNumeric(  
    TCHAR ch // character to test  
);
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is alphanumeric, the return value is nonzero.

If the character is not alphanumeric, 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 winuser.h; include windows.h.

Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Strings Overview, String Functions, **IsCharAlpha**

IsCharLower

The **IsCharLower** function determines whether a character is lowercase. This determination is based on the semantics of the language selected by the user during setup, or through Control Panel.

```
BOOL IsCharLower(  
    TCHAR ch // character to test  
);
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is lowercase, the return value is nonzero.

If the character is not lowercase, 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 winuser.h; include windows.h.

Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **IsCharUpper**

IsCharUpper

The **IsCharUpper** function determines whether a character is uppercase. This determination is based on the semantics of the language selected by the user during setup, or through Control Panel.

```
BOOL IsCharUpper(  
    TCHAR ch // character to test  
);
```

Parameters

ch

[in] Specifies the character to be tested.

Return Values

If the character is uppercase, the return value is nonzero.

If the character is not uppercase, 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **IsCharLower**

LoadString

The **LoadString** function loads a string resource from the executable (.exe) file associated with a specified module, copies the string into a buffer, and appends a terminating null character.

```
int LoadString(
    HINSTANCE hInstance, // handle to resource module
    UINT uID,           // resource identifier
    LPTSTR lpBuffer,    // resource buffer
    int nBufferMax      // size of buffer
);
```

Parameters

hInstance

[in] Handle to an instance of the module whose executable file contains the string resource.

uID

[in] Specifies the integer identifier of the string to be loaded.

lpBuffer

[out] Pointer to the buffer to receive the string.

nBufferMax

[in] Specifies the size of the buffer, in **TCHARs**. The string is truncated and null-terminated if it is longer than the number of characters specified.

Return Values

If the function succeeds, the return value is the number of **TCHARs** copied into the buffer, not including the null-terminating character, or zero if the string resource does not exist. 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`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **FormatMessage**, **LoadAccelerators**, **LoadBitmap**, **LoadCursor**, **LoadIcon**, **LoadMenu**, **LoadMenuIndirect**

Istrcat

The **Istrcat** function appends one string to another.

```
LPTSTR Istrcat(  
    LPTSTR lpString1, // first string  
    LPCWSTR lpString2 // second string  
);
```

Parameters

lpString1

[in/out] Pointer to a null-terminated string. The buffer must be large enough to contain both strings.

lpString2

[in] Pointer to the null-terminated string to be appended to the string specified in the *lpString1* parameter.

Return Values

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is NULL. 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 `winbase.h`; include `windows.h`.

Library: Use `kernel32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **Istrcmp**, **Istrcmpi**, **Istrcpy**, **Istrlen**

Istrcmp

The **Istrcmp** function compares two character strings. The comparison is case-sensitive.

To perform a comparison that is not case-sensitive, use the **Istrcmpi** function.

```
int Istrcmp(  
    LPCTSTR lpString1, // first string  
    LPCTSTR lpString2 // second string  
);
```

Parameters

lpString1

[in] Pointer to the first null-terminated string to be compared.

lpString2

[in] Pointer to the second null-terminated string to be compared.

Return Values

If the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative. If the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, the return value is positive. If the strings are equal, the return value is zero.

Remarks

The **Istrcmp** function compares two strings by checking the first characters against each other, the second characters against each other, and so on, until it finds an inequality or reaches the ends of the strings.

The function returns the difference of the values of the first unequal characters it encounters. For example, **Istrcmp** determines that “abcz” is greater than “abcdefg” and returns the difference of *z* and *d*.

The language (locale) selected by the user at setup time, or through Control Panel, determines the string that is greater (or whether the strings are the same). If no language (locale) is selected, the system performs the comparison by using default values.

With a double-byte character set (DBCS) version of the system, this function can compare two DBCS strings.

The **Istrcmp** function uses a word sort, rather than a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as “coop” and “co-op” stay together within a sorted list. Note that in 16-bit versions of Windows **Istrcmp** uses a string sort. For a detailed discussion of word sorts and string sorts, see the **Remarks** section of the reference page for the **CompareString** 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 winbase.h; include windows.h.

Library: Use kernel32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Strings Overview, String Functions, **CompareString**, **Istrcat**, **Istrcmpi**, **Istrcpy**, **Istrlen**

Istrcmpi

The **Istrcmpi** function compares two character strings. The comparison is not case-sensitive.

To perform a comparison that is case-sensitive, use the **Istrcmp** function.

```
int Istrcmpi(  
    LPCTSTR lpString1, // first string  
    LPCTSTR lpString2 // second string  
);
```

Parameters

lpString1

[in] Pointer to the first null-terminated string to be compared.

lpString2

[in] Pointer to the second null-terminated string to be compared.

Return Values

If the string pointed to by *lpString1* is less than the string pointed to by *lpString2*, the return value is negative. If the string pointed to by *lpString1* is greater than the string pointed to by *lpString2*, the return value is positive. If the strings are equal, the return value is zero.

Remarks

The **lstrcmpi** function compares two strings by checking the first characters against each other, the second characters against each other, and so on, until it finds an inequality or reaches the ends of the strings.

The function returns the difference of the values of the first unequal characters it encounters. For example, **lstrcmpi** determines that “abcz” is greater than “abcdefg” and returns the difference of *z* and *d*.

The language (locale) selected by the user at setup time, or through Control Panel, determines the string that is greater (or whether the strings are the same). If no language (locale) is selected, the system performs the comparison by using default values.

For some locales, the **lstrcmpi** function may be insufficient. If this occurs, use **CompareString** to ensure proper comparison. For example, in Japan call **CompareString** with the **IGNORE_CASE**, **IGNORE_KANATYPE**, and **IGNORE_WIDTH** values to achieve the most appropriate non-exact string comparison. The **IGNORE_KANATYPE** and **IGNORE_WIDTH** values are ignored in non-Asian locales, so you can set these values for all locales and be guaranteed to have a culturally correct “insensitive” sorting, regardless of the locale. Note that specifying these values slows performance, so use them only when necessary.

With a double-byte character set (DBCS) version of the system, this function can compare two DBCS strings.

The **lstrcmpi** function uses a word sort, instead of a string sort. A word sort treats hyphens and apostrophes differently than it treats other symbols that are not alphanumeric, in order to ensure that words such as “coop” and “co-op” stay together within a sorted list. Note that in 16-bit versions of Windows **lstrcmpi** uses a string sort. For a detailed discussion of word sorts and string sorts, see the Remarks section of the reference page for the **CompareString** 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 winbase.h; include windows.h.

Library: Use kernel32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **CompareString**, **Istrcat**, **Istrcmp**, **Istrcpy**, **Istrlen**

Istrcpy

The **Istrcpy** function copies a string to a buffer.

To copy a specified number of characters, use the **Istrcpyn** function.

```
LPTSTR Istrcpy(  
    LPTSTR lpString1, // destination buffer  
    LPCTSTR lpString2 // string  
);
```

Parameters

lpString1

[out] Pointer to a buffer to receive the contents of the string pointed to by the *lpString2* parameter. The buffer must be large enough to contain the string, including the terminating null character.

lpString2

[in] Pointer to the null-terminated string to be copied.

Return Values

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

With a double-byte character set (DBCS) version of the system, this function can be used to copy a DBCS string.

The **Istrcpy** function has an undefined behavior if source and destination buffers overlap.

! 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 winbase.h; include windows.h.

Library: Use kernel32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **Istrcat**, **Istrcmp**, **Istrcmpi**, **Istrlen**

Istrcpyn

The **Istrcpyn** function copies a specified number of characters from a source string into a buffer.

```
LPTSTR Istrcpyn(  
    LPTSTR lpString1, // destination buffer  
    LPCTSTR lpString2, // string  
    int iMaxLength // number of characters to copy  
);
```

Parameters

lpString1

[out] Pointer to a buffer into which the function copies characters. The buffer must be large enough to contain the number of **TCHARs** specified by *iMaxLength*, including room for a terminating null character.

lpString2

[in] Pointer to a null-terminated string from which the function copies characters.

iMaxLength

[in] Specifies the number of **TCHARs** to be copied from the string pointed to by *lpString2* into the buffer pointed to by *lpString1*, including a terminating null character.

Return Values

If the function succeeds, the return value is a pointer to the buffer.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

Note that the buffer pointed to by *lpString1* must be large enough to include a terminating null character, and the string length value specified by *iMaxLength* includes room for a terminating null character. Thus, the following code

```
TCHAR chBuffer[512];  
Istrcpyn(chBuffer, "abcdefghijklmnop", 4);
```

copies the string "abc", followed by a terminating null character, to *chBuffer*.

The **Istrcpyn** function has an undefined behavior if source and destination buffers overlap.

! 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 winbase.h; include windows.h.

Library: Use kernel32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Strings Overview, String Functions, **Istrcat**, **Istrcmp**, **Istrcmpi**, **Istrcpy**, **Istrlen**

Istrlen

The **Istrlen** function returns the length in **TCHARs** of the specified string (not including the terminating null character).

```
int Istrlen(  
    LPCWSTR lpString // string to count  
);
```

Parameters

lpString

[in] Pointer to a null-terminated string.

Return Values

The return value specifies the length of the string, in **TCHARs**.

! 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 winbase.h; include windows.h.

Library: Use kernel32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows and Windows NT/2000.

+ See Also

Strings Overview, String Functions, **Istrcat**, **Istrcmp**, **Istrcmpi**, **Istrcpy**

OemToChar

The **OemToChar** function translates a string from the OEM-defined character set into either an ANSI or a wide-character string. (OEM stands for original equipment manufacturer.)

```
BOOL OemToChar(  
    LPCSTR lpszSrc, // string to translate  
    LPTSTR lpszDst  // translated string  
);
```

Parameters

lpszSrc

[in] Pointer to a null-terminated string of characters from the OEM-defined character set.

lpszDst

[out] Pointer to the buffer for the translated string. If the **OemToChar** function is being used as an ANSI function, the string can be translated in place by setting the *lpszDst* parameter to the same address as the *lpszSrc* parameter. This cannot be done if **OemToChar** is being used as a wide-character function.

Return Values

The return value is always nonzero.

! 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

Strings Overview, String Functions, **CharToOem**, **CharToOemBuff**, **OemToCharBuff**

OemToCharBuff

The **OemToCharBuff** function translates a specified number of characters in a string from the OEM-defined character set into either an ANSI or a wide-character string. (OEM is the acronym for original equipment manufacturer.)

```
BOOL OemToCharBuff(  
    LPCTSTR lpszSrc, // string to translate
```

(continued)

(continued)

```
LPTSTR lpszDst, // translated string buffer
DWORD cchDstLength // size of buffer
);
```

Parameters

lpszSrc

[in] Pointer to a buffer containing one or more characters from the OEM-defined character set.

lpszDst

[out] Pointer to the buffer for the translated string. If the **OemToCharBuff** function is being used as an ANSI function, the string can be translated in place by setting the *lpszDst* parameter to the same address as the *lpszSrc* parameter. This cannot be done if the **OemToCharBuff** function is being used as a wide-character function.

cchDstLength

[in] Specifies the number of **TCHARs** to translate in the buffer identified by the *lpszSrc* parameter.

Return Values

The return value is always nonzero.

Remarks

Unlike the **OemToChar** function, the **OemToCharBuff** function does not stop converting characters when it encounters a null character in the buffer pointed to by *lpszSrc*. The **OemToCharBuff** function converts all *cchDstLength* characters.

! 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

Strings Overview, String Functions, **CharToOem**, **CharToOemBuff**, **OemToChar**

wsprintf

The **wsprintf** function formats and stores a series of characters and values in a buffer. Any arguments are converted and copied to the output buffer, according to the corresponding format specification in the format string. The function appends a

terminating null character to the characters it writes, but the return value does not include the terminating null character in its character count.

```
int wprintf(  
    LPCTSTR lpOut,    // output buffer  
    LPCTSTR lpFmt,    // format-control string  
    ...               // optional arguments  
);
```

Parameters

lpOut

[out] Pointer to a buffer to receive the formatted output. The maximum size of the buffer is 1024 bytes.

lpFmt

[in] Pointer to a null-terminated string that contains the format-control specifications. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. For more information about the format specification, see the Remarks section.

...

[in] Specifies one or more optional arguments. The number and type of argument parameters depend on the corresponding format-control specifications in the *lpFmt* parameter.

Return Values

If the function succeeds, the return value is the number of characters stored in the output buffer, not counting the terminating null character.

If the function fails, the return value is less than the length of the expected output. To get extended error information, call **GetLastError**.

Remarks

The format-control string contains format specifications that determine the output format for the arguments following the *lpFmt* parameter. Format specifications, discussed below, always begin with a percent sign (%). If a percent sign is followed by a character that has no meaning as a format field, the character is not formatted (for example, %% produces a single percent-sign character).

The format-control string is read from left to right. When the first format specification (if any) is encountered, it causes the value of the first argument after the format-control string to be converted and copied to the output buffer, according to the format specification. The second format specification causes the second argument to be converted and copied, and so on. If there are more arguments than format specifications, the extra arguments are ignored. If there are not enough arguments for all of the format specifications, the results are undefined.

A format specification has the following form:

`%[-][#][0][width][.precision]type`

Each field is a single character or a number signifying a particular format option. The *type* characters that appear after the last optional format field determine whether the associated argument is interpreted as a character, string, or number. The simplest format specification contains only the percent sign and a type character (for example, `%s`). The optional fields control other aspects of the formatting. Following are the optional and required fields and their meanings:

| Field | Meaning |
|-------------------|--|
| <code>-</code> | Pad the output with blanks or zeros to the right to fill the field width, justifying output to the left. If this field is omitted, the output is padded to the left, justifying it to the right. |
| <code>#</code> | Prefix hexadecimal values with <code>0x</code> (lowercase) or <code>0X</code> (uppercase). |
| <code>0</code> | Pad the output value with zeros to fill the field width. If this field is omitted, the output value is padded with blank spaces. |
| <i>width</i> | Copy the specified minimum number of characters to the output buffer. The <i>width</i> field is a nonnegative integer. The width specification never causes a value to be truncated. If the number of characters in the output value is greater than the specified width, or if the <i>width</i> field is not present, all characters of the value are printed, subject to the precision specification. |
| <i>.precision</i> | For numbers, copy the specified minimum number of digits to the output buffer. If the number of digits in the argument is less than the specified precision, the output value is padded on the left with zeros. The value is not truncated when the number of digits exceeds the specified precision. If the specified precision is 0 or omitted entirely, or if the period (<code>.</code>) appears without a number following it, the precision is set to 1. For strings, copy the specified maximum number of characters to the output buffer. |
| <i>type</i> | Output the corresponding argument as a character, string, or number. This field can be any of the following values: |
| Value | Meaning |
| <code>c</code> | Single character. This value is interpreted as type WCHAR , if the calling application defines UNICODE, and as type CHAR , otherwise. |
| <code>C</code> | Single character. This value is interpreted as type CHAR , if the calling application defines UNICODE, and as type WCHAR , otherwise. |
| <code>d</code> | Signed decimal integer. This value is equivalent to <code>i</code> . |

| Value | Meaning |
|--------|--|
| hc, hC | Single character. The wprintf function ignores character arguments with a numeric value of zero. This value is always interpreted as type CHAR , even when the calling application defines UNICODE. |
| hd | Signed short integer argument. |
| hs, hS | String. This value is always interpreted as type LPSTR , even when the calling application defines UNICODE. |
| hu | Unsigned short integer. |
| i | Signed decimal integer. This value is equivalent to d. |
| lc, lC | Single character. The wprintf function ignores character arguments with a numeric value of zero. This value is always interpreted as type WCHAR , even when the calling application does not define UNICODE. |
| ld | Long signed integer. This value is equivalent to li. |
| li | Long signed integer. This value is equivalent to ld. |
| ls, lS | String. This value is always interpreted as type LPWSTR , even when the calling application does not define UNICODE. This value is equivalent to ws. |
| lu | Long unsigned integer. |
| lx, lX | Long unsigned hexadecimal integer in lowercase or uppercase. |
| p | Windows 2000: Pointer. The address is printed using hexadecimal. |
| s | String. This value is interpreted as type LPWSTR , when the calling application defines UNICODE, and as type LPSTR , otherwise. |
| S | String. This value is interpreted as type LPSTR , when the calling application defines UNICODE, and as type LPWSTR , otherwise. |
| u | Unsigned integer argument. |
| x, X | Unsigned hexadecimal integer in lowercase or uppercase. |

Note Unlike other Win32 functions, **wprintf** uses the C calling convention (**_cdecl**), instead of the standard call (**_stdcall**) calling convention. As a result, it is the responsibility of the calling process to pop arguments off the stack, and arguments are pushed on the stack from right to left. In C-language modules, the C compiler performs this task.

! 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

Strings Overview, String Functions, **wvsprintf**

wvsprintf

The **wvsprintf** function formats and stores a series of characters and values in a buffer. The items pointed to by the argument list are converted and copied to an output buffer, according to the corresponding format specification in the format-control string. The function appends a terminating null character to the characters it writes, but the return value does not include the terminating null character in its character count.

```
int wvsprintf(  
    LPCTSTR lpOutput, // buffer for output  
    LPCWSTR lpFormat, // format-control string  
    va_list arglist // variable list of format-control arguments  
);
```

Parameters

lpOutput

[out] Pointer to a buffer to receive the formatted output.

lpFormat

[in] Pointer to a null-terminated string that contains the format-control specifications. In addition to ordinary ASCII characters, a format specification for each argument appears in this string. For more information about the format specification, see **wsprintf**.

arglist

[in] A variable argument list; each element of the list specifies an argument for the format-control string. The number, type, and interpretation of the arguments depend on the corresponding format-control specifications in the *lpFmt* parameter.

Return Values

If the function succeeds, the return value is the number of characters stored in the buffer, not counting the terminating null character.

If the function fails, the return value is less than the length of the expected output. To get extended error information, call **GetLastError**.

Remarks

The function copies the format-control string into the output buffer character by character, starting with the first character in the string. When it encounters a format specification in the string, the function retrieves the value of the next available argument (starting with the first argument in the list), converts that value into the specified format, and copies the result to the output buffer. The function continues to copy characters and expand format specifications in this way until it reaches the end of the format-control string. If there are more arguments than format specifications, the extra arguments are ignored. If there are not enough arguments for all of the format specifications, the results are undefined.

! 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

Strings Overview, String Functions, **wsprintf**



 CHAPTER 8

User Input

Common Dialog-Box Library

The Common Dialog-Box Library contains a set of dialog boxes for performing common tasks, such as opening files and printing documents. The common dialog boxes provide a uniform user interface that lets users carry out these common tasks without being forced to learn new techniques with each application.

About Common Dialog Boxes

The common dialog boxes include the **Open** and **Save As** file dialog boxes; the **Find** and **Replace** editing dialog boxes; the **Print**, **Print Setup**, and **Page Setup** printing dialog boxes; and the **Color** and **Font** dialog boxes.

You can create common dialog boxes for your applications by using the common dialog-box functions. These functions supply the dialog-box procedures and templates for the common dialog boxes. You supply the initial values for the dialog boxes and the addresses of the variables and buffers that receive the input from the dialog boxes.

Dialog-Box Types

The Common Dialog-Box Library provides a creation function and a structure for each type of common dialog box. To use a common dialog box in its simplest form, you call its creation function and specify a pointer to a structure containing initial values and option flags. After initializing the dialog box, the dialog-box procedure uses the structure to return information about the user's input. You also can customize a common dialog box to suit the needs of your application.

The following table provides a brief description of the different types of common dialog boxes, and shows the function and structure used with each type:

| Dialog box | Description |
|--------------|--|
| Color | Displays available colors and optionally lets the user create custom colors. The user can select a basic or custom color. Use the ChooseColor function and CHOOSECOLOR structure. |
| Find | Displays a dialog box in which the user can type the string to find. The user can also specify search options, such as the search direction and whether the search is case-sensitive. Use the FindText function and FINDREPLACE structure. |

(continued)

(continued)

| Dialog box | Description |
|-------------------|---|
| Font | Displays lists of available fonts, point sizes, and other font attributes that the user can select. Use the ChooseFont function and CHOOSEFONT structure. |
| Open | Displays a dialog box in which the user can type or select the name of a file or shell name-space object to open. The dialog box includes lists of drives, directories, and shell name-space extensions that enable the user to browse the shell name-space. It also includes a list of file name extensions that enables the user to filter the file names displayed. Use the GetOpenFileName function and OPENFILENAME structure. |
| Page Setup | Displays the current page configuration. The user can select page configuration options, such as paper orientation, size, source, and margins. Use the PageSetupDlg function and PAGESETUPDLG structure. |
| Print | Displays information about the installed printer and its configuration. The user can select print job options, such as the range of pages to print and the number of copies, and start the printing process. Use the PrintDlg function and PRINTDLG structure. Windows 2000: To display a Print property sheet rather than a Print dialog box, use the PrintDlgEx function with the PRINTDLGEX structure. The General page of the property sheet is similar to the Print dialog box. The property sheet can also have additional application-specific and driver-specific property pages following the General page. |
| Replace | Displays a dialog box in which the user can type the string to find and the replacement string. The user can specify search options, such as whether the search is case-sensitive, and replacement options, such as the scope of replacement. Use the ReplaceText function and FINDREPLACE structure. |
| Save As | Displays a dialog box in which the user can type or select the name with which to save a file or shell name-space object. The dialog box includes lists of drives, directories, and shell name-space extensions that enable the user to browse the shell name space. It also includes a list of file name extensions that enables the user to filter the file names displayed. Use the GetSaveFileName function and OPENFILENAME structure. |

Note Although a **Print Setup** dialog box also is available, it has been superseded by the **Page Setup** dialog box. Applications written for Windows 95/98 or for Windows NT versions 3.51 or later should use the **Page Setup** dialog box, instead of the **Print Setup** dialog box.

All common dialog boxes are modal, except the **Find** and **Replace** dialog boxes. Modal dialog boxes must be closed by the user before the function used to create the dialog box can return. The **Find** and **Replace** dialog boxes are modeless; the function returns before the dialog box closes. If you use the **Find** and **Replace** dialog boxes, you must also use the **IsDialogMessage** function in the main message loop of your application to ensure that these dialog boxes correctly process keyboard input, such as the TAB and ESC keys.

Getting More Information About Common Dialog Boxes

The companion CD that is bundled inside the Base Services volume of the *Microsoft Win32 Developer's Reference Library* has the complete set of reference information for Common Dialog Boxes. Publishing constraints associated with volumes in the Windows Programming Reference Series—which are governed by the mission to provide concise, compact, and portable reference books—did not allow all of the overview and reference information about Common Dialog Boxes to be included in the printed version. I've included an overview in this printed version to provide you with some degree of familiarity with them, and more importantly, to alert you to their existence (in case you did not know already).

In order to provide you with the most complete and comprehensive guide to Win32 development, the Win32 Library includes the complete set of information pertaining to Common Dialog Boxes in electronic form on the DVD. If you have not already, go through the installation process on the companion DVD, and everything you want to know about Common Dialog Boxes (and much, much more) will be a click away.

Mouse Input

About Mouse Input

The mouse is an important, but optional, user-input device for Win32-based applications. A well-written Win32-based application should include a mouse interface, but it should not depend solely on the mouse for acquiring user input. The application should provide full keyboard support as well.

A Win32-based application receives mouse-input in the form of messages that are sent or posted to its windows.

Additional overview information about Mouse Input, including double-click messages and how to detect and work with a mouse wheel, can be found on the companion DVD located in the back of the Base Services volume.

Mouse-Input Reference

Mouse-Input Functions

DragDetect

The **DragDetect** function captures the mouse and tracks its movement until the user releases the left button, presses the ESC key, or moves the mouse outside the drag rectangle around the specified point. The width and height of the drag rectangle are specified by the SM_CXDRAG and SM_CYDRAG values returned by the **GetSystemMetrics** function.

```
BOOL DragDetect(  
    HWND hwnd, // handle to window  
    POINT pt   // initial position  
);
```

Parameters

hwnd

[in] Handle to the window receiving mouse input.

pt

[in] Initial position of the mouse, in screen coordinates. The function determines the coordinates of the drag rectangle by using this point.

Return Values

If the user moved the mouse outside of the drag rectangle while holding down the left button, the return value is nonzero.

If the user did not move the mouse outside of the drag rectangle while holding down the left button, the return value is zero.

Remarks

The system metrics for the drag rectangle are configurable, allowing for larger or smaller drag rectangles.

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

Mouse Input Overview, Mouse-Input Functions, **GetSystemMetrics**

GetCapture

The **GetCapture** function retrieves a handle to the window (if any) that has captured the mouse. Only one window at a time can capture the mouse; this window receives mouse input whether or not the cursor is within its borders.

HWND **GetCapture(VOID)**;

Parameters

This function has no parameters.

Return Values

The return value is a handle to the capture window associated with the current thread. If no window in the thread has captured the mouse, the return value is NULL.

Remarks

A NULL return value means the current thread has not captured the mouse. However, it is possible that another thread or process has captured the mouse.

Windows 98 and Windows NT 4.0 SP3 and later: To get a handle to the capture window on another thread, use the **GetGuiThreadInfo** 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

Mouse Input Overview, Mouse-Input Functions, **GetGuiThreadInfo**, **ReleaseCapture**, **SetCapture**

GetDoubleClickTime

The **GetDoubleClickTime** function retrieves the current double-click time for the mouse. A double-click is a series of two clicks of the mouse button, the second occurring within a

specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second click of a double-click.

UINT GetDoubleClickTime(VOID):

Parameters

This function has no parameters.

Return Values

The return value specifies the current double-click time, in milliseconds.

! 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

Mouse Input Overview, Mouse-Input Functions, **SetDoubleClickTime**

GetMouseMovePointsEx

The **GetMouseMovePointsEx** function retrieves a history of up to 64 previous coordinates of the mouse or pen.

```
int GetMouseMovePointsEx(
    UINT cbSize           // size of mouse move point
    LPMOUSEMOVEPOINT lppt, // current mouse move point
    LPMOUSEMOVEPOINT lpptBuf, // buffer to store points
    int nBufPoints,       // points the buffer can store
    DWORD resolution     // resolution of the points
);
```

Parameters

cbSize

[in] Specifies the size, in bytes, of the **MOUSEMOVEPOINT** structure.

lppt

[in] Pointer to a **MOUSEMOVEPOINT** structure containing valid mouse coordinates (in screen coordinates). It may contain also a time stamp.

The **GetMouseMovePointsEx** function searches for the point in the mouse-coordinates history. If the function finds the point, it returns the last *nBufPoints* prior to and including the supplied point.

If your application supplies a time stamp, the **GetMouseMovePointsEx** function will use it to differentiate between two equal points that were recorded at different times. An application should call this function using the mouse coordinates received from the **WM_MOUSEMOVE** message and convert them to screen coordinates.

lpptBuf

[in] Pointer to a buffer that will receive the points. It should be at least $cbSize * nBufPoints$ in size.

nBufPoints

[in] Specifies the number of points to retrieve.

resolution

[in] Specifies the resolution desired. This parameter can be one of the following values:

| Value | Meaning |
|---------------------------------|--|
| GMMP_USE_DISPLAY_POINTS | Retrieves the points using the display resolution. |
| GMMP_USE_HIGH_RESOLUTION_POINTS | Retrieves high resolution points. Points can range from zero to 65,535 (0xFFFF) in both x- and y-coordinates. This is the resolution provided by absolute coordinate pointing devices such as drawing tablets. |

Return Values

If the function succeeds, the return value is the number of points in the buffer. Otherwise, the function returns -1 . For extended error information, your application can call **GetLastError**. The **GetLastError** function can return the following error code:

| Value | Meaning |
|--------------------------|---|
| GMMP_ERR_POINT_NOT_FOUND | The point specified by <i>lppt</i> could not be found or is no longer in the system buffer. |

Remarks

The system retains the last 64 mouse coordinates and their time stamps. If your application supplies a mouse coordinate to **GetMouseMovePointsEx** and the coordinate exists in the system's mouse-coordinate history, the function retrieves the specified number of coordinates from the system's history. You also can supply a time stamp, which will be used to differentiate between identical points in the history.

The **GetMouseMovePointsEx** function will return points that eventually were dispatched not only to the calling thread but also to other threads.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Requires version 2.0 or later.

Header: Declared in winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Mouse Input Overview, Mouse-Input Functions, **MOUSEMOVEPOINT**

mouse_event

The **mouse_event** function synthesizes mouse motion and button clicks.

Windows NT/2000: This function has been superseded. Use **SendInput** instead.

```
VOID mouse_event(
    DWORD dwFlags,           // motion and click options
    DWORD dx,                // horizontal position or change
    DWORD dy,                // vertical position or change
    DWORD dwData,           // wheel movement
    ULONG_PTR dwExtraInfo // application-defined information
);
```

Parameters

dwFlags

[in] Specifies various aspects of mouse motion and button clicking. This parameter can be certain combinations of the following values:

| Value | Meaning |
|------------------------|--|
| MOUSEEVENTF_ABSOLUTE | Specifies that the <i>dx</i> and <i>dy</i> parameters contain normalized absolute coordinates. If not set, those parameters contain relative data: the change in position since the last reported position. This flag can be set, or not set, regardless of what kind of mouse or mouse-like device, if any, is connected to the system. For further information about relative mouse motion, see the following Remarks section. |
| MOUSEEVENTF_LEFTDOWN | Specifies that the left button is down. |
| MOUSEEVENTF_LEFTUP | Specifies that the left button is up. |
| MOUSEEVENTF_MIDDLEDOWN | Specifies that the middle button is down. |
| MOUSEEVENTF_MIDDLEUP | Specifies that the middle button is up. |

| Value | Meaning |
|-----------------------|---|
| MOUSEEVENTF_MOVE | Specifies that movement occurred. |
| MOUSEEVENTF_RIGHTDOWN | Specifies that the right button is down. |
| MOUSEEVENTF_RIGHTUP | Specifies that the right button is up. |
| MOUSEEVENTF_WHEEL | Windows NT/2000: Specifies that the wheel has been moved, if the mouse has a wheel. The amount of movement is specified in <i>dwData</i> |
| MOUSEEVENTF_XDOWN | Windows 2000: Specifies that an X button was pressed. |
| MOUSEEVENTF_XUP | Windows 2000: Specifies that an X button was released. |

The values that specify mouse button status are set to indicate changes in status, not ongoing conditions. For example, if the left mouse button is pressed and held down, `MOUSEEVENTF_LEFTDOWN` is set when the left button is first pressed, but not for subsequent motions. Similarly, `MOUSEEVENTF_LEFTUP` is set only when the button is first released.

dx

[in] Specifies the mouse's absolute position along the x-axis or its amount of motion since the last mouse event was generated, depending on the setting of `MOUSEEVENTF_ABSOLUTE`. Absolute data is specified as the mouse's actual x-coordinate; relative data is specified as the number of mickeys moved. A *mickey* is the amount that a mouse has to move for it to report that it has moved.

dy

[in] Specifies the mouse's absolute position along the y-axis or its amount of motion since the last mouse event was generated, depending on the setting of `MOUSEEVENTF_ABSOLUTE`. Absolute data is specified as the mouse's actual y-coordinate; relative data is specified as the number of mickeys moved.

dwData

[in] If *dwFlags* contains `MOUSEEVENTF_WHEEL`, then *dwData* specifies the amount of wheel movement. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user. One wheel click is defined as `WHEEL_DELTA`, which is 120.

Windows 2000: If *dwFlags* contains `MOUSEEVENTF_XDOWN` or `MOUSEEVENTF_XUP`, then *dwData* specifies which X buttons were pressed or released. This value may be any combination of the following flags:

| Value | Meaning |
|----------|---|
| XBUTTON1 | Set if the first X button was pressed or released. |
| XBUTTON2 | Set if the second X button was pressed or released. |

If *dwFlags* is not `MOUSEEVENTF_WHEEL`, `MOUSEEVENTF_XDOWN`, or `MOUSEEVENTF_XUP`, then *dwData* should be zero.

dwExtraInfo

[in] Specifies an additional value associated with the mouse event. An application calls **GetMessageExtraInfo** to obtain this extra information.

Return Values

This function has no return value.

Remarks

If the mouse has moved, indicated by `MOUSEEVENTF_MOVE` being set, *dx* and *dy* hold information about that motion. The information is specified as absolute or relative integer values.

If `MOUSEEVENTF_ABSOLUTE` value is specified, *dx* and *dy* contain normalized absolute coordinates between 0 and 65,535. The event procedure maps these coordinates onto the display surface. Coordinate (0,0) maps onto the upper-left corner of the display surface, (65535,65535) maps onto the lower-right corner.

If the `MOUSEEVENTF_ABSOLUTE` value is not specified, *dx* and *dy* specify relative motions from when the last mouse event was generated (the last reported position). Positive values mean the mouse moved right (or down); negative values mean the mouse moved left (or up).

Relative mouse motion is subject to the settings for mouse speed and acceleration level. An end user sets these values using the Mouse application in Control Panel. An application obtains and sets these values with the **SystemParametersInfo** function.

The system applies two tests to the specified relative mouse motion when applying acceleration. If the specified distance along either the x or y axis is greater than the first mouse threshold value, and the mouse acceleration level is not zero, the operating system doubles the distance. If the specified distance along either the x- or y-axis is greater than the second mouse threshold value, and the mouse acceleration level is equal to two, the operating system doubles the distance that resulted from applying the first threshold test. It is thus possible for the operating system to multiply relatively-specified mouse motion along the x- or y-axis by up to four times.

Once acceleration has been applied, the system scales the resultant value by the desired mouse speed. Mouse speed can range from 1 (slowest) to 20 (fastest) and represents how much the pointer moves based on the distance the mouse moves. The default value is 10, which results in no additional modification to the mouse motion.

The **mouse_event** function is used to synthesize mouse events by applications that need to do so. It is used also by applications that need to obtain more information from the mouse than its position and button state. For example, if a tablet manufacturer wants to pass pen-based information to its own applications, it can write a dynamic-link library (DLL) that communicates directly to the tablet hardware, obtains the extra information, and saves it in a queue. The DLL then calls **mouse_event** with the standard button and x/y position data, along with, in the *dwExtraInfo* parameter, some pointer or index to the queued extra information. When the application needs the extra information, it calls the

DLL with the pointer or index stored in *dwExtraInfo*, and the DLL returns the extra information.

! 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*.

Library: Use *user32.lib*.

+ See Also

Mouse Input Overview, Mouse-Input Functions, **GetMessageExtraInfo**, **SystemParametersInfo**

ReleaseCapture

The **ReleaseCapture** function releases the mouse capture from a window in the current thread and restores normal mouse-input processing. A window that has captured the mouse receives all mouse input, regardless of the position of the cursor, except when a mouse button is clicked while the cursor hot spot is in the window of another thread.

BOOL ReleaseCapture(VOID);

Parameters

This function has no parameters.

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 calls this function after calling the **SetCapture** function.

Windows 95: Calling **ReleaseCapture** causes the window that is losing the mouse capture to receive a **WM_CAPTURECHANGED** message.

! 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

Mouse Input Overview, Mouse-Input Functions, **GetCapture**, **SetCapture**, **WM_CAPTURECHANGED**

SetCapture

The **SetCapture** function sets the mouse capture to the specified window belonging to the current thread. Once a window has captured the mouse, all mouse input is directed to that window, regardless of whether the cursor is within the borders of that window. Only one window at a time can capture the mouse.

If the mouse cursor is over a window created by another thread, the system will direct mouse input to the specified window only if a mouse button is down.

```
HWND SetCapture(  
    HWND hWnd // handle to window  
);
```

Parameters

hWnd

[in] Handle to the window in the current thread that is to capture the mouse.

Return Values

The return value is a handle to the window that had previously captured the mouse. If there is no such window, the return value is NULL.

Remarks

Only the foreground window can capture the mouse. When a background window attempts to do so, the window receives messages only for mouse events that occur when the cursor hot spot is within the visible portion of the window. Also, even if the foreground window has captured the mouse, the user can still click another window, bringing it to the foreground.

When the window no longer requires all mouse input, the thread that created the window should call the **ReleaseCapture** function to release the mouse.

This function cannot be used to capture mouse input meant for another process.

When the mouse is captured, menu hotkeys and other keyboard accelerators do not work.

Windows 95: Calling **SetCapture** causes the window that is losing the mouse capture to receive a **WM_CAPTURECHANGED** message.

! 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

Mouse Input Overview, Mouse-Input Functions, **GetCapture**, **ReleaseCapture**, **WM_CAPTURECHANGED**

SetDoubleClickTime

The **SetDoubleClickTime** function sets the double-click time for the mouse. A double-click is a series of two clicks of a mouse button, the second occurring within a specified time after the first. The double-click time is the maximum number of milliseconds that may occur between the first and second clicks of a double-click.

```
BOOL SetDoubleClickTime(
    UINT uInterval // double-click interval
);
```

Parameters

uInterval

[in] Specifies the number of milliseconds that may occur between the first and second clicks of a double-click. If this parameter is set to zero, the system uses the default double-click time of 500 milliseconds.

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 **SetDoubleClickTime** function alters the double-click time for all windows in the system.

! 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

Mouse Input Overview, Mouse-Input Functions, **GetDoubleClickTime**

SwapMouseButton

The **SwapMouseButton** function reverses or restores the meaning of the left and right mouse buttons.

```
BOOL SwapMouseButton(  
    BOOL fSwap // reverse or restore buttons  
);
```

Parameters

fSwap

[in] Specifies whether the mouse button meanings are reversed or restored. If this parameter is TRUE, the left button generates right-button messages and the right button generates left-button messages. If this parameter is FALSE, the buttons are restored to their original meanings.

Return Values

If the meaning of the mouse buttons was reversed previously, before the function was called, the return value is nonzero.

If the meaning of the mouse buttons was not reversed, the return value is zero.

Remarks

Button swapping is provided as a convenience to people who use the mouse with their left hands. The **SwapMouseButton** function usually is called by Control Panel only. Although an application is free to call the function, the mouse is a shared resource and reversing the meaning of its buttons affects all applications.

! 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

Mouse Input Overview, Mouse-Input Functions, **SetDoubleClickTime**

TrackMouseEvent

The **TrackMouseEvent** function posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

```
BOOL TrackMouseEvent(
    LPTRACKMOUSEEVENT lpEventTrack // tracking information
);
```

Parameters

lpEventTrack

[in/out] Pointer to a **TRACKMOUSEEVENT** structure that contains tracking information.

Return Values

If the function succeeds, the return value is nonzero .

If the function fails, return value is zero. To get extended error information, call **GetLastError**.

The function can post the following messages:

| Message | Meaning |
|------------------------|--|
| WM_MOUSEHOVER | The mouse hovered over the client area of the window for the period of time specified in a prior call to TrackMouseEvent . Hover tracking stops when this message is generated. The application must call TrackMouseEvent again if it requires further tracking of mouse-hover behavior. |
| WM_MOUSELEAVE | The mouse left the client area of the window specified in a prior call to TrackMouseEvent . All tracking requested by TrackMouseEvent is canceled when this message is generated. The application must call TrackMouseEvent when the mouse reenters its window if it requires further tracking of mouse-hover behavior. |
| WM_NCMOUSEHOVER | Windows 98, Windows 2000: The same meaning as WM_MOUSEHOVER , except this is for the nonclient area of the window. |
| WM_NCMOUSELEAVE | Windows 98, Windows 2000: The same meaning as WM_MOUSELEAVE , except this is for the nonclient area of the window. |

Remarks

The mouse pointer is considered to be hovering when it stays within a specified rectangle for a specified period of time. Call **SystemParametersInfo** and use the values

SPI_GETMOUSEHOVERWIDTH, SPI_GETMOUSEHOVERHEIGHT, and SPI_GETMOUSEHOVERTIME to retrieve the size of the rectangle and the time.

Note The `_TrackMouseEvent` function calls `TrackMouseEvent` if it exists, otherwise `_TrackMouseEvent` emulates `TrackMouseEvent`. The `_TrackMouseEvent` function is in `commctrl.h` and is exported by `COMCTRL32.DLL`.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

See Also

Mouse Input Overview, Mouse-Input Functions, `SystemParametersInfo`, `_TrackMouseEvent`, `TRACKMOUSEEVENT`

`_TrackMouseEvent`

The `_TrackMouseEvent` function posts messages when the mouse pointer leaves a window or hovers over a window for a specified amount of time. This function calls `TrackMouseEvent`, if it exists; otherwise, it emulates it.

Requirements

Windows NT/2000: Requires Windows 2000 (or Windows NT 4.0 with Internet Explorer 3.0 or later).

Windows 95/98: Requires Windows 98 (or Windows 95 with Internet Explorer 3.0 or later).

Windows CE: Unsupported.

Header: Declared in `commctrl.h`.

Library: Use `comctl32.lib`.

See Also

Mouse Input Overview, Mouse-Input Functions, `SystemParametersInfo`, `TrackMouseEvent`, `TRACKMOUSEEVENT`

Mouse-Input Structures

MOUSEMOVEPOINT

The **MOUSEMOVEPOINT** structure contains information about the mouse's location in screen coordinates.

```
typedef struct tagMOUSEMOVEPOINT {
    int x;
    int y;
    DWORD time;
    ULONG_PTR dwExtraInfo;
} MOUSEMOVEPOINT, *PMOUSEMOVEPOINT;
```

Members

x

Specifies the x-coordinate of the mouse.

y

Specifies the y-coordinate of the mouse.

time

Specifies the time stamp of the mouse coordinate.

dwExtraInfo

Specifies extra information associated with this coordinate.

! 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

Mouse Input Overview, Mouse-Input Structures, **GetMouseMovePointsEx**

TRACKMOUSEEVENT

The **TRACKMOUSEEVENT** structure is used by the **TrackMouseEvent** function to track when the mouse pointer leaves a window or hovers over a window for a specified amount of time.

```
typedef struct tagTRACKMOUSEEVENT {
    DWORD cbSize;
```

(continued)

(continued)

```

DWORD dwFlags;
HWND hwndTrack;
DWORD dwHoverTime;
} TRACKMOUSEEVENT, *LPTRACKMOUSEEVENT;

```

Members

cbSize

Specifies the size of the **TRACKMOUSEEVENT** structure.

dwFlags

Specifies the services requested. This member can be a combination of the following values:

| Value | Meaning |
|---------------|--|
| TME_CANCEL | The caller wants to cancel a prior tracking request. The caller should also specify the type of tracking that it wants to cancel. For example, to cancel hover tracking, the caller must pass the TME_CANCEL and TME_HOVER flags. |
| TME_HOVER | The caller wants hover notification. Notification is delivered as a WM_MOUSEHOVER message. If the caller requests hover tracking while hover tracking is already active, the hover timer will be reset. This flag is ignored if the mouse pointer is not over the specified window or area. |
| TME_LEAVE | The caller wants leave notification. Notification is delivered as a WM_MOUSELEAVE message. If the mouse is not over the specified window or area, a leave notification is generated immediately, and no further tracking is performed. |
| TME_NONCLIENT | Windows 98, Windows 2000: The caller wants hover and leave notification for the nonclient areas. Notification is delivered as WM_NCMOUSEHOVER and WM_NCMOUSELEAVE messages. |
| TME_QUERY | The function fills in the structure instead of treating it as a tracking request. The structure is filled such that had that structure been passed to TrackMouseEvent , it would generate the current tracking. The only anomaly is that the hover time-out returned is always the actual time-out and not HOVER_DEFAULT , if HOVER_DEFAULT was specified during the original TrackMouseEvent request. |

hwndTrack

Specifies a handle to the window to track.

dwHoverTime

Specifies the hover time-out (if `TME_HOVER` was specified in `dwFlags`), in milliseconds. Can be `HOVER_DEFAULT`, which means to use the system default hover time-out.

Remarks

The system default hover time-out is initially the menu drop-down time, which is 400 milliseconds. You can call **SystemParametersInfo** and use `SPI_GETMOUSEHOVERTIME` to retrieve the default hover time-out.

The system default hover rectangle is the same as the double-click rectangle. You can call **SystemParametersInfo** and use `SPI_GETMOUSEHOVERWIDTH` and `SPI_GETMOUSEHOVERHEIGHT` to retrieve the size of the rectangle within which the mouse pointer has to stay for **TrackMouseEvent** to generate a **WM_MOUSEHOVER** message.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Structures, **SystemParametersInfo**, **TrackMouseEvent**

Mouse-Input Messages

WM_APPCOMMAND

The **WM_APPCOMMAND** message notifies a window that the user generated an application command event, for example, by clicking an application command button using the mouse or typing an application command key on the keyboard.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_APPCOMMAND  
    WPARAM wParam,      // handle to window (DWORD)  
    LPARAM lParam        // command, device, and virtual keys  
);
```

Parameters

wParam

Handle to the window where the user clicked the button or pressed the key. This can be a child window of the window receiving the message. For more information about processing this message, see the Remarks section.

lParam

Use the following code to crack the *lParam* parameter:

```
cmd = GET_APPCOMMAND_LPARAM(lParam);
uDevice = GET_DEVICE_LPARAM(lParam);
dwKeys = GET_KEYSTATE_LPARAM(lParam);
```

Where *cmd* indicates the application command. This parameter can be one of the following values:

| Value | Meaning |
|--------------------------------|-----------------------------------|
| APPCOMMAND_BASS_BOOST | Toggle the bass boost on and off. |
| APPCOMMAND_BASS_DOWN | Decrease the bass. |
| APPCOMMAND_BASS_UP | Increase the bass. |
| APPCOMMAND_BROWSER_BACKWARD | Move backward. |
| APPCOMMAND_BROWSER_FAVORITES | Open favorites. |
| APPCOMMAND_BROWSER_FORWARD | Move forward. |
| APPCOMMAND_BROWSER_HOME | Move home. |
| APPCOMMAND_BROWSER_REFRESH | Refresh page. |
| APPCOMMAND_BROWSER_SEARCH | Open search. |
| APPCOMMAND_BROWSER_STOP | Stop download. |
| APPCOMMAND_LAUNCH_APP1 | Start App1. |
| APPCOMMAND_LAUNCH_APP2 | Start App2. |
| APPCOMMAND_LAUNCH_MAIL | Open mail. |
| APPCOMMAND_MEDIA_NEXTTRACK | Go to next track. |
| APPCOMMAND_MEDIA_PLAY_PAUSE | Play or pause playback. |
| APPCOMMAND_MEDIA_PREVIOUSTRACK | Go to previous track. |
| APPCOMMAND_MEDIA_SELECT | Go to Media Select mode. |
| APPCOMMAND_MEDIA_STOP | Stop playback. |
| APPCOMMAND_TREBLE_DOWN | Decrease the treble. |
| APPCOMMAND_TREBLE_UP | Increase the treble. |
| APPCOMMAND_VOLUME_DOWN | Lower the volume. |
| APPCOMMAND_VOLUME_MUTE | Mute the volume. |
| APPCOMMAND_VOLUME_UP | Raise the volume. |

Where *uDevice* indicates the input device that generated the input event. It can be one of the following values:

| Value | Meaning |
|-------------------|---|
| FAPPCOMMAND_KEY | User pressed a key. |
| FAPPCOMMAND_MOUSE | User clicked a mouse button. |
| FAPPCOMMAND_OEM | An unidentified hardware source generated the event. It could be a mouse or a keyboard event. |

Where *dwKeys* indicates whether various virtual keys are down. It can be one or more of the following values:

| Value | Meaning |
|-------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | The first X button is down. |
| MK_XBUTTON2 | The second X button is down. |

Return Values

If an application processes this message, it should return TRUE. For more information about processing the return value, see the Remarks section.

Remarks

DefWindowProc generates the **WM_APPCOMMAND** message when it processes the **WM_XBUTTONDOWN** or **WM_NCXBUTTONDOWN** message, or when the user types an application command key.

If a child window does not process this message and instead calls **DefWindowProc**, **DefWindowProc** will send the message to its parent window. If a top level window does not process this message and instead calls **DefWindowProc**, **DefWindowProc** will call a shell hook with the hook code equal to **HSHELL_APPCOMMAND**.

To get the coordinates of the cursor if the message was generated by a button click on the mouse, the application can call **GetMessagePos**. An application can test whether the message was generated by the mouse by checking whether *lParam* contains **FAPPCOMMAND_MOUSE**.

Unlike other windows messages, an application should return TRUE from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_APPCOMMAND_LPARAM**, **GET_DEVICE_LPARAM**, **GET_KEYSTATE_LPARAM**, **ShellProc**, **WM_NCXBUTTONUP**, **WM_XBUTTONUP**

WM_CAPTURECHANGED

The **WM_CAPTURECHANGED** message is sent to the window that is losing the mouse capture.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_CAPTURECHANGED
    WPARAM wParam,       // not used
    LPARAM lParam        // handle to window (HWND)
);
```

Parameters

wParam

This parameter is not used.

lParam

Handle to the window gaining the mouse capture.

Return Values

An application should return zero if it processes this message.

Remarks

A window receives this message even if it calls **ReleaseCapture** itself. An application should not attempt to set the mouse capture in response to this message.

When it receives this message, a window should redraw itself, if necessary, to reflect the new mouse-capture state.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 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

Mouse Input Overview, Mouse-Input Messages, **ReleaseCapture**, **SetCapture**

WM_LBUTTONDOWNBLCLK

The **WM_LBUTTONDOWNBLCLK** message is posted when the user double-clicks the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_LBUTTONDOWNBLCLK
    WPARAM wParam,      // key indicator
    LPARAM lParam       // horizontal and vertical position
);

```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

Only windows that have the **CS_DBLCLKS** style can receive **WM_LBUTTONDOWNBLCLK** messages, which the system generates whenever the user presses, releases, and again presses the left mouse button within the system's double-click time limit. Double-clicking the left mouse button actually generates a sequence of four messages: **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_LBUTTONDOWNBLCLK**, and **WM_LBUTTONUP**.

! 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

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **GetDoubleClickTime**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **SetDoubleClickTime**, **WM_LBUTTONDOWN**, **WM_LBUTTONUP**

WM_LBUTTONDOWN

The **WM_LBUTTONDOWN** message is posted when the user presses the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_LBUTTONDOWN
```

```

    WPARAM wParam, // key indicator
    LPARAM lParam // horizontal and vertical position
);

```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-----------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTONDOWN | Windows 2000: The first X button is down. |
| MK_XBUTTONDOWN2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```

xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);

```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

To detect that the ALT key was pressed, check whether **GetKeyState**(VK_MENU) < 0. Note that this must not be **GetAsyncKeyState**.

! 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

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_LBUTTONDOWNCLK**, **WM_LBUTTONUP**

WM_LBUTTONUP

The **WM_LBUTTONUP** message is posted when the user releases the left mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_LBUTTONUP
    WPARAM wParam,      // key indicator
    LPARAM lParam       // horizontal and vertical position
);
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter 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: Requires version 1.0 or later.

Header: Declared in `winuser.h`; include `windows.h`.

See Also

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_LBUTTONDOWNBLCLK**, **WM_LBUTTONDOWN**

WM_MBUTTONDOWNBLCLK

The **WM_MBUTTONDOWNBLCLK** message is posted when the user double-clicks the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MBUTTONDOWNBLCLK
    WPARAM wParam,      // key indicator
    LPARAM lParam       // horizontal and vertical position
);
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|------------|--------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |

(continued)

(continued)

| Value | Description |
|-------------|---|
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

IParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *IParam* parameter to a **POINTS** structure.

Only windows that have the CS_DBLCLKS style can receive **WM_MBUTTONDOWNBLCLK** messages, which the system generates when the user presses, releases, and again presses the middle mouse button within the system's double-click time limit. Double-clicking the middle mouse button actually generates four messages:

WM_MBUTTONDOWN, **WM_MBUTTONUP**, **WM_MBUTTONDOWNBLCLK**, and **WM_MBUTTONUP** again.

! 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

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **GetDoubleClickTime**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **SetDoubleClickTime**, **WM_MBUTTONDOWN**, **WM_MBUTTONUP**

WM_MBUTTONDOWN

The **WM_MBUTTONDOWN** message is posted when the user presses the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_MBUTTONDOWN
    WPARAM wParam,     // key indicator
    LPARAM lParam      // horizontal and vertical position
);

```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```

xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);

```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

To detect that the ALT key was pressed, check whether **GetKeyState(VK_MENU) < 0**. Note that this must not be **GetAsyncKeyState**.

! 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

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_MBUTTONDOWN**, **WM_MBUTTONUP**

WM_MBUTTONUP

The **WM_MBUTTONUP** message is posted when the user releases the middle mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_MBUTTONUP
    WPARAM wParam,       // key indicator
    LPARAM lParam        // horizontal and vertical position
);
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|------------|---------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |

| Value | Description |
|-------------|---|
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

IParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *IParam* parameter 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 winuser.h; include windows.h.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_MBUTTONDOWNBLCLK**, **WM_MBUTTONDOWN**

WM_MOUSEACTIVATE

The **WM_MOUSEACTIVATE** message is sent when the cursor is in an inactive window and the user presses a mouse button. The parent window receives this message only if the child window passes it to the **DefWindowProc** function.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MOUSEACTIVATE
    WPARAM wParam,      // handle to parent (HWND)
```

(continued)

(continued)

```
LPARAM lParam // hit-test value and message
);
```

Parameters

wParam

Handle to the top-level parent window of the window being activated.

lParam

The low-order word specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word specifies the identifier of the mouse message generated when the user pressed a mouse button. The mouse message is either discarded or posted to the window, depending on the return value.

Return Values

The return value specifies whether the window should be activated and whether the identifier of the mouse message should be discarded. It must be one of the following values:

| Value | Meaning |
|---------------------|---|
| MA_ACTIVATE | Activates the window, and does not discard the mouse message. |
| MA_ACTIVATEANDEAT | Activates the window, and discards the mouse message. |
| MA_NOACTIVATE | Does not activate the window, and does not discard the mouse message. |
| MA_NOACTIVATEANDEAT | Does not activate the window, but discards the mouse message. |

Remarks

The **DefWindowProc** function passes the message to a child window's parent window before any processing occurs. The parent window determines whether to activate the child window. If it activates the child window, the parent window should return **MA_NOACTIVATE** or **MA_NOACTIVATEANDEAT** to prevent the system from processing the message further.

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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **HIWORD**, **LOWORD**, **WM_NCHITTEST**

WM_MOUSEHOVER

The **WM_MOUSEHOVER** message is posted to a window when the cursor hovers over the client area of the window for the period of time specified in a prior call to **TrackMouseEvent**.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_MOUSEHOVER
    WPARAM wParam,       // key indicator
    LPARAM lParam        // horizontal and vertical position
);

```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-------------|---|
| MK_CONTROL | The CTRL key is depressed. |
| MK_LBUTTON | The left mouse button is depressed. |
| MK_MBUTTON | The middle mouse button is depressed. |
| MK_RBUTTON | The right mouse button is depressed. |
| MK_SHIFT | The SHIFT key is depressed. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Hover tracking stops when **WM_MOUSEHOVER** is generated. The application must call **TrackMouseEvent** again if it requires further tracking of mouse hover behavior.

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **TrackMouseEvent**, **TRACKMOUSEEVENT**, **WM_MOUSEHOVER**

WM_MOUSELEAVE

The **WM_MOUSELEAVE** message is posted to a window when the cursor leaves the client area of the window specified in a prior call to **TrackMouseEvent**.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MOUSELEAVE
    WPARAM wParam,     // not used
    LPARAM lParam       // not used
);
```

Parameters

This message has no parameters.

Return Values

If an application processes this message, it should return zero.

Remarks

All tracking requested by **TrackMouseEvent** is canceled when this message is generated. The application must call **TrackMouseEvent** when the mouse reenters its window if it requires further tracking of mouse hover behavior.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GetCapture**, **SetCapture**, **TrackMouseEvent**, **TRACKMOUSEEVENT**, **WM_NCMOUSELEAVE**

WM_MOUSEMOVE

The **WM_MOUSEMOVE** message is posted to a window when the cursor moves. If the mouse is not captured, the message is posted to the window that contains the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_MOUSEMOVE
    WPARAM wParam,      // key indicators
    LPARAM lParam       // horizontal and vertical position
);

```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |

(continued)

(continued)

| Value | Description |
|-------------|---|
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter 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: Requires version 1.0 or later.

Header: Declared in winuser.h; include windows.h.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**

WM_MOUSEWHEEL

The **WM_MOUSEWHEEL** message is sent to the focus window when the mouse wheel is rotated. The **DefWindowProc** function propagates the message to the window's parent. There should be no internal forwarding of the message, since **DefWindowProc** propagates it up the parent chain until it finds a window that processes it.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_MOUSEWHEEL
    WPARAM wParam,     // key indicator and wheel rotation
```

```
LPARAM lParam // horizontal and vertical position
);
```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

The high-order word indicates the distance the wheel is rotated, expressed in multiples or divisions of WHEEL_DELTA, which is 120. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user.

lParam

The low-order word specifies the x-coordinate of the pointer, relative to the upper-left corner of the screen.

The high-order word specifies the y-coordinate of the pointer, relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to crack the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM(wParam);
zDelta = GET_WHEEL_DELTA_WPARAM(wParam);
```

Use the following code to obtain the horizontal and vertical positions:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

The wheel rotation will be a multiple of `WHEEL_DELTA`, which is set at 120. This is the threshold for action to be taken, and one such action (for example, scrolling one increment) should occur for each delta.

The delta was set to 120 to allow Microsoft or other vendors to build finer-resolution wheels in the future, including perhaps a freely rotating wheel with no notches. The expectation is that such a device would send more messages per rotation, but with a smaller value in each message. To support this possibility, you should either add the incoming delta values until `WHEEL_DELTA` is reached (so for a delta-rotation you get the same response), or scroll partial lines in response to the more frequent messages. You could choose also your scroll granularity, and accumulate deltas until it is reached.

Windows 95 and Windows NT 3.51: Support for the mouse wheel is provided through a separately running module, `MSWheel`, that generates a `MSH_MOUSEWHEEL` message. The `MSWheel` module, which consists of `MSWheel.exe` and `MSWheel.dll`, is installed with the IntelliPoint software that is shipped with the IntelliMouse pointing device. In addition, `MSH_MOUSEWHEEL` is defined in the header file (`ZMouse.h`) that an application must use to implement support for the wheel via the `MSWheel` module.

```
MSH_MOUSEWHEEL
zDelta = (int) wParam; // wheel rotation
xPos = LOWORD(lParam); // horizontal position of pointer
yPos = HIWORD(lParam); // vertical position of pointer
```

Note There are no *fwKeys* for `MSH_MOUSEWHEEL`. Otherwise, the parameters are exactly the same as for `WM_MOUSEWHEEL`.

It is up to the application to forward `MSH_MOUSEWHEEL` to any embedded objects or controls. The application is required to send the message to an active embedded OLE application. It is optional that the application sends it to a wheel-enabled control with focus. If the application does send the message to a control, it can check the return value to see if the message was processed. Controls are required to return a value of `TRUE` if they process the message.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

See Also

Mouse Input Overview, Mouse-Input Messages, `GET_KEYSTATE_WPARAM`, `GET_WHEEL_DELTA_WPARAM`, `GET_X_LPARAM`, `GET_Y_LPARAM`, `GetSystemMetrics`, `HIWORD`, `LOWORD`, `mouse_event`, `SystemParametersInfo`

WM_NCHITTEST

The **WM_NCHITTEST** message is sent to a window when the cursor moves, or when a mouse button is pressed or released. If the mouse is not captured, the message is sent to the window beneath the cursor. Otherwise, the message is sent to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_NCHITTEST
    WPARAM wParam,     // not used
    LPARAM lParam      // horizontal and vertical position
);
```

Parameters

wParam

This parameter is not used.

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the screen.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the screen.

Return Values

The return value of the **DefWindowProc** function is one of the following values, indicating the position of the cursor hot spot:

| Value | Location of hot spot |
|---------------|--|
| HTBORDER | In the border of a window that does not have a sizing border |
| HTBOTTOM | In the lower-horizontal border of a resizable window (the user can click the mouse to resize the window vertically) |
| HTBOTTOMLEFT | In the lower-left corner of a border of a resizable window (the user can click the mouse to resize the window diagonally) |
| HTBOTTOMRIGHT | In the lower-right corner of a border of a resizable window (the user can click the mouse to resize the window diagonally) |
| HTCAPTION | In a title bar |
| HTCLIENT | In a client area |
| HTCLOSE | In a Close button |

(continued)

(continued)

| Value | Location of hot spot |
|---------------|---|
| HTERROR | On the screen background or on a dividing line between windows (same as HTNOWHERE, except that the DefWindowProc function produces a system beep to indicate an error) |
| HTGROWBOX | In a size box (same as HTSIZE) |
| HTHELP | In a Help button |
| HTHSCROLL | In a horizontal scroll bar |
| HTLEFT | In the left border of a resizable window (the user can click the mouse to resize the window horizontally) |
| HTMAXBUTTON | In a Maximize button |
| HTMENU | In a menu |
| HTMINBUTTON | In a Minimize button |
| HTNOWHERE | On the screen background or on a dividing line between windows |
| HTREDUCE | In a Minimize button |
| HTRIGHT | In the right border of a resizable window (the user can click the mouse to resize the window horizontally) |
| HTSIZE | In a size box (same as HTGROWBOX) |
| HTSYSTEMMENU | In a window menu or in a Close button in a child window |
| HTTOP | In the upper-horizontal border of a window |
| HTTOPLEFT | In the upper-left corner of a window border |
| HTTOPRIGHT | In the upper-right corner of a window border |
| HTTRANSPARENT | In a window currently covered by another window in the same thread (the message will be sent to underlying windows in the same thread until one of them returns a code that is not HTTRANSPARENT) |
| HTVSCROLL | In the vertical scroll bar |
| HTZOOM | In a Maximize button |

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter 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 `winuser.h`; include `windows.h`.

See Also

Mouse Input Overview, Mouse-Input Messages, `DefWindowProc`, `GET_X_LPARAM`, `GET_Y_LPARAM`, `MAKEPOINTS`, `POINTS`

WM_NCLBUTTONDBLCLK

The `WM_NCLBUTTONDBLCLK` message is posted when the user double-clicks the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCLBUTTONDBLCLK
    WPARAM wParam,     // hit-test value
    LPARAM lParam)     // cursor position
{
    ...
}
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the `WM_NCHITTEST` message. For a list of hit-test values, see `WM_NCHITTEST`.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

You can use also the `GET_X_LPARAM` and `GET_Y_LPARAM` macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

By default, the **DefWindowProc** function tests the specified point to find out the location of the cursor and performs the appropriate action. If appropriate, **DefWindowProc** sends the **WM_SYSCOMMAND** message to the window.

A window does not need to have the **CS_DBLCLKS** style to receive **WM_NCLBUTTONDOWN** messages.

The system generates a **WM_NCLBUTTONDOWN** message when the user presses, releases, and again presses the left mouse button within the system's double-click time limit. Double-clicking the left mouse button actually generates four messages: **WM_NCLBUTTONDOWN**, **WM_NCLBUTTONUP**, **WM_NCLBUTTONDOWN**, and **WM_NCLBUTTONUP** again.

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCLBUTTONDOWN**, **WM_NCLBUTTONUP**, **WM_SYSCOMMAND**

WM_NCLBUTTONDOWN

The **WM_NCLBUTTONDOWN** message is posted when the user presses the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_NCLBUTTONDOWN
    WPARAM wParam,       // hit-test value
    LPARAM lParam        // cursor position
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

The **DefWindowProc** function tests the specified point to find the location of the cursor and performs the appropriate action. If appropriate, **DefWindowProc** sends the **WM_SYSCOMMAND** message to the window.

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCLBUTTONDOWNBLCK**, **WM_NCLBUTTONUP**, **WM_SYSCOMMAND**

WM_NCLBUTTONUP

The **WM_NCLBUTTONUP** message is posted when the user releases the left mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCLBUTTONUP
    WPARAM wParam,     // hit-test value
    LPARAM lParam      // cursor position
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

The **DefWindowProc** function tests the specified point to find out the location of the cursor and performs the appropriate action. If appropriate, **DefWindowProc** sends the **WM_SYSCOMMAND** message to the window.

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);  
yPos = GET_Y_LPARAM(lParam);
```

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCLBUTTONDBLCLK**, **WM_NCLBUTTONDOWN**, **WM_SYSCOMMAND**

WM_NCMBUTTONDOWNBLCLK

The **WM_NCMBUTTONDOWNBLCLK** message is posted when the user double-clicks the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
HRESULT CALLBACK WindowProc(  
    HWND hwnd,          // handle to window  
    UINT uMsg,         // WM_NCMBUTTONDBLCLK  
    WPARAM wParam,     // hit-test value  
    LPARAM lParam      // cursor position  
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

A window does not need to have the **CS_DBLCLKS** style to receive **WM_NCMBUTTONDBLCLK** messages.

The system generates a **WM_NCMBUTTONDBLCLK** message when the user presses, releases, and again presses the middle mouse button within the system's double-click time limit. Double-clicking the middle mouse button actually generates four messages: **WM_NCMBUTTONDOWN**, **WM_NCMBUTTONUP**, **WM_NCMBUTTONDBLCLK**, and **WM_NCMBUTTONUP** again.

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);  
yPos = GET_Y_LPARAM(lParam);
```

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.



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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCMBUTTONDOWN**, **WM_NCMBUTTONUP**, **WM_SYSCOMMAND**

WM_NCMBUTTONDOWN

The **WM_NCMBUTTONDOWN** message is posted when the user presses the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCMBUTTONDOWN
    WPARAM wParam,     // hit-test value
    LPARAM lParam       // cursor position
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCMBUTTONDBLCLK**, **WM_NCMBUTTONUP**, **WM_SYSCOMMAND**

WM_NCMBUTTONUP

The **WM_NCMBUTTONUP** message is posted when the user releases the middle mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_NCMBUTTONUP  
    WPARAM wParam,      // hit-test value  
    LPARAM lParam        // cursor position  
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.


```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCMBUTTONDBLCLK**, **WM_NCMBUTTONDOWN**, **WM_SYSCOMMAND**

WM_NCMOUSEHOVER

The **WM_NCMOUSEHOVER** message is posted to a window when the cursor hovers over the nonclient area of the window for the period of time specified in a prior call to **TrackMouseEvent**.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCMOUSEHOVER
    WPARAM wParam,     // hit-test value
    LPARAM lParam      // cursor position
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

Hover tracking stops when this message is generated. The application must call **TrackMouseEvent** again if it requires further tracking of mouse hover behavior.

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **TrackMouseEvent**, **TRACKMOUSEEVENT**, **WM_NCHITTEST**, **WM_MOUSEHOVER**

WM_NCMOUSELEAVE

The **WM_NCMOUSELEAVE** message is posted to a window when the cursor leaves the nonclient area of the window specified in a prior call to **TrackMouseEvent**.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCMOUSELEAVE
    WPARAM wParam,     // not used
    LPARAM lParam       // not used
);
```

Parameters

This message has no parameters.

Return Values

If an application processes this message, it should return zero.

Remarks

All tracking requested by **TrackMouseEvent** is canceled when this message is generated. The application must call **TrackMouseEvent** when the mouse reenters its window if it requires further tracking of mouse hover behavior.

! 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

Mouse Input Overview, Mouse-Input Messages, **TrackMouseEvent**, **TRACKMOUSEEVENT**, **WM_MOUSELEAVE**, **WM_SYSCOMMAND**

WM_NCMOUSEMOVE

The **WM_NCMOUSEMOVE** message is posted to a window when the cursor is moved within the nonclient area of the window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_NCMOUSEMOVE
    WPARAM wParam,      // hit-test value
    LPARAM lParam       // cursor position
);

```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_SYSCOMMAND**

WM_NCRBUTTONDBLCLK

The **WM_NCRBUTTONDBLCLK** message is posted when the user double-clicks the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_NCRBUTTONDBLCLK
    WPARAM wParam,     // hit-test value
    LPARAM lParam      // cursor position
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

A window does not need to have the `CS_DBLCLKS` style to receive `WM_NCRBUTTONDBLCLK` messages.

The system generates a `WM_NCRBUTTONDBLCLK` message when the user presses, releases, and again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four messages: `WM_NCRBUTTONDOWN`, `WM_NCRBUTTONUP`, `WM_NCRBUTTONDBLCLK`, and `WM_NCRBUTTONUP` again.

You can use also the `GET_X_LPARAM` and `GET_Y_LPARAM` macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

If it is appropriate to do so, the system sends the `WM_SYSCOMMAND` message to the window.

! 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

Mouse Input Overview, Mouse-Input Messages, `DefWindowProc`, `GET_X_LPARAM`, `GET_Y_LPARAM`, `MAKEPOINTS`, `POINTS`, `WM_NCHITTEST`, `WM_NCRBUTTONDOWN`, `WM_NCRBUTTONUP`, `WM_SYSCOMMAND`

WM_NCRBUTTONDOWN

The `WM_NCRBUTTONDOWN` message is posted when the user presses the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its `WindowProc` function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCRBUTTONDOWN
```

```
WPARAM wParam, // hit-test value
LPARAM lParam // cursor position
);
```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCRBUTTONDBLCLK**, **WM_NCRBUTTONUP**, **WM_SYSCOMMAND**

WM_NCRBUTTONUP

The **WM_NCRBUTTONUP** message is posted when the user releases the right mouse button while the cursor is within the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is not posted.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_NCRBUTTONUP
    WPARAM wParam,     // hit-test value
    LPARAM lParam      // cursor position
);

```

Parameters

wParam

Specifies the hit-test value returned by the **DefWindowProc** function as a result of processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

lParam

Specifies a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return zero.

Remarks

You can use also the **GET_X_LPARAM** and **GET_Y_LPARAM** macros to extract the values of the x- and y- coordinates from *lParam*.

```

xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);

```

If it is appropriate to do so, the system sends the **WM_SYSCOMMAND** message to the window.

! 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

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCRBUTTONDBLCLK**, **WM_NCRBUTTONDOWN**, **WM_SYSCOMMAND**

WM_NCXBUTTONDBLCLK

The **WM_NCXBUTTONDBLCLK** message is posted when the user double-clicks the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is *not* posted.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCXBUTTONDBLCLK
    WPARAM wParam,     // hit-test value and button value
    LPARAM lParam      // cursor position
);

```

Parameters

wParam

The low-order word specifies the hit-test value returned by the **DefWindowProc** function from processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word indicates which button was double-clicked. It can be one of the following values:

| Value | Meaning |
|----------|---|
| XBUTTON1 | The first X button was double-clicked. |
| XBUTTON2 | The second X button was double-clicked. |

lParam

Pointer to a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return **TRUE**. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to crack the *wParam* parameter:

```

nHitTest = GET_NCHITTEST_WPARAM(wParam);
fwButton = GET_XBUTTON_WPARAM(wParam);

```

You can use also the following code to get the x- and y-coordinates from *lParam*:

```

xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);

```


By default, the **DefWindowProc** function tests the specified point to get the position of the cursor and performs the appropriate action. If appropriate, it sends the **WM_SYSCOMMAND** message to the window.

A window does not need to have the **CS_DBLCLKS** style to receive **WM_NCXBUTTONDBLCLK** messages. The system generates a **WM_NCXBUTTONDBLCLK** message when the user presses, releases, and again presses an X button within the system's double-click time limit. Double-clicking one of these buttons actually generates four messages: **WM_NCXBUTTONDOWN**, **WM_NCXBUTTONUP**, **WM_NCXBUTTONDBLCLK**, and **WM_NCXBUTTONUP** again.

Unlike the **WM_NCLBUTTONDBLCLK**, **WM_NCMBUTTONDBLCLK**, and **WM_NCRBUTTONDBLCLK** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCXBUTTONDOWN**, **WM_NCXBUTTONUP**, **WM_SYSCOMMAND**

WM_NCXBUTTONDOWN

The **WM_NCXBUTTONDOWN** message is posted when the user presses the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is *not* posted.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_NCXBUTTONDOWN
    WPARAM wParam,      // hit-test value and button value
    LPARAM lParam        // cursor position
);

```

Parameters

wParam

The low-order word specifies the hit-test value returned by the **DefWindowProc** function from processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word indicates which button was pressed. It can be one of the following values:

| Value | Meaning |
|--------------|----------------------------------|
| XBUTTONDOWN1 | The first X button was pressed. |
| XBUTTONDOWN2 | The second X button was pressed. |

lParam

Pointer to a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return TRUE. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to crack the *wParam* parameter:

```
nHitTest = GET_NCHITTEST_WPARAM(wParam);
fwButton = GET_XBUTTONDOWN_WPARAM(wParam);
```

You can use also the following code to get the x- and y-coordinates from *lParam*:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

By default, the **DefWindowProc** function tests the specified point to get the position of the cursor and performs the appropriate action. If appropriate, it sends the **WM_SYSCOMMAND** message to the window.

Unlike the **WM_NCLBUTTONDOWN**, **WM_NCMBUTTONDOWN**, and **WM_NCRBUTTONDOWN** messages, an application should return TRUE from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.



Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

See Also

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCXBUTTONDBLCLK**, **WM_NCXBUTTONUP**, **WM_SYSCOMMAND**

WM_NCXBUTTONUP

The **WM_NCXBUTTONUP** message is posted when the user releases the first or second X button while the cursor is in the nonclient area of a window. This message is posted to the window that contains the cursor. If a window has captured the mouse, this message is *not* posted.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_NCXBUTTONUP
    WPARAM wParam,     // hit-test value and button value
    LPARAM lParam      // cursor position
);
```

Parameters

wParam

The low-order word specifies the hit-test value returned by the **DefWindowProc** function from processing the **WM_NCHITTEST** message. For a list of hit-test values, see **WM_NCHITTEST**.

The high-order word indicates which button was released. It can be one of the following values:

| Value | Meaning |
|----------|-----------------------------------|
| XBUTTON1 | The first X button was released. |
| XBUTTON2 | The second X button was released. |

lParam

Pointer to a **POINTS** structure that contains the x- and y-coordinates of the cursor. The coordinates are relative to the upper-left corner of the screen.

Return Values

If an application processes this message, it should return TRUE. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to crack the *wParam* parameter:

```
nHittest = GET_NCHITTEST_WPARAM(wParam);
fwButton = GET_XBUTTON_WPARAM(wParam);
```

You can use also the following code to get the x- and y-coordinates from *lParam*:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

By default, the **DefWindowProc** function tests the specified point to get the position of the cursor and performs the appropriate action. If appropriate, it sends the **WM_SYSCOMMAND** message to the window.

Unlike the **WM_NCLBUTTONDOWN**, **WM_NCMBUTTONUP**, and **WM_NCRBUTTONUP** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **MAKEPOINTS**, **POINTS**, **WM_NCHITTEST**, **WM_NCXBUTTONDOWNBLCLK**, **WM_NCXBUTTONDOWN**, **WM_SYSCOMMAND**

WM_RBUTTONDOWNBLCLK

The **WM_RBUTTONDOWNBLCLK** message is posted when the user double-clicks the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
HRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_RBUTTONDOWNBLCLK
    WPARAM wParam,      // key indicators
    LPARAM lParam       // horizontal and vertical position
);
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | Windows 2000: The first X button is down. |
| MK_XBUTTON2 | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Only windows that have the CS_DBLCLKS style can receive **WM_RBUTTONDOWNBLCLK** messages, which the system generates whenever the user presses, releases, and again presses the right mouse button within the system's double-click time limit. Double-clicking the right mouse button actually generates four messages:

WM_RBUTTONDOWN, **WM_RBUTTONUP**, **WM_RBUTTONDOWNBLCLK**, and **WM_RBUTTONUP** again.

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter 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 `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, `GET_X_LPARAM`, `GET_Y_LPARAM`, `GetCapture`, `GetDoubleClickTime`, `MAKEPOINTS`, `POINTS`, `SetCapture`, `SetDoubleClickTime`, `WM_RBUTTONDOWN`, `WM_RBUTTONUP`

WM_RBUTTONDOWN

The `WM_RBUTTONDOWN` message is posted when the user presses the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_RBUTTONDOWN
    WPARAM wParam,      // key indicators
    LPARAM lParam       // horizontal and vertical position
);

```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|--------------------------|---|
| <code>MK_CONTROL</code> | The CTRL key is down. |
| <code>MK_LBUTTON</code> | The left mouse button is down. |
| <code>MK_MBUTTON</code> | The middle mouse button is down. |
| <code>MK_RBUTTON</code> | The right mouse button is down. |
| <code>MK_SHIFT</code> | The SHIFT key is down. |
| <code>MK_XBUTTON1</code> | Windows 2000: The first X button is down. |
| <code>MK_XBUTTON2</code> | Windows 2000: The second X button is down. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter 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 winuser.h; include windows.h.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_RBUTTONDOWNBLCLK**, **WM_RBUTTONDOWN**

WM_RBUTTONDOWN

The **WM_RBUTTONDOWN** message is posted when the user releases the right mouse button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_RBUTTONDOWN
    WPARAM wParam,      // key indicators
    LPARAM lParam       // horizontal and vertical position
);
```

Parameters

wParam

Indicates whether various virtual keys are down. This parameter can be one or more of the following values:

| Value | Description |
|-----------------|---|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTONDOWN1 | Windows 2000: The first X button is down. |
| MK_XBUTTONDOWN2 | Windows 2000: The second X button is down. |

IParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return zero.

Remarks

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(IParam);
yPos = GET_Y_LPARAM(IParam);
```

You can use also the **MAKEPOINTS** macro to convert the *IParam* parameter 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 winuser.h; include windows.h.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_RBUTTONDOWNBLCLK**, **WM_RBUTTONDOWN**

WM_XBUTTONDOWNBLCLK

The **WM_XBUTTONDOWNBLCLK** message is posted when the user double-clicks the first or second X button while the cursor is in the client area of a window. If the mouse is not

captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_XBUTTONDOWNBLCLK
    WPARAM wParam,      // key indicators and button value
    LPARAM lParam       // horizontal and vertical position
);

```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. It can be one or more of the following values:

| Value | Meaning |
|-------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | The first X button is down. |
| MK_XBUTTON2 | The second X button is down. |

The high-order word indicates which button was double-clicked. It can be one of the following values:

| Value | Meaning |
|----------|---|
| XBUTTON1 | The first X button was double-clicked. |
| XBUTTON2 | The second X button was double-clicked. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return TRUE. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to crack the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM (wParam);  
fwButton = GET_XBUTTON_WPARAM (wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);  
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

Only windows that have the **CS_DBLCLKS** style can receive **WM_XBUTTONDOWNBLCLK** messages, which the system generates whenever the user presses, releases, and again presses an X button within the system's double-click time limit. Double-clicking one of these buttons actually generates four messages: **WM_XBUTTONDOWN**, **WM_XBUTTONUP**, **WM_XBUTTONDOWNBLCLK**, and **WM_XBUTTONUP** again.

Unlike the **WM_LBUTTONDOWNBLCLK**, **WM_MBUTTONDOWNBLCLK**, and **WM_RBUTTONDOWNBLCLK** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **DefWindowProc**, **GET_KEYSTATE_WPARAM**, **GET_X_LPARAM**, **GET_XBUTTON_WPARAM**, **GET_Y_LPARAM**, **GetCapture**, **GetDoubleClickTime**, **SetCapture**, **SetDoubleClickTime**, **MAKEPOINTS**, **POINTS**, **WM_XBUTTONDOWN**, **WM_XBUTTONUP**

WM_XBUTTONDOWN

The **WM_XBUTTONDOWN** message is posted when the user presses the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_XBUTTONDOWN
    WPARAM wParam,      // key indicators and button value
    LPARAM lParam       // horizontal and vertical position
);

```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. It can be one or more of the following values:

| Value | Meaning |
|-------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | The first X button is down. |
| MK_XBUTTON2 | The second X button is down. |

The high-order word indicates which button was pressed. It can be one of the following values:

| Value | Meaning |
|----------|----------------------------------|
| XBUTTON1 | The first X button was pressed. |
| XBUTTON2 | The second X button was pressed. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return TRUE. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to crack the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM (wParam);  
fwButton = GET_XBUTTON_WPARAM (wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);  
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

Unlike the **WM_LBUTTONDOWN**, **WM_MBUTTONDOWN**, and **WM_RBUTTONDOWN** messages, an application should return **TRUE** from this message if it processes it. Doing so will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_KEYSTATE_WPARAM**, **GET_X_LPARAM**, **GET_XBUTTON_WPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_XBUTTONDOWNLCLK**, **WM_XBUTTONUP**

WM_XBUTTONUP

The **WM_XBUTTONUP** message is posted when the user releases the first or second X button while the cursor is in the client area of a window. If the mouse is not captured, the message is posted to the window beneath the cursor. Otherwise, the message is posted to the window that has captured the mouse.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_XBUTTONUP  
    WPARAM wParam,      // key indicators and button released  
    LPARAM lParam        // horizontal and vertical position  
);
```

Parameters

wParam

The low-order word indicates whether various virtual keys are down. It can be one or more of the following values:

| Value | Meaning |
|-----------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTONDOWN1 | The first X button is down. |
| MK_XBUTTONDOWN2 | The second X button is down. |

The high-order word indicates which button was released. It can be one of the following values:

| Value | Meaning |
|--------------|-----------------------------------|
| XBUTTONDOWN1 | The first X button was released. |
| XBUTTONDOWN2 | The second X button was released. |

lParam

The low-order word specifies the x-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

The high-order word specifies the y-coordinate of the cursor. The coordinate is relative to the upper-left corner of the client area.

Return Values

If an application processes this message, it should return TRUE. For more information about processing the return value, see the Remarks section.

Remarks

Use the following code to crack the *wParam* parameter:

```
fwKeys = GET_KEYSTATE_WPARAM (wParam);
fwButton = GET_XBUTTONDOWN_WPARAM (wParam);
```

Use the following code to obtain the horizontal and vertical position:

```
xPos = GET_X_LPARAM(lParam);
yPos = GET_Y_LPARAM(lParam);
```

You can use also the **MAKEPOINTS** macro to convert the *lParam* parameter to a **POINTS** structure.

Unlike the **WM_LBUTTONDOWN**, **WM_MBUTTONDOWN**, and **WM_RBUTTONDOWN** messages, an application should return **TRUE** from this message, if it processes it. This will allow software that simulates this message on Windows systems earlier than Windows 2000 to determine whether the window procedure processed the message or called **DefWindowProc** to process it.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Mouse Input Overview, Mouse-Input Messages, **GET_KEYSTATE_WPARAM**, **GET_X_LPARAM**, **GET_XBUTTONDOWN_WPARAM**, **GET_Y_LPARAM**, **GetCapture**, **MAKEPOINTS**, **POINTS**, **SetCapture**, **WM_XBUTTONDOWNBLCK**, **WM_XBUTTONDOWN**

Mouse-Input Macros

GET_APPCOMMAND_LPARAM

The **GET_APPCOMMAND_LPARAM** macro retrieves the application command from the specified **LPARAM** value.

```
short GET_APPCOMMAND_LPARAM(
    LPARAM lParam
);
```

Parameters

lParam

Specifies the value to be converted.

Return Values

The return value is the bits of the high-order word representing the application command. It can be one of the following values:

| Value | Meaning |
|------------------------------|-----------------------------------|
| APPCOMMAND_BASS_BOOST | Toggle the bass boost on and off. |
| APPCOMMAND_BASS_DOWN | Decrease the bass. |
| APPCOMMAND_BASS_UP | Increase the bass. |

(continued)

(continued)

| Value | Meaning |
|--------------------------------|--------------------------|
| APPCOMMAND_BROWSER_BACKWARD | Move backward. |
| APPCOMMAND_BROWSER_FAVORITES | Open favorites. |
| APPCOMMAND_BROWSER_FORWARD | Move forward. |
| APPCOMMAND_BROWSER_HOME | Move home. |
| APPCOMMAND_BROWSER_REFRESH | Refresh page. |
| APPCOMMAND_BROWSER_SEARCH | Open search. |
| APPCOMMAND_BROWSER_STOP | Stop download. |
| APPCOMMAND_LAUNCH_APP1 | Start App1. |
| APPCOMMAND_LAUNCH_APP2 | Start App2. |
| APPCOMMAND_LAUNCH_MAIL | Open mail. |
| APPCOMMAND_MEDIA_NEXTTRACK | Go to next track. |
| APPCOMMAND_MEDIA_PLAY_PAUSE | Play or pause playback. |
| APPCOMMAND_MEDIA_PREVIOUSTRACK | Go to previous track. |
| APPCOMMAND_MEDIA_SELECT | Go to Media Select mode. |
| APPCOMMAND_MEDIA_STOP | Stop playback. |
| APPCOMMAND_TREBLE_DOWN | Decrease the treble. |
| APPCOMMAND_TREBLE_UP | Increase the treble. |
| APPCOMMAND_VOLUME_DOWN | Lower the volume. |
| APPCOMMAND_VOLUME_MUTE | Mute the volume. |
| APPCOMMAND_VOLUME_UP | Raise the volume. |

 See Also

Mouse Input Overview, Mouse-Input Macros

GET_DEVICE_LPARAM

The **GET_DEVICE_LPARAM** macro retrieves the input device type from the specified **LPARAM** value.

```
WORD GET_DEVICE_LPARAM(
    LPARAM IParam
);
```

Parameters

IParam

Specifies the value to be converted.

Return Values

The return value is the bit of the high-order word representing the input device type. It can be one of the following values:

| Value | Meaning |
|-------------------|---|
| FAPPCOMMAND_KEY | User pressed a key. |
| FAPPCOMMAND_MOUSE | User clicked a mouse button. |
| FAPPCOMMAND_OEM | An unidentified hardware source generated the event. It could be a mouse or keyboard event. |

+ See Also

Mouse Input Overview, Mouse-Input Macros

GET_KEYSTATE_LPARAM

The **GET_KEYSTATE_LPARAM** macro retrieves the state of certain virtual keys from the specified **LPARAM** value.

```
int GET_KEYSTATE_LPARAM(
    LPARAM lParam
);
```

Parameters

lParam

Specifies the value to be converted.

Return Values

The return value is the low-order word representing the virtual key state. It can be one of the following values:

| Value | Meaning |
|-------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | The first X button is down. |
| MK_XBUTTON2 | The second X button is down. |

+ See Also

Mouse Input Overview, Mouse-Input Macros

GET_KEYSTATE_WPARAM

The **GET_KEYSTATE_WPARAM** macro retrieves the state of certain virtual keys from the specified **WPARAM** value.

```
int GET_KEYSTATE_WPARAM(
    WPARAM wParam
);
```

Parameters

wParam

Specifies the value to be converted.

Return Values

The return value is the low-order word representing the virtual key state. It can be one of the following values:

| Value | Meaning |
|-------------|----------------------------------|
| MK_CONTROL | The CTRL key is down. |
| MK_LBUTTON | The left mouse button is down. |
| MK_MBUTTON | The middle mouse button is down. |
| MK_RBUTTON | The right mouse button is down. |
| MK_SHIFT | The SHIFT key is down. |
| MK_XBUTTON1 | The first X button is down. |
| MK_XBUTTON2 | The second X button is down. |

GET_NCHITTEST_WPARAM

The **GET_NCHITTEST_WPARAM** macro retrieves the hit-test value from the specified **WPARAM** value.

```
short GET_NCHITTEST_WPARAM(
    WPARAM wParam
);
```

Parameters

wParam

Specifies the value to be converted.

Return Values

The return value is the low-order word representing the hit-test value. For a list of hit-test values, see **WM_NCHITTEST**.

GET_XBUTTON_WPARAM

The **GET_XBUTTON_WPARAM** macro retrieves the state of certain buttons from the specified **WPARAM** value.

```
int GET_XBUTTON_WPARAM(  
    WPARAM wParam  
);
```

Parameters

wParam

Specifies the value to be converted.

Return Values

The high-order word indicates which button was released. It can be either of the following values:

| Value | Meaning |
|----------|-----------------------------------|
| XBUTTON1 | The first X button was released. |
| XBUTTON2 | The second X button was released. |

GET_WHEEL_DELTA_WPARAM

The **GET_WHEEL_DELTA_WPARAM** macro retrieves the wheel-delta value from the specified **WPARAM** value.

```
short GET_WHEEL_DELTA_WPARAM(  
    WPARAM wParam  
);
```

Parameters

wParam

Specifies the value to be converted.

Return Values

The return value is the high-order word representing the wheel-delta value. It indicates the distance that the wheel is rotated, expressed in multiples or divisions of `WHEEL_DELTA`, which is 120. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated backward, toward the user.

Keyboard Accelerators

A *keyboard accelerator* (or, simply, accelerator) is a keystroke or combination of keystrokes that generates a `WM_COMMAND` or `WM_SYSCOMMAND` message for an application.

About Keyboard Accelerators

Accelerators are closely related to menus—both provide the user with access to an application’s command set. Typically, users rely on an application’s menus to learn the command set and then switch over to using accelerators as they become more proficient with the application. Accelerators provide faster, more direct access to commands than menus do. At a minimum, an application should provide accelerators for the more commonly used commands. Although accelerators typically generate commands that exist as menu items, they can also generate commands that have no equivalent menu items.

Accelerator Tables

An *accelerator table* consists of an array of `ACCEL` structures, each defining an individual accelerator. Each `ACCEL` structure includes the following information:

- The accelerator’s keystroke combination.
- The accelerator’s identifier.
- Various flags. This includes one that specifies whether the system is to provide visual feedback by highlighting the corresponding menu item, if any, when the accelerator is used

To process accelerator keystrokes for a specified thread, the developer must call the `TranslateAccelerator` function in the message loop associated with the thread’s message queue. The `TranslateAccelerator` function monitors keyboard input to the message queue, checking for key combinations that match an entry in the accelerator table. When `TranslateAccelerator` finds a match, it translates the keyboard input (that is, the `WM_KEYUP` and `WM_KEYDOWN` messages) into a `WM_COMMAND` or `WM_SYSCOMMAND` message, and then sends the message to the window procedure of the specified window. Figure 8-1 shows how accelerators are processed.

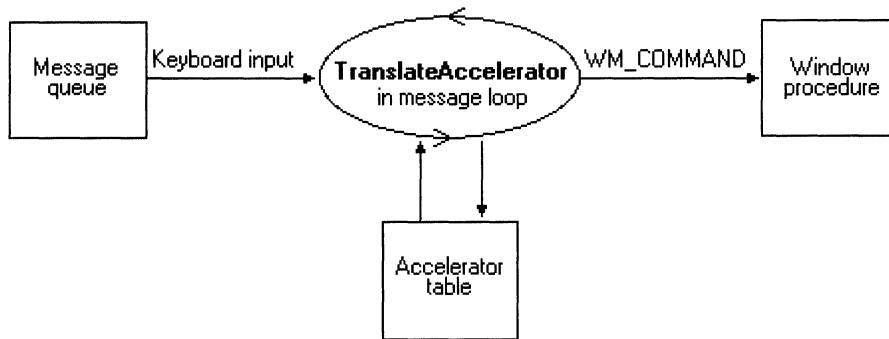


Figure 8-1: Processing keyboard accelerators.

The **WM_COMMAND** message includes the identifier of the accelerator that caused **TranslateAccelerator** to generate the message. The window procedure examines the identifier to determine the source of the message and then processes the message accordingly.

Accelerator tables exist at two different levels. The system maintains a single, system-wide accelerator table that applies to all applications. An application cannot modify the system accelerator table. For a description of the accelerators provided by the system accelerator table, see *Accelerator Keystroke Assignments*.

The system also maintains accelerator tables for each application. An application can define any number of accelerator tables for use with its own windows. A unique 32-bit handle (**HACCEL**) identifies each table. However, only one accelerator table can be active at a time for a specified thread. The handle to the accelerator table passed to the **TranslateAccelerator** function determines which accelerator table is active for a thread. The active accelerator table can be changed at any time by passing a different accelerator-table handle to **TranslateAccelerator**.

Accelerator Table Creation

Several steps are required to create an accelerator table for an application. First, a resource compiler is used to create accelerator-table resources and to add them to the application's executable file. At run time, the **LoadAccelerators** function is used to load the accelerator table into memory and retrieve the handle to the accelerator table. This handle is passed to the **TranslateAccelerator** function to activate the accelerator table.

An accelerator table can also be created for an application at run time by passing an array of **ACCEL** structures to the **CreateAcceleratorTable** function. This method supports user-defined accelerators in the application. Like the **LoadAccelerators** function, **CreateAcceleratorTable** returns an accelerator-table handle that can be passed to **TranslateAccelerator** to activate the accelerator table.

The system automatically destroys accelerator tables loaded by **LoadAccelerators**. An accelerator table created by **CreateAcceleratorTable** must be destroyed before an application closes; otherwise, the table continues to exist in memory after the application

has closed. An accelerator table is destroyed by calling the **DestroyAcceleratorTable** function.

An existing accelerator table can be copied and modified. The existing accelerator table is copied by using the **CopyAcceleratorTable** function. After the copy is modified, a handle to the new accelerator table is retrieved by calling **CreateAcceleratorTable**. Finally, the handle is passed to **TranslateAccelerator** to activate the new table.

Accelerator Keystroke Assignments

An ASCII character code or a virtual-key code can be used to define the accelerator. An ASCII character code makes the accelerator case-sensitive. The ASCII “C” character can define the accelerator as ALT+C rather than ALT+c. Case-sensitive accelerators can, however, be confusing to use. For example, the ALT+C accelerator will be generated if the CAPS LOCK key is down or if the SHIFT key is down, but not if both are down.

Typically, accelerators do not need to be case-sensitive, so most applications use virtual-key codes for accelerators rather than ASCII character codes.

Avoid accelerators that conflict with an application’s menu mnemonics, because the accelerator overrides the mnemonic, which can confuse the user. For more information about menu mnemonics, see *Menus*.

If an application defines an accelerator that is defined also in the system accelerator table, the application-defined accelerator overrides the system accelerator, but only within the context of the application. Avoid this practice, however, because it prevents the system accelerator from performing its standard role in the user interface. The system-wide accelerators are described in the following list:

| Accelerator | Description |
|------------------|---|
| ALT+ESC | Switches to the next application. |
| ALT+F4 | Closes an application or a window. |
| ALT+HYPHEN | Opens the window menu for a document window. |
| ALT+PRINT SCREEN | Copies an image in the active window onto the clipboard. |
| ALT+SPACEBAR | Opens the window menu for the application’s main window. |
| ALT+TAB | Switches to the next application. |
| CTRL+ESC | Switches to the Start menu. |
| CTRL+F4 | Closes the active group or document window. |
| F1 | Starts the application’s help file, if one exists. |
| PRINT SCREEN | Copies an image on the screen onto the clipboard. |
| SHIFT+ALT+TAB | Switches to the previous application. The user must press and hold down ALT+SHIFT while pressing TAB. |

Accelerators and Menus

Using an accelerator is the same as choosing a menu item: Both actions cause the system to send a **WM_COMMAND** or **WM_SYSCOMMAND** message to the corresponding window procedure. The **WM_COMMAND** message includes an identifier that the window procedure examines to determine the source of the message. If an accelerator generated the **WM_COMMAND** message, the identifier is that of the accelerator. Similarly, if a menu item generated the **WM_COMMAND** message, the identifier is that of the menu item. Because an accelerator provides a shortcut for choosing a command from a menu, an application usually assigns the same identifier to the accelerator and the corresponding menu item.

An application processes an accelerator **WM_COMMAND** message in exactly the same way as the corresponding menu item **WM_COMMAND** message. However, the **WM_COMMAND** message contains a flag that specifies whether the message originated from an accelerator or a menu item, in case accelerators must be processed differently from their corresponding menu items. The **WM_SYSCOMMAND** message does not contain this flag.

The identifier determines whether an accelerator generates a **WM_COMMAND** or **WM_SYSCOMMAND** message. If the identifier has the same value as a menu item in the System menu, the accelerator generates a **WM_SYSCOMMAND** message. Otherwise, the accelerator generates a **WM_COMMAND** message.

If an accelerator has the same identifier as a menu item and the menu item is grayed or disabled, the accelerator is disabled and does not generate a **WM_COMMAND** or **WM_SYSCOMMAND** message. Also, an accelerator does not generate a command message if the corresponding window is minimized.

When the user uses an accelerator that corresponds to a menu item, the window procedure receives the **WM_INITMENU** and **WM_INITMENUPOPUP** messages as though the user had selected the menu item. For information about how to process these messages, see *Menus*.

An accelerator that corresponds to a menu item should be included in the text of the menu item.

UI State

Windows 2000 enables applications to hide or show various features in the Windows user interface (UI). These settings are known as the *UI state*. The UI state includes the following settings:

- focus indicators (such as focus rectangles on buttons)
- keyboard accelerators (indicated by underlines in control labels)

A window can send messages to request a change in the UI state, can query the UI state, or enforce a certain state for its child windows. These messages are as follows:

| Message | Description |
|-------------------------|--|
| WM_CHANGEUISTATE | Indicates that the UI state should change. |
| WM_QUERYUISTATE | Retrieves the UI state for a window. |
| WM_UPDATEUISTATE | Changes the UI state. |

By default, all child windows of a top-level window are created with the same UI state as their parent.

The system handles the UI state for controls in dialog boxes. At dialog-box creation, the system initializes the UI state accordingly. All child controls inherit this state. After the dialog box is created, the system monitors the user's keystrokes. If the UI-state settings are hidden, and the user moves by using the keyboard, the system updates the UI state. For example, if the user presses the TAB key to move the focus to the next control, the system calls **WM_CHANGEUISTATE** to make the focus indicators visible. If the user presses the ALT key, the system calls **WM_CHANGEUISTATE** to make the keyboard accelerators visible.

If a control supports exploration between the UI elements it contains, it can update its own UI state. The control can call **WM_QUERYUISTATE** to retrieve and cache the initial UI state. Whenever the control receives an **WM_UPDATEUISTATE** message, it can update its UI state and send a **WM_CHANGEUISTATE** message to its parent. Each window will continue to send the message to its parent until it reaches the top-level window. The top-level window sends the **WM_UPDATEUISTATE** message to the windows in the window tree. If a window does not pass on the **WM_CHANGEUISTATE** message, it will not reach the top-level window and the UI state will not be updated.

Keyboard Accelerator Reference

Keyboard Accelerator Functions

CopyAcceleratorTable

The **CopyAcceleratorTable** function copies the specified accelerator table. This function is used to obtain the accelerator-table data that corresponds to an accelerator-table handle, or to determine the size of the accelerator-table data.

```
int CopyAcceleratorTable(
    HACCEL hAccelSrc, // handle to accelerator table
    LPACCEL lpAccelDst, // information buffer
    int cAccelEntries // number of entries in buffer
);
```

Parameters

hAccelSrc

[in] Handle to the accelerator table to copy.

lpAccelDst

[out] Pointer to an array of **ACCEL** structures that receives the accelerator-table information.

cAccelEntries

[in] Specifies the number of **ACCEL** structures to copy to the buffer pointed to by the *lpAccelDst* parameter.

Return Values

If *lpAccelDst* is NULL, the return value specifies the number of accelerator-table entries in the original table. Otherwise, it specifies the number of accelerator-table entries that were copied.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Functions, **ACCEL**, **CreateAcceleratorTable**, **DestroyAcceleratorTable**, **LoadAccelerators**, **TranslateAccelerator**

CreateAcceleratorTable

The **CreateAcceleratorTable** function creates an accelerator table.

```
HACCEL CreateAcceleratorTable(  
    LPACCEL lpaccl, // accelerator data array  
    int cEntries // number of entries in array  
);
```

Parameters

lpaccl

[in] Pointer to an array of **ACCEL** structures that describes the accelerator table.

cEntries

[in] Specifies the number of **ACCEL** structures in the array.

Return Values

If the function succeeds, the return value is the handle to the created accelerator table; otherwise, it is NULL. To get extended error information, call **GetLastError**.

Remarks

Before an application closes, it must use the **DestroyAcceleratorTable** function to destroy each accelerator table that it created by using the **CreateAcceleratorTable** 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.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Keyboard Accelerators Overview, Keyboard Accelerator Functions, **ACCEL**, **CopyAcceleratorTable**, **DestroyAcceleratorTable**, **LoadAccelerators**, **TranslateAccelerator**

DestroyAcceleratorTable

The **DestroyAcceleratorTable** function destroys an accelerator table. Before an application closes, it must use this function to destroy each accelerator table that it created by using the **CreateAcceleratorTable** function.

```
BOOL DestroyAcceleratorTable(  
    HACCEL hAccel // handle to accelerator table  
);
```

Parameters

hAccel

[in] Handle to the accelerator table to destroy. This handle must have been created by a call to the **CreateAcceleratorTable** function.

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**.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Functions, **CopyAcceleratorTable**, **CreateAcceleratorTable**, **LoadAccelerators**, **TranslateAccelerator**

LoadAccelerators

The **LoadAccelerators** function loads the specified accelerator table.

```
HACCEL LoadAccelerators(  
    HINSTANCE hInstance, // handle to module  
    LPCTSTR lpTableName // accelerator table name  
);
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the accelerator table to load.

lpTableName

[in] Pointer to a null-terminated string that contains the name of the accelerator table to load. Alternatively, this parameter can specify the resource identifier of an accelerator-table resource in the low-order word and zero in the high-order word. To create this value, use the **MAKEINTRESOURCE** macro.

Return Values

If the function succeeds, the return value is a handle to the loaded accelerator table.

If the function fails, the return value is `NULL`. To get extended error information, call **GetLastError**.

Remarks

If the accelerator table has not yet been loaded, the function loads it from the specified executable file.

Accelerator tables loaded from resources are freed automatically when the application terminates.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Functions, **CopyAcceleratorTable**, **CreateAcceleratorTable**, **DestroyAcceleratorTable**, **MAKEINTRESOURCE**

TranslateAccelerator

The **TranslateAccelerator** function processes accelerator keys for menu commands. The function translates a **WM_KEYDOWN** or **WM_SYSKEYDOWN** message to a **WM_COMMAND** or **WM_SYSCOMMAND** message (if there is an entry for the key in the specified accelerator table) and then sends the **WM_COMMAND** or **WM_SYSCOMMAND** message directly to the appropriate window procedure. **TranslateAccelerator** does not return until the window procedure has processed the message.

```
int TranslateAccelerator(
    HWND hWnd,           // handle to destination window
    HACCEL hAccTable,   // handle to accelerator table
    LPMMSG lpMsg        // message information
);
```

Parameters

hWnd

[in] Handle to the window whose messages are to be translated.

hAccTable

[in] Handle to the accelerator table. The accelerator table must have been loaded by a call to the **LoadAccelerators** function or created by a call to the **CreateAcceleratorTable** function.

lpMsg

[in] Pointer to an **MSG** structure that contains message information retrieved from the calling thread's message queue using the **GetMessage** or **PeekMessage** function.

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

To differentiate the message that this function sends from messages sent by menus or controls, the high-order word of the *wParam* parameter of the **WM_COMMAND** or **WM_SYSCOMMAND** message contains the value 1.

Accelerator key combinations used to select items from the **window** menu are translated into **WM_SYSCOMMAND** messages; all other accelerator key combinations are translated into **WM_COMMAND** messages.

When **TranslateAccelerator** returns a nonzero value and the message is translated, the application should not use the **TranslateMessage** function to process the message again.

An accelerator need not correspond to a menu command.

If the accelerator command corresponds to a menu item, the application is sent **WM_INITMENU** and **WM_INITMENUPOPUP** messages, as if the user were trying to display the menu. However, these messages are not sent if any of the following conditions exist:

- The window is disabled.
- The menu item is disabled.
- The accelerator key combination does not correspond to an item on the **window** menu and the window is minimized.
- A mouse capture is in effect. For information about mouse capture, see the **SetCapture** function.

If the specified window is the active window and no window has the keyboard focus (which is generally the case if the window is minimized), **TranslateAccelerator** translates **WM_SYSKEYUP** and **WM_SYSKEYDOWN** messages instead of **WM_KEYUP** and **WM_KEYDOWN** messages.

If an accelerator keystroke occurs that corresponds to a menu item when the window that owns the menu is minimized, **TranslateAccelerator** does not send a **WM_COMMAND** message. However, if an accelerator keystroke occurs that does not match any of the items in the window's menu or in the **window** menu, the function sends a **WM_COMMAND** message, even if the window is minimized.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Functions, **CreateAcceleratorTable**, **GetMessage**, **LoadAccelerators**, **MSG**, **PeekMessage**, **SetCapture**, **TranslateMessage**, **WM_COMMAND**, **WM_INITMENU**, **WM_INITMENUPOPUP**, **WM_KEYDOWN**, **WM_SYSKEYDOWN**, **WM_SYSCOMMAND**

Keyboard Accelerator Structures

ACCEL

The **ACCEL** structure defines an accelerator key used in an accelerator table.

```
typedef struct tagACCEL {
    BYTE    fVirt;
    WORD    key;
    WORD    cmd;
} ACCEL, *LPACCEL;
```

Members

fVirt

Specifies the accelerator behavior. This member can be one or more of the following values:

| Value | Meaning |
|-----------|---|
| FALT | The ALT key must be held down when the accelerator key is pressed. |
| FCONTROL | The CTRL key must be held down when the accelerator key is pressed. |
| FNOINVERT | Specifies that no top-level menu item is highlighted when the accelerator is used. If this flag is not specified, a top-level menu item will be highlighted, if possible, when the accelerator is used. |
| FSHIFT | The SHIFT key must be held down when the accelerator key is pressed. |
| FVIRTKEY | The key member specifies a virtual-key code. If this flag is not specified, key is assumed to specify a character code. |

key

Specifies the accelerator key. This member can be either a virtual-key code or a character code.

cmd

Specifies the accelerator identifier. This value is placed in the low-order word of the *wParam* parameter of the **WM_COMMAND** or **WM_SYSCOMMAND** message when the accelerator is pressed.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Structures, **WM_COMMAND**, **WM_SYSCOMMAND**

Keyboard Accelerator Messages

WM_CHANGEUISTATE

An application sends the **WM_CHANGEUISTATE** message to indicate that the user interface (UI) state should be changed.

To send this message, call the **SendMessage** function with the following parameters:

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    WM_CHANGEUISTATE,     // message to send
    (WPARAM) wParam,     // action
    (LPARAM) lParam;     // not used; must be NULL
);
```

Parameters*wParam*

The low-order word specifies the action to be taken. This member can be one of the following values:

| Value | Meaning |
|-----------|---|
| UIS_CLEAR | The UI-state element specified by the high-order word should be hidden. |

(continued)

(continued)

| Value | Meaning |
|----------------|---|
| UIS_INITIALIZE | The UI-state element specified by the high-order word should be changed based on the last input event. For more information, see Remarks. |
| UIS_SET | The UI-state element specified by the high-order word should be visible. |

The high-order word specifies which UI-state elements are affected. This member can be one or more of the following values:

| Flag | Meaning |
|----------------|-----------------------|
| UISF_HIDEACCEL | Keyboard accelerators |
| UISF_HIDEFOCUS | Focus indicators |

lParam

This parameter is not used and must be NULL.

Remarks

A window should send this message to itself or its parent when it must change the UI-state elements of all windows in the same hierarchy. The window procedure must let **DefWindowProc** process this message, so that the entire window tree has a consistent UI state. When the top-level window receives the **WM_CHANGEUISTATE** message, it sends a **WM_UPDATEUISTATE** message with the same parameters to all of the child windows. When the system processes the **WM_UPDATEUISTATE** message, it makes the change in the UI state.

If the low-order word of *wParam* is **UIS_INITIALIZE**, the system will send the **WM_UPDATEUISTATE** message with a UI state based on the last input event. For example, if the last input came from the mouse, the system will hide the keyboard cues. And, if the last input came from the keyboard, the system will show the keyboard cues. If the state that results from processing **WM_CHANGEUISTATE** is the same as the old state, **DefWindowProc** does not send this message.



Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.



See Also

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **DefWindowProc**, **HIWORD**, **LOWORD**, **WM_QUERYUISTATE**, **WM_UPDATEUISTATE**

WM_INITMENU

The **WM_INITMENU** message is sent when a menu is about to become active. It occurs when the user clicks an item on the menu bar or presses a menu key. This allows the application to modify the menu before it is displayed.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_INITMENU
    WPARAM wParam,      // handle to menu (HMENU)
    LPARAM lParam       // not used
);
```

Parameters

wParam

Handle to the menu to be initialized.

lParam

This parameter is not used.

Return Values

If an application processes this message, it should return zero.

Remarks

A **WM_INITMENU** message is sent only when a menu is first accessed; only one **WM_INITMENU** message is generated for each access. For example, moving the mouse across several menu items while holding down the button does not generate new messages. **WM_INITMENU** does not provide information about menu items.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **WM_INITMENUPOPUP**

WM_INITMENUPOPUP

The **WM_INITMENUPOPUP** message is sent when a drop-down menu or submenu is about to become active. This allows an application to modify the menu before it is displayed, without changing the entire menu.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_INITMENUPOPUP  
    WPARAM wParam,       // handle to menu (HMENU)  
    LPARAM lParam        // item position and indicator  
);
```

Parameters

wParam

Handle to the drop-down menu or submenu.

lParam

The low-order word specifies the zero-based relative position of the menu item that opens the drop-down menu or submenu.

The high-order word indicates whether the drop-down menu is the window menu. If the menu is the window menu, this parameter is TRUE; otherwise, it is FALSE.

Return Values

If an application processes this message, it should return 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 winuser.h; include windows.h.

+ See Also

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **HIWORD**, **LOWORD**, **WM_INITMENU**

WM_MENUCHAR

The **WM_MENUCHAR** message is sent when a menu is active and the user presses a key that does not correspond to any mnemonic or accelerator key. This message is sent to the window that owns the menu.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_MENUCHAR
    WPARAM wParam,     // character code (TCHAR), type
    LPARAM lParam      // handle to menu (HMENU)
):

```

Parameters

wParam

The low-order word specifies the character code that corresponds to the key the user pressed.

The high-order word specifies the active menu type. This parameter can be one of the following values:

| Value | Meaning |
|---------------|---|
| MF_POPUP | A drop-down menu, submenu, or shortcut menu |
| MF_SYSTEMMENU | The window menu |

lParam

Handle to the active menu.

Return Values

An application that processes this message should return one of the following values in the high-order word of the return value:

| Value | Meaning |
|-------------|---|
| MNC_IGNORE | Informs the system that it should discard the character the user pressed, and create a short beep on the system speaker. |
| MNC_CLOSE | Informs the system that it should close the active menu. |
| MNC_EXECUTE | Informs the system that it should choose the item specified in the low-order word of the return value. The owner window receives a WM_COMMAND message. |
| MNC_SELECT | Informs the system that it should select the item specified in the low-order word of the return value. |

Remarks

The low-order word is ignored if the high-order word contains 0 or 1.

An application should process this message when an accelerator is used to select a menu item that displays a 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 winuser.h; include windows.h.

+ See Also

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **HIWORD**, **LOWORD**

WM_MENUSELECT

The **WM_MENUSELECT** message is sent to a menu's owner window when the user selects a menu item.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_MENUSELECT
    WPARAM wParam,      // menu item (UINT) and flags (UINT)
    LPARAM lParam       // handle to menu (HMENU)
);
```

Parameters

wParam

The low-order word specifies the menu item or submenu index. If the selected item is a command item, this parameter contains the identifier of the menu item. If the selected item opens a drop-down menu or submenu, this parameter contains the index of the drop-down menu or submenu in the main menu, and the *lParam* parameter contains the handle to the main (clicked) menu; use the **GetSubMenu** function to get the menu handle to the drop-down menu or submenu.

The high-order word specifies one or more menu flags. This parameter can be one or more of the following values:

| Value | Description |
|----------------|----------------------------------|
| MF_BITMAP | Item displays a bitmap. |
| MF_CHECKED | Item is checked. |
| MF_DISABLED | Item is disabled. |
| MF_GRAYED | Item appears dimmed. |
| MF_HILITE | Item is highlighted. |
| MF_MOUSESELECT | Item is selected with the mouse. |

| | |
|---------------|--|
| MF_OWNERDRAW | Item is an owner-drawn item. |
| MF_POPUP | Item opens a drop-down menu or submenu. |
| MF_SYSTEMMENU | Item is contained in the window menu. The <i>lParam</i> parameter contains a handle to the menu associated with the message. |

lParam

Handle to the menu that was clicked.

Return Values

If an application processes this message, it should return zero.

Remarks

If the high-order word of *wParam* contains 0xFFFF and the *lParam* parameter contains NULL, the system has closed the menu.

Do not use the value -1 for the high-order word of *wParam*, because this value is specified as (**UINT**) **HIWORD**(*wParam*). If the value is 0xFFFF, it would be interpreted as 0x0000FFFF, not -1 , because of the cast to a **UINT**.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **GetSubMenu**, **HIWORD**, **LOWORD**

WM_QUERYUISTATE

An application sends the **WM_QUERYUISTATE** message to retrieve the user interface (UI) state for a window.

To send this message, call the **SendMessage** function with the following parameters:

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    WM_QUERYUISTATE,      // message to send
    (WPARAM) wParam,      // not used; must be zero
    (LPARAM) lParam;      // not used; must be zero
);
```

Parameters

This message has no parameters.

Return Values

The return value is NULL if the focus indicators and the keyboard accelerators are visible. Otherwise, the return value can be one or more of the following values:

| Value | Meaning |
|----------------|-----------------------------------|
| UISF_HIDEACCEL | Keyboard accelerators are hidden. |
| UISF_HIDEFOCUS | Focus indicators are hidden. |

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

See Also

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **WM_CHANGEUISTATE**, **WM_UPDATEUISTATE**

WM_SYSCHAR

The **WM_SYSCHAR** message is posted to the window with the keyboard focus when a **WM_SYSKEYDOWN** message is translated by the **TranslateMessage** function. It specifies the character code of a system character key—that is, a character key that is pressed while the ALT key is down.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_SYSCHAR
    WPARAM wParam,      // character code (TCHAR)
    LPARAM lParam       // key data
);

```

Parameters

wParam

Specifies the character code of the window menu key.

IParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Meaning |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke was auto-repeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |
| 29 | Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0. |
| 30 | Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up. |
| 31 | Specifies the transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed. |

Return Values

An application should return zero if it processes this message.

Remarks

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a standard key message instead of a system character-key message. This allows accelerator keys to be used with the active window even if the active window does not have the keyboard focus.

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; the PRINT SCRN key; the BREAK key; the NUM LOCK key; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards might support the extended-key bit in the *IParam* 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 winuser.h; include windows.h.

+ See Also

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **TranslateAccelerator**, **TranslateMessage**, **WM_SYSKEYDOWN**

WM_SYSCOMMAND

A window receives this message when the user chooses a command from the **window** menu (formerly known as the system or control menu) or when the user chooses the maximize button, minimize button, restore button, or close button.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_SYSCOMMAND
    WPARAM wParam,     // system command type
    LPARAM lParam      // horizontal and vertical position
);

```

Parameters

wParam

Specifies the type of system command requested. This parameter can be one of the following values:

| Value | Meaning |
|----------------|---|
| SC_CLOSE | Closes the window. |
| SC_CONTEXTHELP | Changes the cursor to a question mark with a pointer. If the user then clicks a control in the dialog box, the control receives a WM_HELP message. |
| SC_DEFAULT | Selects the default item; the user double-clicked the window menu. |
| SC_HOTKEY | Activates the window associated with the application-specified hot key. The <i>lParam</i> parameter identifies the window to activate. |
| SC_HSCROLL | Scrolls horizontally. |
| SC_KEYMENU | Retrieves the window menu as a result of a keystroke. For more information, see the Remarks section. |
| SC_MAXIMIZE | Maximizes the window. |

| Value | Meaning |
|-----------------|---|
| SC_MINIMIZE | Minimizes the window. |
| SC_MONITORPOWER | Sets the state of the display. This command supports devices that have power-saving features, such as a battery-powered personal computer. The <i>IParam</i> parameter can have the following values: 1 – the display is going to low power 2 – the display is being shut off |
| SC_MOUSEMENU | Retrieves the window menu as a result of a mouse click. |
| SC_MOVE | Moves the window. |
| SC_NEXTWINDOW | Moves to the next window. |
| SC_PREVWINDOW | Moves to the previous window. |
| SC_RESTORE | Restores the window to its normal position and size. |
| SC_SCREENSAVE | Executes the screen-saver application specified in the [boot] section of the System.ini file. |
| SC_SIZE | Sizes the window. |
| SC_TASKLIST | Activates the Start menu. |
| SC_VSCROLL | Scrolls vertically. |

IParam

The low-order word specifies the horizontal position of the cursor, in screen coordinates, if a window menu command is chosen with the mouse. Otherwise, this parameter is not used.

The high-order word specifies the vertical position of the cursor, in screen coordinates, if a window menu command is chosen with the mouse. This parameter is -1 if the command is chosen using a system accelerator, or zero if using a mnemonic.

Return Values

An application should return zero if it processes this message.

Remarks

To obtain the position coordinates in screen coordinates, use the following code:

```
xPos = GET_X_LPARAM(IParam); // horizontal position
yPos = GET_Y_LPARAM(IParam); // vertical position
```

The **DefWindowProc** function carries out the window menu request for the predefined actions specified in the previous table.

In **WM_SYSCOMMAND** messages, the four low-order bits of the *wParam* parameter are used internally by the system. To obtain the correct result when testing the value of

wParam, an application must combine the value 0xFFFF0 with the *wParam* value by using the bitwise AND operator.

The menu items in a window menu can be modified by using the **AppendMenu**, **GetSystemMenu**, **InsertMenu**, **InsertMenuItem**, **ModifyMenu**, and **SetMenuItem** functions. Applications that modify the window menu must process **WM_SYSCOMMAND** messages.

An application can carry out any system command at any time by passing a **WM_SYSCOMMAND** message to **DefWindowProc**. Any **WM_SYSCOMMAND** messages not handled by the application must be passed to **DefWindowProc**. Any command values added by an application must be processed by the application and cannot be passed to **DefWindowProc**.

Accelerator keys that are defined to choose items from the window menu are translated into **WM_SYSCOMMAND** messages; all other accelerator keystrokes are translated into **WM_COMMAND** messages.

If the *wParam* is **SC_KEYMENU**, *lParam* contains the character code of the key that is used with the ALT key to display the popup menu. For example, pressing ALT+F to display the **File** pop-up menu will cause a **WM_SYSCOMMAND** with *wParam* equal to **SC_KEYMENU** and *lParam* equal to 'f'.

! 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

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **AppendMenu**, **DefWindowProc**, **GET_X_LPARAM**, **GET_Y_LPARAM**, **GetSystemMenu**, **InsertMenu**, **ModifyMenu**, **WM_COMMAND**

WM_UPDATEUISTATE

An application sends the **WM_UPDATEUISTATE** message to change the user interface (UI) state for the specified window and all its child windows.

To send this message, call the **SendMessage** function with the following parameters:

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    WM_UPDATEUISTATE,     // message to send
    (LPARAM) wParam,      // action
    (LPARAM) lParam;      // not used; must be NULL
);
```

Parameters

wParam

The low-order word specifies the action to be performed. This parameter can be one of the following values:

| Value | Meaning |
|----------------|---|
| UIS_CLEAR | The UI-state element specified by the high-order word should be hidden. |
| UIS_INITIALIZE | The UI-state element specified by the high-order word should be changed based on the last input event. For more information, see Remarks. |
| UIS_SET | The UI-state element specified by the high-order word should be visible. |

The high-order word specifies which UISTATE elements are affected. This parameter can be one or more of the following values:

| Flag | Meaning |
|----------------|-----------------------|
| UISF_HIDEACCEL | Keyboard accelerators |
| UISF_HIDEFOCUS | Focus indicators |

lParam

This parameter is not used.

Remarks

A window should send this message to change the UI state of all its child windows. In contrast to the **WM_CHANGEUISTATE** message, which is a notification, when **DefWindowProc** processes the **WM_UPDATEUISTATE** message it changes the UI state and propagates the changes to all child windows.

The **DefWindowProc** function updates the UI state according to the *wParam* value. If the UI state is modified, the function sends the message to all the immediate child windows. **DefWindowProc** also sends this message when it receives a **WM_CHANGEUISTATE** message notifying the system that a child window intends to modify the UI state.

Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Keyboard Accelerators Overview, Keyboard Accelerator Messages, **DefWindowProc**, **WM_CHANGEUISTATE**, **WM_QUERYUISTATE**

Keyboard Input

About Keyboard Input

All Win32-based applications should accept user input from the keyboard as well as from the mouse. A Win32-based application receives keyboard input in the form of messages posted to its windows.

Keyboard-Input Model

The system provides device-independent keyboard support for applications by installing a keyboard device driver appropriate for the current keyboard. The system provides language-independent keyboard support by using the language-specific keyboard layout currently selected by the user or the application. The keyboard device driver receives scan codes from the keyboard, which are sent to the keyboard layout where they are translated into messages and posted to the appropriate windows in your application.

Assigned to each key on a keyboard is a unique value called a *scan code*, a device-dependent identifier for the key on the keyboard. A keyboard generates two scan codes when the user types a key—one when the user presses the key and another when the user releases the key.

The keyboard device driver interprets a scan code and translates (maps) it to a *virtual-key code*, a device-independent value defined by the system that identifies the purpose of a key. After translating a scan code, the keyboard layout creates a message that includes the scan code, the virtual-key code, and other information about the keystroke, and then places the message in the system message queue. The system removes the message from the system message queue and posts it to the message queue of the appropriate thread. Eventually, the thread's message loop removes the message and passes it to the appropriate window procedure for processing. Figure 8-2 illustrates the keyboard-input model.

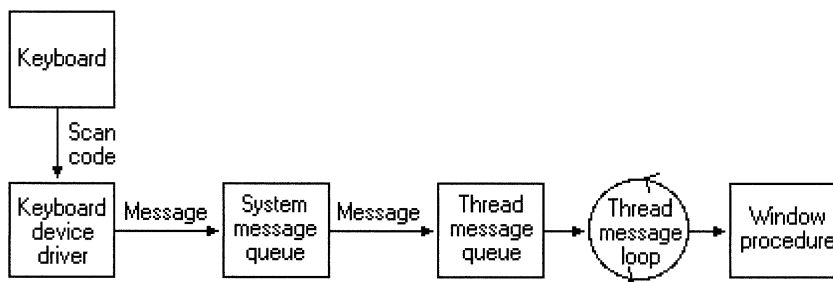


Figure 8-2: Keyboard-input model.

Keyboard-Input Reference

Keyboard-Input Functions

ActivateKeyboardLayout

The **ActivateKeyboardLayout** function sets the input locale identifier (formerly called the keyboard layout handle) for the calling thread or the current process. The input locale identifier specifies a locale as well as the physical layout of the keyboard.

```
HKL ActivateKeyboardLayout(
    HKL hkl,        // input locale identifier
    UINT Flags     // input locale identifier options
);
```

Parameters

hkl

[in] Input locale identifier to be activated.

Windows 95/98: This parameter can be obtained using **LoadKeyboardLayout** or **GetKeyboardLayoutList**, or it can be one of the values in the table that follows.

Windows NT: The input locale identifier must have been loaded by a previous call to the **LoadKeyboardLayout** function. This parameter must be either the handle to a keyboard layout or one of the following values:

| Value | Meaning |
|----------|--|
| HKL_NEXT | Selects the next locale identifier in the circular list of loaded locale identifiers maintained by the system. |
| HKL_PREV | Selects the previous locale identifier in the circular list of loaded locale identifiers maintained by the system. |

Flags

[in] Specifies how the input locale identifier is to be activated. This parameter can be one of the following values:

| Value | Meaning |
|--------------------|--|
| KLF_REORDER | <p>If this bit is set, the system's circular list of loaded locale identifiers is reordered by moving the locale identifier to the head of the list. If this bit is not set, the list is rotated without a change of order.</p> <p>For example, if a user had an English locale identifier active, as well as French, German, and Spanish locale identifiers loaded (in that order), then activating the German locale identifier with the KLF_REORDER bit set would produce the following order: German, English, French, Spanish. Activating the German locale identifier without the KLF_REORDER bit set would produce the following order: German, Spanish, English, French.</p> <p>If fewer than three locale identifiers are loaded, the value of this flag is irrelevant.</p> |
| KLF_RESET | <p>Windows 2000: If set but KLF_SHIFTLOCK is not set, the Caps Lock state is turned off by pressing the CAPS LOCK key again. If set and KLF_SHIFTLOCK is set also, the Caps Lock state is turned off by pressing either SHIFT key.</p> <p>These two methods are mutually exclusive, and the setting persists as part of the User's profile in the registry.</p> |
| KLF_SETFORPROCESS | <p>Windows 2000: Activates the specified locale identifier for the entire process and sends the WM_INPUTLANGCHANGE message to the current thread's Focus or Active window.</p> |
| KLF_SHIFTLOCK | <p>Windows 2000: This is used with KLF_RESET. See KLF_RESET for an explanation.</p> |
| KLF_UNLOADPREVIOUS | <p>This flag is unsupported. Use the UnloadKeyboardLayout function instead.</p> |

Return Values

Windows NT 3.51 and earlier: The return value is of type **BOOL**. If the function succeeds, it is nonzero. If the function fails, it is zero.

Windows 95/98, Windows NT 4.0 and later: The return value is of type **HKL**. If the function succeeds, the return value is the previous input locale identifier. Otherwise, it is zero.

To get extended error information, use the **GetLastError** function.

Remarks

This function is not restricted to keyboard layouts. The *hkl* parameter is actually an input locale identifier. This is a broader concept than a keyboard layout, since it also can encompass a speech-to-text converter, an IME, or any other form of input. Several input locale identifiers can be loaded at any one time, but only one is active at a time. Loading multiple input locale identifiers makes it possible to switch rapidly between them.

Windows 95/98: An application can create a valid input locale identifier by setting the high word to zero and the low word to a locale identifier. Using such an input locale identifier changes the input language without affecting the physical layout.

When multiple input method editors (IMEs) are allowed for each locale, passing an input locale identifier in which the high word (the device handle) is zero activates the first IME in the list belonging to the locale.

Windows 2000: The `KLF_RESET` and `KLF_SHIFTLOCK` flags alter the method by which the Caps Lock state is turned off. By default, the Caps Lock state is turned off by hitting the `CAPS LOCK` key again. If only `KLF_RESET` is set, the default state is reestablished. If `KLF_RESET` and `KLF_SHIFTLOCK` are set, the Caps Lock state is turned off by pressing either `CAPS LOCK` key. This feature is used to conform to local keyboard behavior standards as well as for personal preferences.

! 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

Keyboard Input Overview, Keyboard-Input Functions, `LoadKeyboardLayout`, `GetKeyboardLayoutName`, `UnloadKeyboardLayout`

BlockInput

Blocks keyboard and mouse-input events from reaching applications.

```
BOOL BlockInput(  
    BOOL fBlock // block option  
);
```

Parameters

fBlock

[in] Specifies the function's purpose. If this parameter is `TRUE`, keyboard and mouse input events are blocked. If this parameter is `FALSE`, keyboard and mouse events are unblocked. Note that only the thread that blocked input can successfully unblock input.

Return Values

If the function succeeds, the return value is nonzero.

If input is already blocked, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When input is blocked, real physical input from the mouse or keyboard will not affect the input queue's synchronous key state (reported by **GetKeyState** and **GetKeyboardState**), nor will it affect the asynchronous key state (reported by **GetAsyncKeyState**). However, the thread that is blocking input can affect both these key states by calling **SendInput**. Any other thread will not be able to do so.

The system will unblock input in the following cases:

- The thread that blocked input unexpectedly exits without calling **BlockInput** with *fBlock* set to FALSE. In this case, the system cleans up properly and re-enables input.
- **Windows 95/98**: The system displays the **Close Program/Fault** dialog box. This can occur if the thread faults or if the user presses CTRL+ALT+DEL.
- **Windows 2000**: The user presses CTRL+ALT+DEL or the system invokes the **Hard System Error** modal message box (for example, when a program faults or a device fails).

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winable.h.

Library: Use user32.lib.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **GetKeyState**, **SendInput**

EnableWindow

The **EnableWindow** function enables or disables mouse input and keyboard input to the specified window or control. When input is disabled, the window does not receive input such as mouse clicks and key presses. When input is enabled, the window receives all input.

```
BOOL EnableWindow(  
    HWND hWnd,        // handle to window  
    BOOL bEnable      // enable or disable input  
);
```

Parameters

hWnd

[in] Handle to the window to be enabled or disabled.

bEnable

[in] Specifies whether to enable or disable the window. If this parameter is TRUE, the window is enabled. If the parameter is FALSE, the window is disabled.

Return Values

If the window was previously disabled, the return value is nonzero.

If the window was not previously disabled, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If the window is being disabled, the system sends a **WM_CANCELMODE** message. If the enabled state of a window is changing, the system sends a **WM_ENABLE** message after the **WM_CANCELMODE** message. (These messages are sent before **EnableWindow** returns.) If a window is disabled already, its child windows are implicitly disabled, although they are not sent a **WM_ENABLE** message.

A window must be enabled before it can be activated. For example, if an application is displaying a modeless dialog box and has disabled its main window, the application must enable the main window before destroying the dialog box. Otherwise, another window will receive the keyboard focus and be activated. If a child window is disabled, it is ignored when the system tries to determine which window should receive mouse messages.

By default, a window is enabled when it is created. To create a window that is initially disabled, an application can specify the **WS_DISABLED** style in the **CreateWindow** or **CreateWindowEx** function. After a window has been created, an application can use **EnableWindow** to enable or disable the window.

An application can use this function to enable or disable a control in a dialog box. A disabled control cannot receive the keyboard focus, nor can a user gain access to 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **CreateWindow**, **CreateWindowEx**, **IsWindowEnabled**, **WM_ENABLE**

GetActiveWindow

The **GetActiveWindow** function retrieves the window handle to the active window attached to the calling thread's message queue.

```
HWND GetActiveWindow(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is the handle to the active window attached to the calling thread's message queue. Otherwise, the return value is NULL.

Remarks

To get the handle to the foreground window, you can use **GetForegroundWindow**.

Windows 98 and Windows NT 4.0 SP3 and later: To get the window handle to the active window in the message queue for another thread, use **GetGuiThreadInfo**.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **GetForegroundWindow**, **GetGuiThreadInfo**, **SetActiveWindow**

GetAsyncKeyState

The **GetAsyncKeyState** function determines whether a key is up or down at the time the function is called, and whether the key was pressed after a previous call to **GetAsyncKeyState**.

```
SHORT GetAsyncKeyState(  
    int vkey // virtual key code  
);
```

Parameters

vKey

[in] Specifies one of 256 possible virtual-key codes. For more information, see *Virtual-Key Codes*.

Windows NT/2000: You can use left-distinguishing and right-distinguishing constants to specify certain keys. See the Remarks section for further information.

Return Values

If the function succeeds, the return value specifies whether the key was pressed since the last call to **GetAsyncKeyState**, and whether the key is currently up or down. If the most significant bit is set, the key is down, and if the least significant bit is set, the key was pressed after the previous call to **GetAsyncKeyState**. The return value is zero if a window in another thread or process currently has the keyboard focus.

Windows 95: Windows 95 does not support the left- and right-distinguishing constants. If you call **GetAsyncKeyState** with these constants, the return value is zero.

Remarks

The **GetAsyncKeyState** function works with mouse buttons. However, it checks on the state of the physical mouse buttons, not on the logical mouse buttons that the physical buttons are mapped to. For example, the call **GetAsyncKeyState(VK_LBUTTONDOWN)** always returns the state of the left physical mouse button, regardless of whether it is mapped to the left or right logical mouse button. You can determine the system's current mapping of physical mouse buttons to logical mouse buttons by calling

`GetSystemMetrics(SM_SWAPBUTTON)`

which returns TRUE if the mouse buttons have been swapped.

You can use the virtual-key code constants `VK_SHIFT`, `VK_CONTROL`, and `VK_MENU` as values for the *vKey* parameter. This gives the state of the SHIFT, CTRL, or ALT keys without distinguishing between left and right.

Windows NT/2000: You can use the following virtual-key code constants as values for *vKey* to distinguish between the left and right instances of those keys:

| Code | Meaning |
|--------------------------|--------------------------|
| <code>VK_LSHIFT</code> | <code>VK_RSHIFT</code> |
| <code>VK_LCONTROL</code> | <code>VK_RCONTROL</code> |
| <code>VK_LMENU</code> | <code>VK_RMENU</code> |

These left-distinguishing and right-distinguishing constants are available only when you call the **GetKeyboardState**, **SetKeyboardState**, **GetAsyncKeyState**, **GetKeyState**, and **MapVirtualKey** 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **GetKeyboardState**, **GetKeyState**, **GetSystemMetrics**, **MapVirtualKey**, **SetKeyboardState**

GetFocus

The **GetFocus** function retrieves the handle to the window that has the keyboard focus, if the window is attached to the calling thread's message queue.

HWND **GetFocus**(VOID);

Parameters

This function has no parameters.

Return Values

The return value is the handle to the window with the keyboard focus. If the calling thread's message queue does not have an associated window with the keyboard focus, the return value is NULL.

Remarks

GetFocus returns the window with the keyboard focus for the current thread's message queue. If **GetFocus** returns NULL, another thread's queue may be attached to a window that has the keyboard focus.

Use the **GetForegroundWindow** function to retrieve the handle to the window with which the user is currently working. You can associate your thread's message queue with the windows owned by another thread by using the **AttachThreadInput** function.

Windows 98 and Windows NT 4.0 SP3 and later: To get the window with the keyboard focus on the foreground queue or the queue of another thread, use the **GetGuiThreadInfo** 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

Keyboard Input Overview, Keyboard-Input Functions, **`AttachThreadInput`**, **`GetForegroundWindow`**, **`GetGuiThreadInfo`**, **`SetFocus`**, **`WM_KILLFOCUS`**, **`WM_SETFOCUS`**

GetKeyboardLayout

The **`GetKeyboardLayout`** function retrieves the active input locale identifier (formerly called the keyboard layout) for a specified thread. If the *idThread* parameter is zero, the input locale identifier for the active thread is returned.

```
HKL GetKeyboardLayout(  
    DWORD idThread // thread identifier  
);
```

Parameters

idThread

[in] Identifies the thread to query or is zero for the current thread.

Return Values

The return value is the input locale identifier for the thread. The low word contains a language identifier for the input language and the high word contains a device handle for the physical layout of the keyboard.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can encompass also a speech-to-text converter, an IME, or any other form of input.

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

Keyboard Input Overview, Keyboard-Input Functions, **`ActivateKeyboardLayout`**, **`CreateThread`**, **`LoadKeyboardLayout`**

GetKeyboardLayoutList

The **GetKeyboardLayoutList** function retrieves the input locale identifiers (formerly called keyboard layout handles) corresponding to the current set of input locales in the system. The function copies the identifiers to the specified buffer.

```
UINT GetKeyboardLayoutList(  
    int nBuff,           // size of array  
    HKL FAR *lpList    // array of input locale identifiers  
);
```

Parameters

nBuff

[in] Specifies the maximum number of handles that the buffer can hold.

lpList

[out] Pointer to the buffer that receives the array of input locale identifiers.

Return Values

If the function succeeds, the return value is the number of input locale identifiers copied to the buffer or, if *nBuff* is zero, the return value is the size, in array elements, of the buffer needed to receive all current input locale identifiers.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can encompass also a speech-to-text converter, an IME, or any other form of input.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **GetKeyboardLayout**

GetKeyboardLayoutName

The **GetKeyboardLayoutName** function retrieves the name of the active input locale identifier (formerly called the keyboard layout).

```
BOOL GetKeyboardLayoutName(  
    LPWSTR pwszKLID // input locale identifier name  
);
```

Parameters

pwszKLID

[out] Pointer to the buffer (of at least KL_NAMELENGTH characters in length) that receives the name of the input locale identifier, including the NULL terminator. This will be a copy of the string provided to the **LoadKeyboardLayout** function, unless layout substitution took place.

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 input locale identifier is a broader concept than a keyboard layout, since it can encompass also a speech-to-text converter, an IME, or any other form of input.

Windows NT/2000: **GetKeyboardLayoutName** receives the name of the active input locale identifier for the system.

Windows 95: **GetKeyboardLayoutName** receives the name of the active input locale identifier for the calling 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 winuser.h; include windows.h.

Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **ActivateKeyboardLayout**, **LoadKeyboardLayout**, **UnloadKeyboardLayout**

GetKeyboardState

The **GetKeyboardState** function copies the status of the 256 virtual keys to the specified buffer.

```
BOOL GetKeyboardState(  
    PBYTE lpKeyState // array of status data  
);
```

Parameters

lpKeyState

[in] Pointer to the 256-byte array that will receive the status data for each virtual key.

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 call this function to retrieve the current status of all the virtual keys. The status changes as a thread removes keyboard messages from its message queue. The status does not change as keyboard messages are posted to the thread's message queue, nor does it change as keyboard messages are posted to or retrieved from message queues of other threads. (Exception: Threads that are connected through **AttachThreadInput** share the same keyboard state.)

When the function returns, each member of the array pointed to by the *lpKeyState* parameter contains status data for a virtual key. If the high-order bit is 1, the key is down; otherwise, it is up. If the low-order bit is 1, the key is toggled. A key, such as the CAPS LOCK key, is toggled if it is turned on. The key is off and untoggled if the low-order bit is 0. A toggle key's indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

To retrieve status information for an individual key, use the **GetKeyState** function. To retrieve the current state for an individual key regardless of whether the corresponding keyboard message has been retrieved from the message queue, use the **GetAsyncKeyState** function.

An application can use the virtual-key code constants **VK_SHIFT**, **VK_CONTROL** and **VK_MENU** as indices into the array pointed to by *lpKeyState*. This gives the status of the SHIFT, CTRL, or ALT keys without distinguishing between left and right. An application can also use the following virtual-key code constants as indices to distinguish between the left and right instances of those keys:

```
VK_LSHIFT  
VK_RSHIFT
```

VK_LCONTROL
 VK_RCONTROL
 VK_LMENU
 VK_RMENU

These left- and right-distinguishing constants are available to an application only through the **GetKeyboardState**, **SetKeyboardState**, **GetAsyncKeyState**, **GetKeyState**, and **MapVirtualKey** 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyState**, **MapVirtualKey**, **SetKeyboardState**

GetKeyNameText

The **GetKeyNameText** function retrieves a string that represents the name of a key.

```
int GetKeyNameText(
    LONG lParam, // second parameter of keyboard message
    LPTSTR lpString, // buffer for key name
    int nSize // maximum length of key name
);
```

Parameters

lParam

[in] Specifies the second parameter of the keyboard message (such as **WM_KEYDOWN**) to be processed. The function interprets the following portions of *lParam*:

| Bits | Meaning |
|-------|---|
| 16–23 | Scan code. |
| 24 | Extended-key flag. Distinguishes some keys on an enhanced keyboard. |
| 25 | “Don’t care” bit. The application calling this function sets this bit to indicate that the function should not distinguish between left and right CTRL and SHIFT keys, for example. |

lpString

[out] Pointer to a buffer that will receive the key name.

nSize

[in] Specifies the maximum length, in characters, of the key name, including the terminating null character. (This parameter should be equal to the size of the buffer pointed to by the *lpString* parameter.)

Return Values

If the function succeeds, a null-terminated string is copied into the specified buffer, and the return value is the length of the string, in characters, not counting the terminating null character.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The format of the key-name string depends on the current keyboard layout. The keyboard driver maintains a list of names in the form of character strings for keys with names longer than a single character. The key name is translated according to the layout of the currently installed keyboard. The name of a character key is the character itself. The names of dead keys are spelled out in full.

**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**

Keyboard Input Overview, Keyboard-Input Functions

GetKeyState

The **GetKeyState** function retrieves the status of the specified virtual key. The status specifies whether the key is up, down, or toggled (on, off—alternating each time the key is pressed).

```
SHORT GetKeyState(  
    int nVirtKey // virtual-key code  
);
```

Parameters

nVirtKey

[in] Specifies a virtual key. If the desired virtual key is a letter or digit (A through Z, a through z, or 0 through 9), *nVirtKey* must be set to the ASCII value of that character. For other keys, it must be a virtual-key code.

If a non-English keyboard layout is used, virtual keys with values in the range ASCII A through Z and 0 through 9 are used to specify most of the character keys. For example, for the German keyboard layout, the virtual key of value ASCII O (0x4F) refers to the “o” key, whereas VK_OEM_1 refers to the “o with umlaut” key.

Return Values

The return value specifies the status of the specified virtual key, as follows:

- If the high-order bit is 1, the key is down; otherwise, it is up.
- If the low-order bit is 1, the key is toggled. A key, such as the CAPS LOCK key, is toggled if it is turned on. The key is off and untoggled if the low-order bit is 0. A toggle key’s indicator light (if any) on the keyboard will be on when the key is toggled, and off when the key is untoggled.

Remarks

The key status returned from this function changes as a thread reads key messages from its message queue. The status does not reflect the interrupt-level state associated with the hardware. Use the **GetAsyncKeyState** function to retrieve that information.

An application calls **GetKeyState** in response to a keyboard-input message. This function retrieves the state of the key when the input message was generated.

To retrieve state information for all the virtual keys, use the **GetKeyboardState** function.

An application can use the virtual-key code constants VK_SHIFT, VK_CONTROL, and VK_MENU as values for the *nVirtKey* parameter. This gives the status of the SHIFT, CTRL, or ALT keys without distinguishing between left and right. An application can also use the following virtual-key code constants as values for *nVirtKey* to distinguish between the left and right instances of those keys:

```
VK_LSHIFT  
VK_RSHIFT  
VK_LCONTROL  
VK_RCONTROL  
VK_LMENU  
VK_RMENU
```

These left-distinguishing and right-distinguishing constants are available to an application only through the **GetKeyboardState**, **SetKeyboardState**, **GetAsyncKeyState**, **GetKeyState**, and **MapVirtualKey** 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **MapVirtualKey**, **SetKeyboardState**

GetLastInputInfo

The **GetLastInputInfo** function gets the time of the last input event.

```
BOOL GetLastInputInfo(  
    PLASTINPUTINFO pIii // last input event  
);
```

Parameters

pIii

[out] Pointer to a **LASTINPUTINFO** structure that receives the time of the last input event.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

This is useful for input idle detection.

! 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.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **LASTINPUTINFO**

IsWindowEnabled

The **IsWindowEnabled** function determines whether the specified window is enabled for mouse and keyboard input.

```
BOOL IsWindowEnabled(
    HWND hWnd // handle to window
);
```

Parameters

hWnd

[in] Handle to the window to test.

Return Values

If the window is enabled, the return value is nonzero.

If the window is not enabled, the return value is zero.

Remarks

A child window receives input only if it is both enabled and visible.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **EnableWindow**, **IsWindowVisible**

keybd_event

The **keybd_event** function synthesizes a keystroke. The system can use such a synthesized keystroke to generate a **WM_KEYUP** or **WM_KEYDOWN** message. The keyboard driver's interrupt handler calls the **keybd_event** function.

Windows NT/2000: This function has been superseded. Use **SendInput** instead.

```
VOID keybd_event(
    BYTE bVk,           // virtual-key code
    BYTE bScan,        // hardware scan code
    DWORD dwFlags,     // function options
```

(continued)

(continued)

```
ULONG_PTR dwExtraInfo // additional keystroke data
);
```

Parameters

bVk

[in] Specifies a virtual-key code. The code must be a value in the range 1 to 254. For a complete list, see *Virtual-Key Codes*.

bScan

This parameter is not used.

dwFlags

[in] Specifies various aspects of function operation. This parameter can be one or more of the following values:

| Value | Meaning |
|-----------------------|--|
| KEYEVENTF_EXTENDEDKEY | If specified, the scan code was preceded by a prefix byte having the value 0xE0 (224). |
| KEYEVENTF_KEYUP | If specified, the key is being released. If not specified, the key is being depressed. |

dwExtraInfo

[in] Specifies an additional value associated with the keystroke.

Return Values

This function has no return value.

Remarks

An application can simulate a press of the PRINTSCRN key in order to obtain a screen snapshot and save it to the clipboard. To do this, call **keybd_event** with the *bVk* parameter set to VK_SNAPSHOT.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyState**, **MapVirtualKey**, **SetKeyboardState**

LoadKeyboardLayout

The **LoadKeyboardLayout** function loads a new input locale identifier (formerly called the keyboard layout) into the system. Several input locale identifiers can be loaded at a time, but only one per process is active at a time. Loading multiple input locale identifiers makes it possible to rapidly switch between them.

```
HKL LoadKeyboardLayout(
    LPCTSTR pwszKLID, // input locale identifier
    UINT Flags        // input locale identifier options
);
```

Parameters

pwszKLID

[in] Pointer to the buffer that specifies the name of the input locale identifier to load. This name is a string composed of the hexadecimal value of the language identifier (low word) and a device identifier (high word). For example, U.S. English has a language identifier of 0x0409, so the primary U.S. English layout is named “00000409”. Variants of U.S. English layout (such as the Dvorak layout) are named “00010409”, “00020409”, and so on.

Flags

[in] Specifies how the input locale identifier is to be loaded. This parameter can be one of the following values:

| Value | Meaning |
|-----------------|--|
| KLF_ACTIVATE | If the specified input locale identifier is not already loaded, the function loads and activates the input locale identifier for the current thread. |
| KLF_NOTELLSHELL | Prevents a ShellProc hook procedure from receiving an HShell_LANGUAGE hook code when the new input locale identifier is loaded. This value is typically used when an application loads multiple input locale identifiers one after another. Applying this value to all but the last input locale identifier delays the shell’s processing until all input locale identifiers have been added. |
| KLF_REORDER | Moves the specified input locale identifier to the head of the input locale identifier list, making that locale identifier the active locale identifier for the current thread. This value reorders the input locale identifier list even if KLF_ACTIVATE is not provided. |

(continued)

(continued)

| Value | Meaning |
|--------------------|---|
| KLF_REPLACELANG | Windows 95/98, Windows NT 4.0, and Windows 2000: If the new input locale identifier has the same language identifier as a current input locale identifier, the new input locale identifier replaces the current one as the input locale identifier for that language. If this value is not provided and the input locale identifiers have the same language identifiers, the current input locale identifier is not replaced and the function returns NULL. |
| KLF_SUBSTITUTE_OK | Substitutes the specified input locale identifier with another locale preferred by the user. The system starts with this flag set, and it is recommended that your application always use this flag. The substitution occurs only if the HKEY_CURRENT_USER\Keyboard Layout\Substitutes registry key explicitly defines a substitution locale. For example, if the key includes the value name "00000409" with value "00010409", loading the U.S. English layout ("00000409") causes the Dvorak U.S. English layout ("00010409") to be loaded instead. The system uses KLF_SUBSTITUTE_OK when booting, and it is recommended that all applications use this value when loading input locale identifiers to ensure that the user's preference is selected. |
| KLF_SETFORPROCESS | Windows 2000: This flag is valid only with KLF_ACTIVATE. Activates the specified input locale identifier for the entire process and sends the WM_INPUTLANGCHANGE message to the current thread's Focus or Active window. Typically, LoadKeyboardLayout activates an input locale identifier only for the current thread. |
| KLF_UNLOADPREVIOUS | This flag is unsupported. Use the UnloadKeyboardLayout function, instead. |

Return Values

If the function succeeds, the return value is the input locale identifier to the locale matched with the requested name. If no matching locale is available, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can encompass also a speech-to-text converter, an IME, or any other form of input.

An application can and will typically load the default input locale identifier or IME for a language and can do so by specifying only a string version of the language identifier. If an application wants to load a specific locale or IME, it should read the registry to determine the specific input locale identifier to pass to **LoadKeyboardLayout**. In this case, a request to activate the default input locale identifier for a locale will activate the first matching one. A specific IME should be activated using an explicit input locale identifier returned from **GetKeyboardLayout**, **GetKeyboardLayoutList**, or **LoadKeyboardLayout**.

Windows 95/98: If an input locale identifier is to be loaded with the same language as a previously loaded input locale identifier and the `KLF_REPLACELANG` flag is *not* set, the call fails. Only one loaded locale may be associated with a language. (It is acceptable for multiple IMEs to be loaded with associations to the same language.)

! 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

Keyboard Input Overview, Keyboard-Input Functions, **ActivateKeyboardLayout**, **GetKeyboardLayoutName**, **MAKELANGID**, **UnloadKeyboardLayout**

MapVirtualKey

The **MapVirtualKey** function translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code.

To specify a handle to the keyboard layout to use for translating the specified code, use the **MapVirtualKeyEx** function.

```
UINT MapVirtualKey(
    UINT uCode,        // virtual-key code or scan code
    UINT uMapType      // translation to perform
);
```

Parameters

uCode

[in] Specifies the virtual-key code or scan code for a key. How this value is interpreted depends on the value of the *uMapType* parameter.

uMapType

[in] Specifies the translation to perform. The value of this parameter depends on the value of the *uCode* parameter.

| Value | Meaning |
|-------|--|
| 0 | <i>uCode</i> is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left-hand and right-hand keys, the left-hand scan code is returned. If there is no translation, the function returns 0. |
| 1 | <i>uCode</i> is a scan code and is translated into a virtual-key code that does not distinguish between left-hand and right-hand keys. If there is no translation, the function returns 0. |
| 2 | <i>uCode</i> is a virtual-key code and is translated into an unshifted character value in the low-order word of the return value. Dead keys (diacritics) are indicated by setting the top bit of the return value. If there is no translation, the function returns 0. |
| 3 | Windows NT/2000: <i>uCode</i> is a scan code and is translated into a virtual-key code that distinguishes between left-hand and right-hand keys. If there is no translation, the function returns 0. |

Return Values

The return value is either a scan code, virtual-key code, or character value, depending on the value of *uCode* and *uMapType*. If there is no translation, the return value is zero.

Remarks

An application can use **MapVirtualKey** to translate scan codes to the virtual-key code constants `VK_SHIFT`, `VK_CONTROL`, and `VK_MENU`, and vice versa. These translations do not distinguish between the left and right instances of the `SHIFT`, `CTRL`, or `ALT` keys.

Windows NT/2000: An application can get the scan code corresponding to the left or right instance of one of these keys by calling **MapVirtualKey** with *uCode* set to one of the following virtual-key code constants:

```
VK_LCONTROL
VK_LMENU
VK_LSHIFT
VK_RCONTROL
VK_RMENU
VK_RSHIFT
```

These left-distinguishing and right-distinguishing constants are available to an application only through the **GetKeyboardState**, **SetKeyboardState**, **GetAsyncKeyState**, **GetKeyState**, and **MapVirtualKey** 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **GetKeyState**, **MapVirtualKeyEx**, **SetKeyboardState**

MapVirtualKeyEx

The **MapVirtualKeyEx** function translates (maps) a virtual-key code into a scan code or character value, or translates a scan code into a virtual-key code. The function translates the codes using the input language and an input locale identifier.

```
UINT MapVirtualKeyEx(
    UINT uCode, // virtual-key code or scan code
    UINT uMapType, // translation to perform
    HKL dwk1 // input locale identifier
);
```

Parameters

uCode

[in] Specifies the virtual-key code or scan code for a key. How this value is interpreted depends on the value of the *uMapType* parameter.

uMapType

[in] Specifies the translation to perform. The value of this parameter depends on the value of the *uCode* parameter.

| Value | Meaning |
|-------|--|
| 0 | <i>uCode</i> is a virtual-key code and is translated into a scan code. If it is a virtual-key code that does not distinguish between left-hand and right-hand keys, the left-hand scan code is returned. If there is no translation, the function returns 0. |
| 1 | <i>uCode</i> is a scan code and is translated into a virtual-key code that does not distinguish between left-hand and right-hand keys. If there is no translation, the function returns 0. |
| 2 | <i>uCode</i> is a virtual-key code and is translated into an unshifted character value in the low order word of the return value. Dead keys (diacritics) are indicated by setting the top bit of the return value. If there is no translation, the function returns 0. |
| 3 | Windows NT/2000: <i>uCode</i> is a scan code and is translated into a virtual-key code that distinguishes between left-hand and right-hand keys. If there is no translation, the function returns 0. |

dwhkl

[in] Input locale identifier to use for translating the specified code. This parameter can be any input locale identifier previously returned by the **LoadKeyboardLayout** function.

Return Values

The return value is either a scan code, virtual-key code, or character value, depending on the value of *uCode* and *uMapType*. If there is no translation, the return value is zero.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

An application can use **MapVirtualKeyEx** to translate scan codes to the virtual-key code constants `VK_SHIFT`, `VK_CONTROL`, and `VK_MENU`, and vice versa. These translations do not distinguish between the left and right instances of the SHIFT, CTRL, or ALT keys.

Windows NT/2000: An application can get the scan code corresponding to the left or right instance of one of these keys by calling **MapVirtualKeyEx** with *uCode* set to one of the following virtual-key code constants:

`VK_LCONTROL`
`VK_LMENU`
`VK_LSHIFT`
`VK_RCONTROL`
`VK_RMENU`
`VK_RSHIFT`

These left-distinguishing and right-distinguishing constants are available to an application only through the **GetKeyboardState**, **SetKeyboardState**, **GetAsyncKeyState**, **GetKeyState**, **MapVirtualKey**, and **MapVirtualKeyEx** functions. For list complete table of virtual key codes, see *Virtual-Key Codes*.



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

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **GetKeyState**, **LoadKeyboardLayout**, **SetKeyboardState**

OemKeyScan

The **OemKeyScan** function maps OEM ASCII codes 0 through 0x0FF into the OEM scan codes and shift states. The function provides information that allows a program to send OEM text to another program by simulating keyboard input.

```
DWORD OemKeyScan(  
    WORD wOemChar // ASCII value of OEM character  
);
```

Parameters

wOemChar

[in] Specifies the ASCII value of the OEM character.

Return Values

The low-order word of the return value contains the scan code of the OEM character, and the high-order word contains the shift state, which can be a combination of the following bits:

| Bit | Meaning |
|-----|---|
| 1 | Either SHIFT key is pressed. |
| 2 | Either CTRL key is pressed. |
| 4 | Either ALT key is pressed. |
| 8 | The Hankaku key is pressed. |
| 16 | Reserved (defined by the keyboard layout driver). |
| 32 | Reserved (defined by the keyboard layout driver). |

If the character cannot be produced by a single keystroke using the current keyboard layout, the return value is -1.

Remarks

This function does not provide translations for characters that require CTRL+ALT or dead keys. Characters not translated by this function must be copied by simulating input using the ALT+ keypad mechanism. The NUM LOCK key must be off.

This function does not provide translations for characters that cannot be typed with one keystroke using the current keyboard layout, such as characters with diacritics requiring

dead keys. Characters not translated by this function may be simulated using the ALT+ keypad mechanism. The NUM LOCK key must be on.

This function is implemented using the **VkKeyScan** 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **VkKeyScan**

RegisterHotKey

The **RegisterHotKey** function defines a system-wide hot key.

```

BOOL RegisterHotKey(
    HWND hWnd,           // handle to window
    int id,              // hot-key identifier
    UINT fsModifiers,    // key-modifier options
    UINT vk              // virtual-key code
);

```

Parameters

hWnd

[in] Handle to the window that will receive **WM_HOTKEY** messages generated by the hot key. If this parameter is NULL, **WM_HOTKEY** messages are posted to the message queue of the calling thread and must be processed in the message loop.

id

[in] Specifies the identifier of the hot key. No other hot key in the calling thread should have the same identifier. An application must specify a value in the range 0x0000 through 0xBFFF. A shared dynamic-link library (DLL) must specify a value in the range 0xC000 through 0xFFFF (the range returned by the **GlobalAddAtom** function). To avoid conflicts with hot-key identifiers defined by other shared DLLs, a DLL should use the **GlobalAddAtom** function to obtain the hot-key identifier.

fsModifiers

[in] Specifies keys that must be pressed in combination with the key specified by the *nVirtKey* parameter in order to generate the **WM_HOTKEY** message. The *fsModifiers* parameter can be a combination of the following values:

| Value | Meaning |
|-------------|---|
| MOD_ALT | Either ALT key must be held down. |
| MOD_CONTROL | Either CTRL key must be held down. |
| MOD_SHIFT | Either SHIFT key must be held down. |
| MOD_WIN | Either WINDOWS key was held down. These keys are labeled with the Microsoft Windows logo. |

vk

[in] Specifies the virtual-key code of the hot key.

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

When a key is pressed, the system looks for a match against all hot keys. Upon finding a match, the system posts the **WM_HOTKEY** message to the message queue of the thread that registered the hot key. This message is posted to the beginning of the queue so it is removed by the next iteration of the message loop.

This function cannot associate a hot key with a window created by another thread.

RegisterHotKey fails if the keystrokes specified for the hot key have already been registered by another hot key.

If the window identified by the *hWnd* parameter already registered a hot key with the same identifier as that specified by the *id* parameter, the new values for the *fsModifiers* and *vk* parameters replace the previously specified values for these parameters.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **GlobalAddAtom**, **UnregisterHotKey**, **WM_HOTKEY**

SendInput

The **SendInput** function synthesizes keystrokes, mouse motions, and button clicks.

```
UINT SendInput(  
    UINT nInputs,           // count of input events  
    LPINPUT pInputs,       // array of input events  
    int cbSize              // size of structure  
);
```

Parameters

nInputs

[in] Specifies the number of structures in the *pInputs* array.

pInputs

[in] Pointer to an array of **INPUT** structures. Each structure represents an event to be inserted into the keyboard-input or mouse-input stream.

cbSize

[in] Specifies the size, in bytes, of an **INPUT** structure. If *cbSize* is not the size of an **INPUT** structure, the function will fail.

Return Values

The function returns the number of events that it successfully inserted into the keyboard-input or mouse-input stream. If the function returns zero, the input was already blocked by another thread.

To get extended error information, call **GetLastError**.

Remarks

The **SendInput** function inserts the events in the **INPUT** structures serially into the keyboard-input or mouse-input stream. These events are not interspersed with other keyboard-input or mouse-input events inserted either by the user (with the keyboard or mouse) or by calls to **keybd_event**, **mouse_event**, or other calls to **SendInput**.

This function does not reset the keyboard's current state. Any keys that are already pressed when the function is called might interfere with the events that this function generates. To avoid this problem, check the keyboard's state with the **GetAsyncKeyState** function, and correct as necessary.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Requires version 2.0 or later.

Header: Declared in winuser.h; include windows.h.

Library: Use user32.lib.

 See Also

Keyboard Input Overview, Keyboard-Input Functions, **INPUT**, **GetAsyncKeyState**, **keybd_event**, **mouse_event**

SetActiveWindow

The **SetActiveWindow** function activates a window. The window must be attached to the calling thread's message queue.

```
HWND SetActiveWindow(  
    HWND hWnd // handle to window  
);
```

Parameters

hWnd

[in] Handle to the top-level window to be activated.

Return Values

If the function succeeds, the return value is the handle to the window that was previously active.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **SetActiveWindow** function activates a window, but not if the application is in the background. The window will be brought into the foreground (top of Z order) if its application is in the foreground when the system activates the window.

If the window identified by the *hWnd* parameter was created by the calling thread, the active window status of the calling thread is set to *hWnd*. Otherwise, the active window status of the calling thread is set to NULL.

By using the **AttachThreadInput** function, a thread can attach its input processing to another thread. This allows a thread to call **SetActiveWindow** to activate a window attached to another thread's message queue.

 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

Keyboard Input Overview, Keyboard-Input Functions, **GetActiveWindow**, **SetForegroundWindow**, **WM_ACTIVATE**

SetFocus

The **SetFocus** function sets the keyboard focus to the specified window. The window must be attached to the calling thread's message queue.

```
HWND SetFocus(  
    HWND hWnd // handle to window  
);
```

Parameters

hWnd

[in] Handle to the window that will receive the keyboard input. If this parameter is NULL, keystrokes are ignored.

Return Values

If the function succeeds, the return value is the handle to the window that previously had the keyboard focus. If the *hWnd* parameter is invalid or the window is not attached to the calling thread's message queue, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **SetFocus** function sends a **WM_KILLFOCUS** message to the window that loses the keyboard focus and a **WM_SETFOCUS** message to the window that receives the keyboard focus. It also activates either the window that receives the focus or the parent of the window that receives the focus.

If a window is active but does not have the focus, any key pressed will produce the **WM_SYSCHAR**, **WM_SYSKEYDOWN**, or **WM_SYSKEYUP** message. If the VK_MENU key is pressed also, the *lParam* parameter of the message will have bit 30 set. Otherwise, the messages produced do not have this bit set.

By using the **AttachThreadInput** function, a thread can attach its input processing to another thread. This allows a thread to call **SetFocus** to set the keyboard focus to a window attached to another thread's message queue.

 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

Keyboard Input Overview, Keyboard-Input Functions, **AttachThreadInput**, **GetFocus**, **WM_KILLFOCUS**, **WM_SETFOCUS**, **WM_SYSCHAR**, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**

SetKeyboardState

The **SetKeyboardState** function copies a 256-byte array of keyboard key states into the calling thread's keyboard input-state table. This is the same table accessed by the **GetKeyboardState** and **GetKeyState** functions. Changes made to this table do not affect keyboard input to any other thread.

```
BOOL SetKeyboardState(  
    LPBYTE lpKeyState // array of virtual-key codes  
);
```

Parameters

lpKeyState

[in] Pointer to a 256-byte array that contains keyboard key states.

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

Because the **SetKeyboardState** function alters the input state of the calling thread and not the global input state of the system, an application cannot use **SetKeyboardState** to set the NUM LOCK, CAPS LOCK, or SCROLL LOCK (or the Japanese KANA) indicator lights on the keyboard. These can be set or cleared using **SendInput** to simulate keystrokes.

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

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **GetKeyState**, **MapVirtualKey**, **SendInput**

ToAscii

The **ToAscii** function translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the keyboard layout handle.

To specify a handle to the keyboard layout to use to translate the specified code, use the **ToAsciiEx** function.

```
int ToAscii(
    UINT uVirtKey,           // virtual-key code
    UINT uScanCode,        // scan code
    CONST PBYTE lpKeyState, // key-state array
    LPWORD lpChar,         // buffer for translated key
    UINT uFlags             // active-menu option
);
```

Parameters

uVirtKey

[in] Specifies the virtual-key code to be translated.

uScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant. The toggle state of the NUM LOCK and SCROLL LOCK keys is ignored.

lpChar

[out] Pointer to the buffer that receives the translated character or characters.

uFlags

[in] Specifies whether a menu is active. This parameter must be 1 if a menu is active, or 0 otherwise.

Return Values

If the specified key is a dead key, the return value is negative. Otherwise, it is one of the following values:

| Value | Meaning |
|-------|--|
| 0 | The specified virtual key has no translation for the current state of the keyboard. |
| 1 | One character was copied to the buffer. |
| 2 | Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character. |

Remarks

The parameters supplied to the **ToAscii** function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, **ToAscii** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *uScanCode* parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+*number key* combinations.

Although NUM LOCK is a toggle key that affects keyboard behavior, **ToAscii** ignores the toggle setting (the low bit) of *lpKeyState* (VK_NUMLOCK, because the *uVirtKey* parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0–VK_NUMPAD9).

! 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

Keyboard Input Overview, Keyboard-Input Functions, **OemKeyScan**, **ToAsciiEx**, **ToUnicode**, **VkKeyScan**

ToAsciiEx

The **ToAsciiEx** function translates the specified virtual-key code and keyboard state to the corresponding character or characters. The function translates the code using the input language and physical keyboard layout identified by the input locale identifier.

```
int ToAsciiEx(
    UINT uVirtKey,           // virtual-key code
    UINT uScanCode,        // scan code
```

(continued)

(continued)

```

CONST PBYTE lpKeyState, // key-state array
LPWORD lpChar, // buffer for translated key
UINT uFlags, // active-menu option
HKL dwHkl // input locale identifier
};

```

Parameters

uVirtKey

[in] Specifies the virtual-key code to be translated.

uScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up (not pressed).

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down (pressed).

The low bit, if set, indicates that the key is toggled on. In this function, only the toggle bit of the CAPS LOCK key is relevant. The toggle state of the NUM LOCK and SCOLL LOCK keys is ignored.

lpChar

[out] Pointer to the buffer that receives the translated character or characters.

uFlags

[in] Specifies whether a menu is active. This parameter must be 1 if a menu is active, zero otherwise.

dwHkl

[in] Input locale identifier to use to translate the code. This parameter can be any input locale identifier previously returned by the **LoadKeyboardLayout** function.

Return Values

If the specified key is a dead key, the return value is negative. Otherwise, it is one of the following values:

| Value | Meaning |
|-------|--|
| 0 | The specified virtual key has no translation for the current state of the keyboard. |
| 1 | One character was copied to the buffer. |
| 2 | Two characters were copied to the buffer. This usually happens when a dead-key character (accent or diacritic) stored in the keyboard layout cannot be composed with the specified virtual key to form a single character. |

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

The parameters supplied to the **ToAsciiEx** function might not be sufficient to translate the virtual-key code, because a previous dead key is stored in the keyboard layout.

Typically, **ToAsciiEx** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *uScanCode* parameter may be used to distinguish between a key press and a key release. The scan code is used for translating ALT+*number key* combinations.

Although NUM LOCK is a toggle key that affects keyboard behavior, **ToAsciiEx** ignores the toggle setting (the low bit) of *lpKeyState* (VK_NUMLOCK, because the *uVirtKey* parameter alone is sufficient to distinguish the cursor movement keys (VK_HOME, VK_INSERT, and so on) from the numeric keys (VK_DECIMAL, VK_NUMPAD0–VK_NUMPAD9).

! 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

Keyboard Input Overview, Keyboard-Input Functions, **LoadKeyboardLayout**, **MapVirtualKeyEx**, **ToUnicodeEx**, **VkKeyScan**

ToUnicode

The **ToUnicode** function translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.

To specify a handle to the keyboard layout to use to translate the specified code, use the **ToUnicodeEx** function.

```
int ToUnicode(  
    UINT wVirtKey,           // virtual-key code  
    UINT wScanCode,         // scan code  
    CONST PBYTE lpKeyState, // key-state array  
    LPWSTR pwszBuff,        // translated key buffer  
    int cchBuff,            // size of translated key buffer  
    UINT wFlags              // function options  
);
```

Parameters

wVirtKey

[in] Specifies the virtual-key code to be translated.

wScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up.

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down.

pwszBuff

[out] Pointer to the buffer that receives the translated Unicode character or characters.

cchBuff

[in] Specifies the size, in wide characters, of the buffer pointed to by the *pwszBuff* parameter.

wFlags

[in] Specifies the behavior of the function. If bit 0 is set, a menu is active. Bits 1 through 31 are reserved.

Return Values

The function returns one of the following values:

| Value | Meaning |
|-----------|---|
| -1 | The specified virtual key is a dead-key character (accent or diacritic). This value is returned regardless of the keyboard layout, even if several characters have been typed and are stored in the keyboard state. If possible, even with Unicode keyboard layouts, the function has written a spacing version of the dead-key character to the buffer specified by <i>pwszBuffer</i> . For example, the function writes the character SPACING ACUTE (0x00B4), rather than the character NON_SPACING ACUTE (0x0301). |
| 0 | The specified virtual key has no translation for the current state of the keyboard. Nothing was written to the buffer specified by <i>pwszBuffer</i> . |
| 1 | One character was written to the buffer specified by <i>pwszBuffer</i> . |
| 2 or more | Two or more characters were written to the buffer specified by <i>pwszBuff</i> . The most common cause for this is that a dead-key character (accent or diacritic) stored in the keyboard layout could not be combined with the specified virtual key to form a single character. |

Remarks

The parameters supplied to the **ToUnicode** function might not be sufficient to translate the virtual-key code because a previous dead key is stored in the keyboard layout.

Typically, **ToUnicode** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *wScanCode* parameter can be used to distinguish between a key press and a key release.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

+ See Also

Keyboard Input Overview, Keyboard-Input Functions, **ToAscii**, **ToUnicodeEx**, **VkKeyScan**

ToUnicodeEx

The **ToUnicodeEx** function translates the specified virtual-key code and keyboard state to the corresponding Unicode character or characters.

```
int ToUnicodeEx(
    UINT wVirtKey,           // virtual-key code
    UINT wScanCode,         // scan code
    CONST PBYTE lpKeyState, // key-state array
    LPWSTR pwszBuff,        // translated key buffer
    int cchBuff,            // size of translated key buffer
    UINT wFlags,            // function options
    HKL dwHkl               // input locale identifier
);
```

Parameters

wVirtKey

[in] Specifies the virtual-key code to be translated.

wScanCode

[in] Specifies the hardware scan code of the key to be translated. The high-order bit of this value is set if the key is up.

lpKeyState

[in] Pointer to a 256-byte array that contains the current keyboard state. Each element (byte) in the array contains the state of one key. If the high-order bit of a byte is set, the key is down.

pwszBuff

[out] Pointer to the buffer that receives the translated Unicode character or characters.

cchBuff

[in] Specifies the size, in wide characters, of the buffer pointed to by the *pwszBuff* parameter.

wFlags

[in] Specifies the behavior of the function. If bit 0 is set, a menu is active. Bits 1 through 31 are reserved.

dwhkl

[in] Input locale identifier used to translate the specified code. This parameter can be any input locale identifier previously returned by the **LoadKeyboardLayout** function.

Return Values

The function returns one of the following values:

| Value | Meaning |
|-----------|---|
| -1 | The specified virtual key is a dead-key character (accent or diacritic). This value is returned regardless of the keyboard layout, even if several characters have been typed and are stored in the keyboard state. If possible, even with Unicode keyboard layouts, the function has written a spacing version of the dead-key character to the buffer specified by <i>pwszBuffer</i> . For example, the function writes the character SPACING ACUTE (0x00B4), rather than the character NON_SPACING ACUTE (0x0301). |
| 0 | The specified virtual key has no translation for the current state of the keyboard. Nothing was written to the buffer specified by <i>pwszBuffer</i> . |
| 1 | One character was written to the buffer specified by <i>pwszBuffer</i> . |
| 2 or more | Two or more characters were written to the buffer specified by <i>pwszBuff</i> . The most common cause for this is that a dead-key character (accent or diacritic) stored in the keyboard layout could not be combined with the specified virtual key to form a single character. |

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can also encompass a speech-to-text converter, an IME, or any other form of input.

The parameters supplied to the **ToUnicodeEx** function might not be sufficient to translate the virtual-key code because a previous dead key is stored in the keyboard layout.

Typically, **ToUnicodeEx** performs the translation based on the virtual-key code. In some cases, however, bit 15 of the *wScanCode* parameter can be used to distinguish between a key press and a key release.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

See Also

Keyboard Input Overview, Keyboard-Input Functions, **LoadKeyboardLayout**, **ToAsciiEx**, **VkKeyScan**

UnloadKeyboardLayout

The **UnloadKeyboardLayout** function removes an input locale identifier (formerly called a keyboard layout).

```
BOOL UnloadKeyboardLayout(  
    HKL hkl // input locale identifier  
);
```

Parameters

hkl

[in] Input locale identifier to unload.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The function can fail for the following reasons:

- An invalid input locale identifier was passed.
- The input locale identifier was preloaded.
- The input locale identifier is in use.

To get extended error information, call **GetLastError**.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can encompass also a speech-to-text converter, an IME, or any other form of input.

Windows 95: **UnloadKeyboardLayout** cannot unload the system default input locale identifier. This ensures that an appropriate character set is always available for the user to type commands for the shell or names for the file system.

Windows NT/2000: **UnloadKeyboardLayout** can unload the system default input locale identifier.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **ActivateKeyboardLayout**, **GetKeyboardLayoutName**, **LoadKeyboardLayout**

UnregisterHotKey

The **UnregisterHotKey** function frees a hot key previously registered by the calling thread.

```
BOOL UnregisterHotKey(  
    HWND hWnd, // handle to window  
    int id     // hot-key identifier  
);
```

Parameters

hWnd

[in] Handle to the window associated with the hot key to be freed. This parameter should be NULL if the hot key is not associated with a window.

id

[in] Specifies the identifier of the hot key to be freed.

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**.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **RegisterHotKey**, **WM_HOTKEY**

VkKeyScan

The **VkKeyScan** function translates a character to the corresponding virtual-key code and shift state for the current keyboard.

This function has been superseded by the **VkKeyScanEx** function. You can still use **VkKeyScan**, however, if you do not need to specify a keyboard layout.

```
SHORT VkKeyScan(  
    TCHAR ch // character to translate  
);
```

Parameters

ch

[in] Specifies the character to be translated into a virtual-key code.

Return Values

If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which can be a combination of the following flag bits:

| Bit | Meaning |
|-----|---|
| 1 | Either SHIFT key is pressed. |
| 2 | Either CTRL key is pressed. |
| 4 | Either ALT key is pressed. |
| 8 | The Hankaku key is pressed |
| 16 | Reserved (defined by the keyboard layout driver). |
| 32 | Reserved (defined by the keyboard layout driver). |

If the function finds no key that translates to the passed character code, both the low-order and high-order bytes contain -1 .

Remarks

For keyboard layouts that use the right-hand ALT key as a shift key (for example, the French keyboard layout), the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL+ALT.

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to translate characters into keystrokes from the main keyboard section only. For example, the character “7” is translated into VK_7, not VK_NUMPAD7.

VkKeyScan is used by applications that send characters by using the **WM_KEYUP** and **WM_KEYDOWN** messages.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **GetKeyNameText**, **GetKeyState**, **SetKeyboardState**, **VkKeyScanEx**, **WM_KEYDOWN**, **WM_KEYUP**

VkKeyScanEx

The **VkKeyScanEx** function translates a character to the corresponding virtual-key code and shift state. The function translates the character using the input language and physical keyboard layout identified by the input locale identifier.

```
SHORT VkKeyScanEx(
    TCHAR ch, // character to translate
    HKL dwHkl // input locale identifier
);
```

Parameters

ch

[in] Specifies the character to be translated into a virtual-key code.

dwHkl

[in] Input locale identifier used to translate the character. This parameter can be any input locale identifier previously returned by the **LoadKeyboardLayout** function.

Return Values

If the function succeeds, the low-order byte of the return value contains the virtual-key code and the high-order byte contains the shift state, which can be a combination of the following flag bits:

| Bit | Meaning |
|-----|------------------------------|
| 1 | Either SHIFT key is pressed. |
| 2 | Either CTRL key is pressed. |

| Bit | Meaning |
|-----|---|
| 4 | Either ALT key is pressed. |
| 8 | The Hankaku key is pressed |
| 16 | Reserved (defined by the keyboard layout driver). |
| 32 | Reserved (defined by the keyboard layout driver). |

If the function finds no key that translates to the passed character code, both the low-order and high-order bytes contain `-1`.

Remarks

The input locale identifier is a broader concept than a keyboard layout, since it can encompass also a speech-to-text converter, an IME, or any other form of input.

For keyboard layouts that use the right-hand ALT key as a shift key (for example, the French keyboard layout), the shift state is represented by the value 6, because the right-hand ALT key is converted internally into CTRL+ALT.

Translations for the numeric keypad (VK_NUMPAD0 through VK_DIVIDE) are ignored. This function is intended to translate characters into keystrokes from the main keyboard section only. For example, the character "7" is translated into VK_7, not VK_NUMPAD7.

VkKeyScanEx is used by applications that send characters by using the **WM_KEYUP** and **WM_KEYDOWN** messages.

! 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

Keyboard Input Overview, Keyboard-Input Functions, **GetAsyncKeyState**, **GetKeyboardState**, **GetKeyNameText**, **GetKeyState**, **LoadKeyboardLayout**, **SetKeyboardState**, **ToAsciiEx**

Keyboard-Input Structures

HARDWAREINPUT

The **HARDWAREINPUT** structure contains information about a simulated message generated by an input device other than a keyboard or mouse.

```
typedef struct tagHARDWAREINPUT {
    DWORD    uMsg;
    WORD     wParamL;
    WORD     wParamH;
} HARDWAREINPUT, *PHARDWAREINPUT;
```

Members

uMsg

Value specifying the message generated by the input hardware.

wParamL

Specifies the low-order word of the *lParam* parameter for **uMsg**.

wParamH

Specifies the high-order word of the *lParam* parameter for **uMsg**.

! Requirements

Windows NT/2000: Unsupported.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Keyboard Input Overview, Keyboard-Input Structures, **INPUT**, **SendInput**

INPUT

The **INPUT** structure is used by **SendInput** to store information for synthesizing input events such as keystrokes, mouse movement, and mouse clicks.

```
typedef struct tagINPUT {
    DWORD    type;
    union {
        MOUSEINPUT    mi;
        KEYBDINPUT    ki;
        HARDWAREINPUT hi;
    };
} INPUT, *PINPUT;
```

Members

type

Specifies the type of the input event. This member can be one of the following values:

| Value | Meaning |
|----------------|--|
| INPUT_HARDWARE | Windows 95/98: The event is from input hardware other than a keyboard or mouse. Use the hi structure of the union. |
| INPUT_KEYBOARD | The event is a keyboard event. Use the ki structure of the union. |
| INPUT_MOUSE | The event is a mouse event. Use the mi structure of the union. |

mi

A **MOUSEINPUT** structure that contains information about a simulated mouse event.

ki

A **KEYBDINPUT** structure that contains information about a simulated keyboard event.

hi

Windows 95/98: A **HARDWAREINPUT** structure that contains information about a simulated event from input hardware other than a keyboard or mouse.

Remarks

This structure contains information identical to that used in the parameter list of the **keybd_event** or **mouse_event** function.

Windows 2000: INPUT_KEYBOARD supports nonkeyboard-input methods, such as handwriting recognition or voice recognition, as if it were text input by using the KEYEVENTF_UNICODE flag. For more information, see the remarks section of **KEYBDINPUT**.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Structures, **GetMessageExtraInfo**, **SendInput**, **SystemParametersInfo**, **keybd_event**, **mouse_event**, **HARDWARE**, **KEYBDINPUT**, **MOUSEINPUT**

KEYBDINPUT

The **KEYBDINPUT** structure contains information about a simulated keyboard event.


```
typedef struct tagKEYBDINPUT {
    WORD        wVk;
    WORD        wScan;
    DWORD       dwFlags;
    DWORD       time;
    ULONG_PTR   dwExtraInfo;
} KEYBDINPUT, *PKEYBDINPUT;
```

Members

wVk

Specifies a virtual-key code. The code must be a value in the range 1 to 254. The Winuser.h header file provides macro definitions (VK_*) for each value. If the **dwFlags** member specifies KEYEVENTF_UNICODE, **wVk** must be 0.

wScan

Specifies a hardware scan code for the key. If **dwFlags** specifies KEYEVENTF_UNICODE, **wScan** specifies a Unicode character that is to be sent to the foreground application.

dwFlags

Specifies various aspects of a keystroke. This member can be certain combinations of the following values:

| Value | Meaning |
|-----------------------|--|
| KEYEVENTF_EXTENDEDKEY | If specified, the scan code was preceded by a prefix byte that has the value 0xE0 (224). |
| KEYEVENTF_KEYUP | If specified, the key is being released. If not specified, the key is being pressed. |
| KEYEVENTF_SCANCODE | If specified, wScan identifies the key and wVk is ignored. |
| KEYEVENTF_UNICODE | Windows 2000: If specified, the system synthesizes a VK_PACKET keystroke. The wVk parameter must be zero. This flag can only be combined with the KEYEVENTF_KEYUP flag. For more information, see the Remarks section. |

time

Time stamp for the event, in milliseconds. If this parameter is zero, the system will provide its own time stamp.

dwExtraInfo

Specifies an additional value associated with the keystroke. Use the **GetMessageExtraInfo** function to obtain this information.

Remarks

Windows 2000: INPUT_KEYBOARD supports nonkeyboard-input methods—such as handwriting recognition or voice recognition—as if it were text input by using the KEYEVENTF_UNICODE flag. If KEYEVENTF_UNICODE is specified, **SendInput** sends a **WM_KEYDOWN** or **WM_KEYUP** message to the foreground thread's message queue with *wParam* equal to VK_PACKET. Once **GetMessage** or **PeekMessage** obtains this message, passing the message to **TranslateMessage** posts a **WM_CHAR** message with the Unicode character originally specified by *wScan*. This Unicode character will automatically be converted to the appropriate ANSI value if it is posted to an ANSI window.

Windows 2000: Set the KEYEVENTF_SCANCODE flag to define keyboard input in terms of the scan code. This is useful to simulate a physical keystroke regardless of which keyboard is currently being used. The virtual key value of a key can change, depending on the current keyboard layout or what other keys were pressed, but the scan code will be the same always.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Structures, **GetMessageExtraInfo**, **INPUT**, **SendInput**

LASTINPUTINFO

The **LASTINPUTINFO** structure contains the time of the last input.

```
typedef struct tagLASTINPUTINFO {
    UINT cbSize;
    DWORD dwTime;
} LASTINPUTINFO, *PLASTINPUTINFO;
```

Members

cbSize

Must be set to sizeof (LASTINPUTINFO).

dwTime

Tick count when the last input event was received.

Remarks

This function is useful for input idle detection. For more information on tick counts, see ***GetTickCount***.

! Requirements

Windows NT/2000: Requires Windows 2000.

Windows 95/98: Unsupported.

Windows CE: Unsupported.

Header: Declared in winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Structures, ***GetLastInputInfo***, ***GetTickCount***

MOUSEINPUT

The **MOUSEINPUT** structure contains information about a simulated mouse event.

```
typedef struct tagMOUSEINPUT {  
    LONG    dx;  
    LONG    dy;  
    DWORD   mouseData;  
    DWORD   dwFlags;  
    DWORD   time;  
    ULONG_PTR dwExtraInfo;  
} MOUSEINPUT, *PMOUSEINPUT;
```

Members

dx

Specifies the absolute position of the mouse, or the amount of motion since the last mouse event was generated, depending on the value of the **dwFlags** member. Absolute data is specified as the x coordinate of the mouse; relative data is specified as the number of pixels moved.

dy

Specifies the absolute position of the mouse, or the amount of motion since the last mouse event was generated, depending on the value of the **dwFlags** member. Absolute data is specified as the y coordinate of the mouse; relative data is specified as the number of pixels moved.

mouseData

If **dwFlags** contains MOUSEEVENTF_WHEEL, then **mouseData** specifies the amount of wheel movement. A positive value indicates that the wheel was rotated forward, away from the user; a negative value indicates that the wheel was rotated

backward, toward the user. One wheel click is defined as WHEEL_DELTA, which is 120.

Windows 2000: If **dwFlags** contains MOUSEEVENTF_XDOWN or MOUSEEVENTF_XUP, then **mouseData** specifies which X buttons were pressed or released. This value may be any combination of the following flags:

| Value | Meaning |
|----------|--|
| XBUTTON1 | Set if the first X button is pressed or released. |
| XBUTTON2 | Set if the second X button is pressed or released. |

If **dwFlags** does not contain MOUSEEVENTF_WHEEL, MOUSEEVENTF_XDOWN, or MOUSEEVENTF_XUP, then **mouseData** should be zero.

dwFlags

A set of bit flags that specify various aspects of mouse motion and button clicks. The bits in this member can be any reasonable combination of the following values:

| Value | Meaning |
|-------------------------|--|
| MOUSEEVENTF_ABSOLUTE | Specifies that the dx and dy members contain normalized absolute coordinates. If the flag is not set, dx and dy contain relative data (the change in position since the last reported position). This flag can be set, or not set, regardless of what kind of mouse or other pointing device, if any, is connected to the system. For further information about relative mouse motion, see the following Remarks section. |
| MOUSEEVENTF_LEFTDOWN | Specifies that the left button was pressed. |
| MOUSEEVENTF_LEFTUP | Specifies that the left button was released. |
| MOUSEEVENTF_MIDDLEDOWN | Specifies that the middle button was pressed. |
| MOUSEEVENTF_MIDDLEUP | Specifies that the middle button was released. |
| MOUSEEVENTF_MOVE | Specifies that movement occurred. |
| MOUSEEVENTF_RIGHTDOWN | Specifies that the right button was pressed. |
| MOUSEEVENTF_RIGHTUP | Specifies that the right button was released. |
| MOUSEEVENTF_VIRTUALDESK | Windows 2000: Maps coordinates to the entire desktop. Must be used with MOUSEEVENTF_ABSOLUTE. |
| MOUSEEVENTF_WHEEL | Windows NT/2000: Specifies that the wheel was moved, if the mouse has a wheel. The amount of movement is specified in mouseData . |

(continued)

(continued)

MOUSEEVENTF_XDOWN

Windows 2000: Specifies that an X button was pressed.

MOUSEEVENTF_XUP

Windows 2000: Specifies that an X button was released.

The bit flags that specify mouse button status are set to indicate changes in status, not ongoing conditions. For example, if the left mouse button is pressed and held down, `MOUSEEVENTF_LEFTDOWN` is set when the left button is first pressed, but not for subsequent motions. Similarly, `MOUSEEVENTF_LEFTUP` is set only when the button is first released.

You cannot specify both the `MOUSEEVENTF_WHEEL` flag and either `MOUSEEVENTF_XDOWN` or `MOUSEEVENTF_XUP` flags simultaneously in the `dwFlags` parameter, because they both require use of the **mouseData** field.

time

Time stamp for the event, in milliseconds. If this parameter is 0, the system will provide its own time stamp.

dwExtraInfo

Specifies an additional value associated with the mouse event. An application calls **GetMessageExtraInfo** to obtain this extra information.

Remarks

If the mouse has moved, indicated by `MOUSEEVENTF_MOVE`, **dx** and **dy** specify information about that movement. The information is specified as absolute or relative integer values.

If `MOUSEEVENTF_ABSOLUTE` value is specified, **dx** and **dy** contain normalized absolute coordinates between 0 and 65,535. The event procedure maps these coordinates onto the display surface. Coordinate (0,0) maps onto the upper-left corner of the display surface; coordinate (65535,65535) maps onto the lower-right corner. In a multimonitor system, the coordinates map to the primary monitor.

Windows 2000: If `MOUSEEVENTF_VIRTUALDESK` is specified, the coordinates map to the entire virtual desktop.

If the `MOUSEEVENTF_ABSOLUTE` value is not specified, **dx** and **dy** specify movement relative to the previous mouse event (the last reported position). Positive values mean the mouse moved right (or down); negative values mean the mouse moved left (or up).

Relative mouse motion is subject to the effects of the mouse speed and the two-mouse threshold values. A user sets these three values with the **Pointer Speed** slider of the Control Panel's **Mouse Properties** sheet. You can obtain and set these values using the **SystemParametersInfo** function.

The system applies two tests to the specified relative mouse movement. If the specified distance along either the x or y axis is greater than the first mouse threshold value, and the mouse speed is not zero, the system doubles the distance. If the specified distance along either the x or y axis is greater than the second mouse threshold value, and the

mouse speed is equal to two, the system doubles the distance that resulted from applying the first threshold test. Thus, it is possible for the system to multiply specified relative mouse movement along the x or y axis by up to four times.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Unsupported.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Keyboard Input Overview, Keyboard-Input Structures, **GetMessageExtraInfo**, **INPUT**, **SendInput**, **SystemParametersInfo**

Keyboard-Input Messages

WM_ACTIVATE

The **WM_ACTIVATE** message is sent to both the window being activated and the window being deactivated. If the windows use the same input queue, the message is sent synchronously, first to the window procedure of the top-level window being deactivated, then to the window procedure of the top-level window being activated. If the windows use different input queues, the message is sent asynchronously, so the window is activated immediately.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_ACTIVATE
    WPARAM wParam,      // activation and minimization options
    LPARAM lParam       // handle to window (HWND)
);
```

Parameters

wParam

The low-order word specifies whether the window is being activated or deactivated.

This parameter can be one of the following values:

| Value | Meaning |
|----------------|---|
| WA_ACTIVE | Activated by some method other than a mouse click (for example, by a call to the SetActiveWindow function or by use of the keyboard interface to select the window). |
| WA_CLICKACTIVE | Activated by a mouse click. |
| WA_INACTIVE | Deactivated. |

The high-order word specifies the minimized state of the window being activated or deactivated. A nonzero value indicates the window is minimized.

lParam

Handle to the window being activated or deactivated, depending on the value of the *wParam* parameter. If the low-order word of *wParam* is `WA_INACTIVE`, *lParam* is the handle to the window being activated. If the low-order word of *wParam* is `WA_ACTIVE` or `WA_CLICKACTIVE`, *lParam* is the handle to the window being deactivated. This handle can be `NULL`.

Return Values

If an application processes this message, it should return zero.

Remarks

If the window is being activated and is not minimized, the **DefWindowProc** function sets the keyboard focus to the window. If the window is activated by a mouse click, it also receives a **WM_MOUSEACTIVATE** message.

! 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

Keyboard Input Overview, Keyboard-Input Messages, **DefWindowProc**, **SetActiveWindow**, **WM_MOUSEACTIVATE**, **WM_NCACTIVATE**

WM_CHAR

The **WM_CHAR** message is posted to the window with the keyboard focus when a **WM_KEYDOWN** message is translated by the **TranslateMessage** function. **WM_CHAR** contains the character code of the key that was pressed.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_CHAR
    WPARAM wParam,     // character code (TCHAR)
    LPARAM lParam       // key data
);

```

Parameters

wParam

Specifies the character code of the key.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101- or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |
| 29 | Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0. |
| 30 | Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up. |
| 31 | Specifies the transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed. |

Return Values

An application should return zero if it processes this message.

Remarks

Because there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent **WM_KEYDOWN** message that precedes the posting of the **WM_CHAR** message.

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Some other keyboards might support the extended-key bit in the *lParam* 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 winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Messages, **TranslateMessage**, **WM_KEYDOWN**

WM_DEADCHAR

The **WM_DEADCHAR** message is posted to the window with the keyboard focus when a **WM_KEYUP** message is translated by the **TranslateMessage** function.

WM_DEADCHAR specifies a character code generated by a dead key. A dead key is a key that generates a character, such as the umlaut (double-dot), that is combined with another character to form a composite character. For example, the umlaut-O character (Ö) is generated by typing the dead key for the umlaut character, and then typing the O key.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_DEADCHAR
    WPARAM wParam,      // character code (TCHAR)
    LPARAM lParam       // key data
);
```

Parameters

wParam

Specifies the character code generated by the dead key.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |

| Value | Description |
|-------|---|
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |
| 29 | Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0. |
| 30 | Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up. |
| 31 | Specifies the transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed. |

Return Values

An application should return zero if it processes this message.

Remarks

The **WM_DEADCHAR** message typically is used by applications to give the user feedback about each key pressed. For example, an application can display the accent in the current character position without moving the caret.

Because there is not necessarily a one-to-one correspondence between keys pressed and character messages generated, the information in the high-order word of the *lParam* parameter is generally not useful to applications. The information in the high-order word applies only to the most recent **WM_KEYDOWN** message that precedes the posting of the **WM_DEADCHAR** message.

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and the right CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Some other keyboards might support the extended-key bit in the *lParam* 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 winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Messages, **TranslateMessage**, **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSDEADCHAR**

WM_GETHOTKEY

An application sends a **WM_GETHOTKEY** message to determine the hot key associated with a window.

To send this message, call the **SendMessage** function with the following parameters:

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_GETHOTKEY,         // message to send  
    (LPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

The return value is the virtual-key code and modifiers for the hot key, or NULL if no hot key is associated with the window. The virtual-key code is in the low byte of the return value and the modifiers are in the high byte. The modifiers can be a combination of the following flags:

| Value | Meaning |
|-----------------|--------------|
| HOTKEYF_ALT | ALT key |
| HOTKEYF_CONTROL | CTRL key |
| HOTKEYF_EXT | Extended key |
| HOTKEYF_SHIFT | SHIFT key |

Remarks

These hot keys are unrelated to the hot keys set by the **RegisterHotKey** 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 winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Messages, **RegisterHotKey**, **WM_SETHOTKEY**

WM_HOTKEY

The **WM_HOTKEY** message is posted when the user presses a hot key registered by the **RegisterHotKey** function. The message is placed at the top of the message queue associated with the thread that registered the hot key.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_HOTKEY
    WPARAM wParam,      // hot-key identifier
    LPARAM lParam       // options and virtual-key code
);
```

Parameters

wParam

Specifies the identifier of the hot key that generated the message. If the message was generated by a system-defined hot key, this parameter will be one of the following values:

| Value | Meaning |
|-------------------|---|
| IDHOT_SNAPDESKTOP | The “snap desktop” hot key was pressed. |
| IDHOT_SNAPWINDOW | The “snap window” hot key was pressed. |

lParam

The low-order word specifies the keys that were to be pressed in combination with the key specified by the high-order word to generate the **WM_HOTKEY** message. This word can be one or more of the following values:

| Value | Meaning |
|-------------|---|
| MOD_ALT | Either ALT key was held down. |
| MOD_CONTROL | Either CTRL key was held down. |
| MOD_SHIFT | Either SHIFT key was held down. |
| MOD_WIN | Either WINDOWS key was held down. These keys are labeled with the Microsoft Windows logo. |

The high-order word specifies the virtual-key code of the hot key.

Remarks

WM_HOTKEY is unrelated to the **WM_GETHOTKEY** and **WM_SETHOTKEY** hot keys. The **WM_HOTKEY** message is sent for generic hot keys while the **WM_SETGETHOTKEY** messages relate to window activation hot keys.

! 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

Keyboard Input Overview, Keyboard-Input Messages, **RegisterHotKey**, **WM_GETHOTKEY**, **WM_SETHOTKEY**

WM_KEYDOWN

The **WM_KEYDOWN** message is posted to the window with the keyboard focus when a nonsystem key is pressed. A nonsystem key is a key that is pressed when the ALT key is *not* pressed.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_KEYDOWN
    WPARAM wParam,       // virtual-key code
    LPARAM lParam        // key data
);
```

Parameters

wParam

Specifies the virtual-key code of the nonsystem key.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |

| Value | Description |
|-------|--|
| 29 | Specifies the context code. The value is always 0 for a WM_KEYDOWN message. |
| 30 | Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up. |
| 31 | Specifies the transition state. The value is always 0 for a WM_KEYDOWN message. |

Return Values

An application should return zero if it processes this message.

Remarks

If the F10 key is pressed, the **DefWindowProc** function sets an internal flag. When **DefWindowProc** receives the **WM_KEYUP** message, the function checks whether the internal flag is set and, if so, sends a **WM_SYSCOMMAND** message to the top-level window. The *wParam* parameter of the message is set to **SC_KEYMENU**.

Because of the autorepeat feature, more than one **WM_KEYDOWN** message may be posted before a **WM_KEYUP** message is posted. The previous key state (bit 30) can be used to determine whether the **WM_KEYDOWN** message indicates the first down transition or a repeated down transition.

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards might support the extended-key bit in the *lParam* parameter.

Windows 2000: Applications must pass *wParam* to **TranslateMessage** without altering it at all.

! 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

Keyboard Input Overview, Keyboard-Input Messages, **DefWindowProc**, **TranslateMessage**, **WM_CHAR**, **WM_KEYUP**, **WM_SYSCOMMAND**

WM_KEYUP

The **WM_KEYUP** message is posted to the window with the keyboard focus when a nonsystem key is released. A nonsystem key is a key that is pressed when the ALT key is *not* pressed, or a keyboard key that is pressed when a window has the keyboard focus.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_KEYUP
    WPARAM wParam,      // virtual-key code
    LPARAM lParam       // key data
);
```

Parameters

wParam

Specifies the virtual-key code of the nonsystem key.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|--|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. The repeat count is always one for a WM_KEYUP message. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |
| 29 | Specifies the context code. The value is always 0 for a WM_KEYUP message. |
| 30 | Specifies the previous key state. The value is always 1 for a WM_KEYUP message. |
| 31 | Specifies the transition state. The value is always 1 for a WM_KEYUP message. |

Return Values

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function sends a **WM_SYSCOMMAND** message to the top-level window if the F10 key or the ALT key was released. The *wParam* parameter of the message is set to **SC_KEYMENU**.

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards might support the extended-key bit in the *lParam* parameter.

Windows 2000: Applications must pass *wParam* to **TranslateMessage** without altering it at all.

! 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

Keyboard Input Overview, Keyboard-Input Messages, **DefWindowProc**, **TranslateMessage**, **WM_KEYDOWN**, **WM_SYSCOMMAND**

WM_KILLFOCUS

The **WM_KILLFOCUS** message is sent to a window immediately before it loses the keyboard focus.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_KILLFOCUS
    WPARAM wParam,     // handle to window (HWND)
    LPARAM lParam       // not used
);

```

Parameters

wParam

Handle to the window that receives the keyboard focus. This parameter can be NULL.

lParam

This parameter is not used.

Return Values

An application should return zero if it processes this message.

Remarks

If an application is displaying a caret, the caret should be destroyed at this 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`.

+ See Also

Keyboard Input Overview, Keyboard-Input Messages, **SetFocus**, **WM_SETFOCUS**

WM_SETFOCUS

The **WM_SETFOCUS** message is sent to a window after it has gained the keyboard focus.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,           // WM_SETFOCUS  
    WPARAM wParam,      // handle to window (HWND)  
    LPARAM lParam        // not used  
);
```

Parameters

wParam

Handle to the window that has lost the keyboard focus. This parameter can be `NULL`.

lParam

This parameter is not used.

Return Values

An application should return zero if it processes this message.

Remarks

To display a caret, an application should call the appropriate caret functions when it receives the **WM_SETFOCUS** message.

! 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

Keyboard Input Overview, Keyboard-Input Messages, **SetFocus**, **WM_KILLFOCUS**

WM_SETHOTKEY

An application sends a **WM_SETHOTKEY** message to a window to associate a hot key with the window. When the user presses the hot key, the system activates the window.

To send this message, call the **SendMessage** function with the following parameters:

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_SETHOTKEY,         // message to send  
    (WPARAM) wParam,      // virtual-key code and modifiers  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

The low-order word specifies the virtual-key code to associate with the window.

The high-order word can be one or more of the following values:

| Value | Meaning |
|-----------------|--------------|
| HOTKEYF_ALT | ALT key |
| HOTKEYF_CONTROL | CTRL key |
| HOTKEYF_EXT | Extended key |
| HOTKEYF_SHIFT | SHIFT key |

Setting *wParam* to NULL removes the hot key associated with a window.

lParam

This parameter is not used.

Return Values

The return value is one of the following:

| Value | Meaning |
|-------|--|
| -1 | The function is unsuccessful—the hot key is invalid. |
| 0 | The function is unsuccessful—the window is invalid. |
| 1 | The function is successful, and no other window has the same hot key. |
| 2 | The function is successful, but another window already has the same hot key. |

Remarks

A hot key cannot be associated with a child window.

VK_ESCAPE, VK_SPACE, and VK_TAB are invalid hot keys.

When the user presses the hot key, the system generates a **WM_SYSCOMMAND** message with *wParam* equal to SC_HOTKEY and *lParam* equal to the window's handle. If this message is passed on to **DefWindowProc**, the system will bring the window's last active popup (if it exists) or the window itself (if there is no popup window) to the foreground.

A window can only have one hot key. If the window already has a hot key associated with it, the new hot key replaces the old one. If more than one window has the same hot key, the window that is activated by the hot key is random.

These hot keys are unrelated to the hot keys set by **RegisterHotKey**.

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

Keyboard Input Overview, Keyboard-Input Messages, **RegisterHotKey**, **WM_GETHOTKEY**, **WM_SYSCOMMAND**

WM_SYSDEADCHAR

The **WM_SYSDEADCHAR** message is sent to the window with the keyboard focus when a **WM_SYSKEYDOWN** message is translated by the **TranslateMessage** function. **WM_SYSDEADCHAR** specifies the character code of a system dead key—that is, a dead key that is pressed while holding down the ALT key.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_SYSDEADCHAR
    WPARAM wParam,      // character code (TCHAR)
    LPARAM lParam       // key data
);

```

Parameters

wParam

Specifies the character code generated by the system dead key—that is, a dead key that is pressed while holding down the ALT key.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |
| 29 | Specifies the context code. The value is 1 if the ALT key is held down while the key is pressed; otherwise, the value is 0. |
| 30 | Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up. |
| 31 | Specifies the transition state. The value is 1 if the key is being released, or it is 0 if the key is being pressed. |

Return Values

An application should return zero if it processes this message.

Remarks

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards might support the extended-key bit in the *lParam* 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 winuser.h; include windows.h.

+ See Also

Keyboard Input Overview, Keyboard-Input Messages, **TranslateMessage**, **WM_DEADCHAR**, **WM_SYSKEYDOWN**

WM_SYSKEYDOWN

The **WM_SYSKEYDOWN** message is posted to the window with the keyboard focus when the user presses the F10 key (which activates the menu bar) or holds down the ALT key and then presses another key. It also occurs when no window currently has the keyboard focus; in this case, the **WM_SYSKEYDOWN** message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,           // WM_SYSKEYDOWN
    WPARAM wParam,       // virtual-key code
    LPARAM lParam        // key data
);

```

Parameters

wParam

Specifies the virtual-key code of the key being pressed.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. If the keystroke is held long enough, multiple messages are sent. However, the repeat count is not cumulative. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |

| Value | Description |
|-------|--|
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |
| 29 | Specifies the context code. The value is 1 if the ALT key is down while the key is pressed; it is 0 if the WM_SYSKEYDOWN message is posted to the active window because no window has the keyboard focus. |
| 30 | Specifies the previous key state. The value is 1 if the key is down before the message is sent, or it is 0 if the key is up. |
| 31 | Specifies the transition state. The value is always 0 for a WM_SYSKEYDOWN message. |

Return Values

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function examines the specified key and generates a **WM_SYSCOMMAND** message if the key is either TAB or ENTER.

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a character-key message. This allows accelerator keys to be used with the active window even if the active window does not have the keyboard focus.

Because of automatic repeat, more than one **WM_SYSKEYDOWN** message may occur before a **WM_SYSKEYUP** message is sent. The previous key state (bit 30) can be used to determine whether the **WM_SYSKEYDOWN** message indicates the first down transition or a repeated down transition.

For enhanced 101-key and 102-key keyboards, enhanced keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards might support the extended-key bit in the *lParam* parameter.

This message is sent also whenever the user presses the F10 key without the ALT key.



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

Keyboard Input Overview, Keyboard-Input Messages, **DefWindowProc**, **TranslateAccelerator**, **WM_SYSCOMMAND**, **WM_SYSKEYUP**

WM_SYSKEYUP

The **WM_SYSKEYUP** message is posted to the window with the keyboard focus when the user releases a key that was pressed while the ALT key was held down. It also occurs when no window currently has the keyboard focus; in this case, the **WM_SYSKEYUP** message is sent to the active window. The window that receives the message can distinguish between these two contexts by checking the context code in the *lParam* parameter.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_SYSKEYUP
    WPARAM wParam,     // virtual-key code
    LPARAM lParam       // key data
);
```

Parameters

wParam

Specifies the virtual-key code of the key being released.

lParam

Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag, as shown in the following table:

| Value | Description |
|-------|---|
| 0–15 | Specifies the repeat count for the current message. The value is the number of times the keystroke is autorepeated as a result of the user holding down the key. The repeat count is always one for a WM_SYSKEYUP message. |
| 16–23 | Specifies the scan code. The value depends on the original equipment manufacturer (OEM). |
| 24 | Specifies whether the key is an extended key, such as the right-hand ALT and CTRL keys that appear on an enhanced 101-key or 102-key keyboard. The value is 1 if it is an extended key; otherwise, it is 0. |
| 25–28 | Reserved; do not use. |

| Value | Description |
|-------|---|
| 29 | Specifies the context code. The value is 1 if the ALT key is down while the key is released; it is 0 if the WM_SYSKEYDOWN message is posted to the active window because no window has the keyboard focus. |
| 30 | Specifies the previous key state. The value is always 1 for a WM_SYSKEYUP message. |
| 31 | Specifies the transition state. The value is always 1 for a WM_SYSKEYUP message. |

Return Values

An application should return zero if it processes this message.

Remarks

The **DefWindowProc** function sends a **WM_SYSCOMMAND** message to the top-level window if the F10 key or the ALT key was released. The *wParam* parameter of the message is set to **SC_KEYMENU**.

When the context code is zero, the message can be passed to the **TranslateAccelerator** function, which will handle it as though it were a normal key message instead of a character-key message. This allows accelerator keys to be used with the active window even if the active window does not have the keyboard focus.

For enhanced 101-key and 102-key keyboards, extended keys are the right ALT and CTRL keys on the main section of the keyboard; the INS, DEL, HOME, END, PAGE UP, PAGE DOWN, and arrow keys in the clusters to the left of the numeric keypad; and the divide (/) and ENTER keys in the numeric keypad. Other keyboards might support the extended-key bit in the *lParam* parameter.

For non-U.S. enhanced 102-key keyboards, the right ALT key is handled as a CTRL+ALT key combination. The following table shows the sequence of messages that result when the user presses and releases this key:

| Message | Virtual-key code |
|--------------------|------------------|
| WM_KEYDOWN | VK_CONTROL |
| WM_KEYDOWN | VK_MENU |
| WM_KEYUP | VK_CONTROL |
| WM_SYSKEYUP | VK_MENU |

CHAPTER 9

Windowing

Dialog Boxes

A *dialog box* is a temporary window an application creates to retrieve user input. An application typically uses dialog boxes to prompt the user for additional information for menu items. A dialog box usually contains one or more controls (child windows) with which the user enters text, chooses options, or directs the action.

The Win32 API also provides predefined dialog boxes that support common menu items such as **Open** and **Print**. Applications that use these menu items should use the common dialog boxes to prompt for this user input, regardless of the type of application. For more information about using common dialog boxes in your applications, see *Common Dialog Box Library*.

About Dialog Boxes

The Win32 API provides many functions, messages, and predefined controls to help create and manage dialog boxes, making it easier to develop the user interface for an application.

Dialog Box Reference

Dialog Box Functions

CreateDialog

The **CreateDialog** macro creates a modeless dialog box from a dialog box template resource. The **CreateDialog** macro uses the **CreateDialogParam** function.

```
HWND CreateDialog(  
    HINSTANCE hInstance, // handle to module  
    LPCTSTR lpTemplate, // dialog box template name  
    HWND hWndParent, // handle to owner window  
    DLGPROC lpDialogFunc // dialog box procedure  
);
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the dialog box template.

lpTemplate

[in] Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the **MAKEINTRESOURCE** macro to create this value.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

Return Values

If the function succeeds, the return value is the handle to the dialog box.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **CreateDialog** function uses the **CreateWindowEx** function to create the dialog box. **CreateDialog** then sends a **WM_INITDIALOG** message (and a **WM_SETFONT** message, if the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style) to the dialog box procedure. The function displays the dialog box if the template specifies the **WS_VISIBLE** style. Finally, **CreateDialog** returns the window handle to the dialog box.

After **CreateDialog** returns, the application displays the dialog box (if it is not already displayed) by using the **ShowWindow** function. The application destroys the dialog box by using the **DestroyWindow** function.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **CreateDialogParam**, **CreateWindowEx**, **DestroyWindow**, **DialogBox**, **DialogProc**, **ShowWindow**, **WM_INITDIALOG**, **WM_SETFONT**

CreateDialogIndirect

The **CreateDialogIndirect** macro creates a modeless dialog box from a dialog box template in memory. The **CreateDialogIndirect** macro uses the **CreateDialogIndirectParam** function.

```
HWND CreateDialogIndirect(  
    HINSTANCE hInstance, // handle to module  
    LPCDLGTEMPLATE lpTemplate, // dialog box template  
    HWND hWndParent, // handle to owner window  
    DLGPROC lpDialogFunc // dialog box procedure  
);
```

Parameters

hInstance

[in] Handle to the module that creates the dialog box.

lpTemplate

[in] Pointer to a global memory object containing a template that **CreateDialogIndirect** uses to create the dialog box. A dialog box template consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format.

In a standard template, the header is a **DLGTEMPLATE** structure followed by additional variable-length arrays. The data for each control consists of a **DLGITEMTEMPLATE** structure, followed by additional variable-length arrays.

In an extended dialog box template, the header uses the **DLGTEMPLATEEX** format and the control definitions use the **DLGITEMTEMPLATEEX** format.

After **CreateDialogIndirect** returns, you can free the template, which is only used to get the dialog box started.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

Return Values

If the function succeeds, the return value is the window handle to the dialog box.

If the function fails, the return value is `NULL`. To get extended error information, call `GetLastError`.

Remarks

The `CreateDialogIndirect` macro uses the `CreateWindowEx` function to create the dialog box. `CreateDialogIndirect` then sends a `WM_INITDIALOG` message to the dialog box procedure. If the template specifies the `DS_SETFONT` or `DS_SHELLFONT` style, the function also sends a `WM_SETFONT` message to the dialog box procedure. The function displays the dialog box, if the template specifies the `WS_VISIBLE` style. Finally, `CreateDialogIndirect` returns the window handle to the dialog box.

After `CreateDialogIndirect` returns, you can use the `ShowWindow` function to display the dialog box (if it is not already visible). To destroy the dialog box, use the `DestroyWindow` function.

In a standard dialog box template, the `DLGTEMPLATE` structure and each of the `DLGITEMTEMPLATE` structures must be aligned on `DWORD` boundaries. The creation data array that follows a `DLGITEMTEMPLATE` structure must also be aligned on a `DWORD` boundary. All of the other variable-length arrays in the template must be aligned on `WORD` boundaries.

In an extended dialog box template, the `DLGTEMPLATEEX` header and each of the `DLGITEMTEMPLATEEX` control definitions must be aligned on `DWORD` boundaries. The creation data array, if any, that follows a `DLGITEMTEMPLATEEX` structure must also be aligned on a `DWORD` boundary. All of the other variable-length arrays in the template must be aligned on `WORD` boundaries.

All character strings in the dialog box template, such as titles for the dialog box and buttons, must be Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the `MultiByteToWideChar` function to generate these Unicode strings.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the `WM_INITDIALOG` message handler, instead of placing them in the template.

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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialog**, **CreateDialogIndirectParam**, **CreateDialogParam**, **CreateWindowEx**, **DestroyWindow**, **DialogProc**, **DLGITEMTEMPLATE**, **DLGITEMTEMPLATEEX**, **DLGTEMPLATE**, **DLGTEMPLATEEX**, **MultiByteToWideChar**, **ShowWindow**, **WM_INITDIALOG**, **WM_SETFONT**

CreateDialogIndirectParam

The **CreateDialogIndirectParam** function creates a modeless dialog box from a dialog box template in memory. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the *lParam* parameter of the **WM_INITDIALOG** message. An application can use this value to initialize dialog box controls.

```

HWND CreateDialogIndirectParam(
    HINSTANCE hInstance,           // handle to module
    LPCDLGTEMPLATE lpTemplate,    // dialog box template
    HWND hWndParent,             // handle to owner window
    DLGPROC lpDialogFunc,        // dialog box procedure
    LPARAM lParamInit            // initialization value
);

```

Parameters

hInstance

[in] Handle to the module that will create the dialog box.

lpTemplate

[in] Pointer to a global memory object that contains the template

CreateDialogIndirectParam uses to create the dialog box. A dialog box template consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format.

In a standard template, the header is a **DLGTEMPLATE** structure followed by additional variable-length arrays. The data for each control consists of a **DLGITEMTEMPLATE** structure, followed by additional variable-length arrays.

In an extended dialog box template, the header uses the **DLGTEMPLATEEX** format and the control definitions use the **DLGITEMTEMPLATEEX** format.

After **CreateDialogIndirectParam** returns, you can free the template, which is used only to get the dialog box started.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

lParamInit

[in] Specifies the value to pass to the dialog box in the *lParam* parameter of the **WM_INITDIALOG** message.

Return Values

If the function succeeds, the return value is the window handle to the dialog box.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **CreateDialogIndirectParam** function uses the **CreateWindowEx** function to create the dialog box. **CreateDialogIndirectParam** then sends a **WM_INITDIALOG** message to the dialog box procedure. If the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style, the function also sends a **WM_SETFONT** message to the dialog box procedure. The function displays the dialog box, if the template specifies the **WS_VISIBLE** style. Finally, **CreateDialogIndirectParam** returns the window handle to the dialog box.

After **CreateDialogIndirectParam** returns, you can use the **ShowWindow** function to display the dialog box (if it is not already visible). To destroy the dialog box, use the **DestroyWindow** function.

In a standard dialog box template, the **DLGTEMPLATE** structure and each of the **DLGITEMTEMPLATE** structures must be aligned on **DWORD** boundaries. The creation data array that follows a **DLGITEMTEMPLATE** structure must also be aligned on a **DWORD** boundary. All of the other variable-length arrays in the template must be aligned on **WORD** boundaries.

In an extended dialog box template, the **DLGTEMPLATEEX** header and each of the **DLGITEMTEMPLATEEX** control definitions must be aligned on **DWORD** boundaries. The creation data array, if any, that follows a **DLGITEMTEMPLATEEX** structure must also be aligned on a **DWORD** boundary. All of the other variable-length arrays in the template must be aligned on **WORD** boundaries.

All character strings in the dialog box template, such as titles for the dialog box and buttons, must be Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialog**, **CreateDialogIndirect**, **CreateDialogParam**, **CreateWindowEx**, **DestroyWindow**, **DialogProc**, **DLGITEMTEMPLATE**, **DLGITEMTEMPLATEEX**, **DLGTEMPLATE**, **DLGTEMPLATEEX**, **MultiByteToWideChar**, **ShowWindow**, **WM_INITDIALOG**, **WM_SETFONT**

CreateDialogParam

The **CreateDialogParam** function creates a modeless dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the *lParam* parameter of the **WM_INITDIALOG** message. An application can use this value to initialize dialog box controls.

```

HWND CreateDialogParam(
    HINSTANCE hInstance, // handle to module
    LPCTSTR lpTemplateName, // dialog box template
    HWND hWndParent, // handle to owner window
    DLGPROC lpDialogFunc, // dialog box procedure
    LPARAM dwInitParam // initialization value
);

```

Parameters

hInstance

[in] Handle to the module whose executable file contains the dialog box template.

lpTemplateName

[in] Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and low-order word must contain the identifier. You can use the **MAKEINTRESOURCE** macro to create this value.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

dwInitParam

[in] Specifies the value to pass to the dialog box procedure in the *IParam* parameter in the **WM_INITDIALOG** message.

Return Values

If the function succeeds, the return value is the window handle to the dialog box.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **CreateDialogParam** function uses the **CreateWindowEx** function to create the dialog box. **CreateDialogParam** then sends a **WM_INITDIALOG** message (and a **WM_SETFONT** message, if the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style) to the dialog box procedure. The function displays the dialog box if the template specifies the **WS_VISIBLE** style. Finally, **CreateDialogParam** returns the window handle of the dialog box.

After **CreateDialogParam** returns, the application displays the dialog box (if it is not already displayed) by using the **ShowWindow** function. The application destroys the dialog box by using the **DestroyWindow** function.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialog**, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **CreateWindowEx**, **DestroyWindow**, **DialogProc**, **MAKEINTRESOURCE**, **ShowWindow**, **WM_INITDIALOG**, **WM_SETFONT**

DefDlgProc

The **DefDlgProc** function carries out default message processing for a window procedure belonging to an application-defined dialog box class.

```
LRESULT DefDlgProc(
    HWND hDlg,        // handle to dialog box
    UINT Msg,         // message
    WPARAM wParam,    // first message parameter
    LPARAM lParam     // second message parameter
);
```

Parameters

hDlg

[in] Handle to the dialog box.

Msg

[in] Specifies the message.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message sent.

Remarks

The **DefDlgProc** function is the window procedure for the predefined class of dialog box. This procedure provides internal processing for the dialog box by forwarding messages to the dialog box procedure and carrying out default processing for any messages that the dialog box procedure returns as **FALSE**. Applications that create custom window procedures for their custom dialog boxes often use **DefDlgProc** instead of the **DefWindowProc** function to carry out default message processing.

Applications create custom dialog box classes by filling a **WNDCLASS** structure with appropriate information and registering the class with the **RegisterClass** function. Some applications fill the structure by using the **GetClassInfo** function, specifying the name of the predefined dialog box. In such cases, the applications modify at least the **IpszClassName** member before registering. In all cases, the **cbWndExtra** member of **WNDCLASS** for a custom dialog box class must be set to at least **DLGWINDOWEXTRA**.

The **DefDlgProc** function must *not* be called by a dialog box procedure; doing so results in recursive execution.

! 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

Dialog Boxes Overview, Dialog Box Functions, **DefWindowProc**, **GetClassInfo**, **RegisterClass**, **WNDCLASS**

DialogBox

The **DialogBox** macro creates a modal dialog box from a dialog box template resource. **DialogBox** does not return control until the specified callback function terminates the modal dialog box by calling the **EndDialog** function. The **DialogBox** macro uses the **DialogBoxParam** function.

```
INT_PTR DialogBox(
    HINSTANCE hInstance, // handle to module
    LPCTSTR lpTemplate, // dialog box template
    HWND hWndParent, // handle to owner window
    DLGPROC lpDialogFunc // dialog box procedure
);
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the dialog box template.

lpTemplate

[in] Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the **MAKEINTRESOURCE** macro to create this value.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

Return Values

If the function succeeds, the return value is the *nResult* parameter in the call to the **EndDialog** function used to terminate the dialog box.

If the function fails because the *hWndParent* parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is -1. To get extended error information, call **GetLastError**.

Remarks

The **DialogBox** macro uses the **CreateWindowEx** function to create the dialog box. **DialogBox** then sends a **WM_INITDIALOG** message (and a **WM_SETFONT** message, if the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style) to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the **WS_VISIBLE** style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the **EndDialog** function, **DialogBox** destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the *nResult* parameter specified by the dialog box procedure when it called **EndDialog**.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialog**, **CreateWindowEx**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DialogBoxParam**, **DialogProc**, **EndDialog**, **MAKEINTRESOURCE**, **WM_INITDIALOG**, **WM_SETFONT**

DialogBoxIndirect

The **DialogBoxIndirect** macro creates a modal dialog box from a dialog box template in memory. **DialogBoxIndirect** does not return control until the specified callback function

terminates the modal dialog box by calling the **EndDialog** function. The **DialogBoxIndirect** macro uses the **DialogBoxIndirectParam** function.

```
INT_PTR DialogBoxIndirect(
    HINSTANCE hInstance, // handle to module
    LPCDLGTEMPLATE lpTemplate, // dialog box template
    HWND hWndParent, // handle to owner window
    DLGPROC lpDialogFunc // dialog box procedure
);
```

Parameters

hInstance

[in] Handle to the module that creates the dialog box.

lpTemplate

[in] Pointer to a global memory object that contains the template that **DialogBoxIndirect** uses to create the dialog box. A dialog box template consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format.

In a standard template for a dialog box, the header is a **DLGTEMPLATE** structure, followed by additional variable-length arrays. The data for each control consists of a **DLGITEMTEMPLATE** structure, followed by additional variable-length arrays.

In an extended template for a dialog box, the header uses the **DLGTEMPLATEEX** format and the control definitions use the **DLGITEMTEMPLATEEX** format.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

Return Values

If the function succeeds, the return value is the *nResult* parameter specified in the call to the **EndDialog** function that was used to terminate the dialog box.

If the function fails because the *hWndParent* parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is -1. To get extended error information, call **GetLastError**.

Remarks

The **DialogBoxIndirect** macro uses the **CreateWindowEx** function to create the dialog box. **DialogBoxIndirect** then sends a **WM_INITDIALOG** message to the dialog box procedure. If the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style, the function also sends a **WM_SETFONT** message to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the

WS_VISIBLE style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the **EndDialog** function, **DialogBoxIndirect** destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the *nResult* parameter specified by the dialog box procedure when it called **EndDialog**.

In a standard dialog box template, the **DLGTEMPLATE** structure and each of the **DLGITEMTEMPLATE** structures must be aligned on **DWORD** boundaries. The creation data array that follows a **DLGITEMTEMPLATE** structure must also be aligned on a **DWORD** boundary. All of the other variable-length arrays in the template must be aligned on **WORD** boundaries.

In an extended dialog box template, the **DLGTEMPLATEEX** header and each of the **DLGITEMTEMPLATEEX** control definitions must be aligned on **DWORD** boundaries. The creation data array, if any, that follows a **DLGITEMTEMPLATEEX** structure must also be aligned on a **DWORD** boundary. All of the other variable-length arrays in the template must be aligned on **WORD** boundaries.

All character strings in the dialog box template, such as titles for the dialog box and buttons, must be Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateWindowEx**, **DialogBox**, **DialogBoxIndirectParam**, **DialogBoxParam**, **DialogProc**, **DLGITEMTEMPLATE**, **DLGITEMTEMPLATEEX**, **DLGTEMPLATE**, **DLGTEMPLATEEX**, **EndDialog**, **MultiByteToWideChar**, **WM_INITDIALOG**, **WM_SETFONT**

DialogBoxIndirectParam

The **DialogBoxIndirectParam** function creates a modal dialog box from a dialog box template in memory. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the *IParam* parameter of the **WM_INITDIALOG** message. An application can use this value to initialize dialog box controls.

```
INT_PTR DialogBoxIndirectParam(
    HINSTANCE hInstance,           // handle to module
    LPCDLGTEMPLATE hDialogTemplate, // dialog box template
    HWND hWndParent,             // handle to owner window
    DLGPROC lpDialogFunc,        // dialog box procedure
    LPARAM dwInitParam           // initialization value
);
```

Parameters

hInstance

[in] Handle to the module that creates the dialog box.

hDialogTemplate

[in] Pointer to a global memory object that contains the template that **DialogBoxIndirectParam** uses to create the dialog box. A dialog box template consists of a header that describes the dialog box, followed by one or more additional blocks of data that describe each of the controls in the dialog box. The template can use either the standard format or the extended format.

In a standard template for a dialog box, the header is a **DLGTEMPLATE** structure, followed by additional variable-length arrays. The data for each control consists of a **DLGITEMTEMPLATE** structure, followed by additional variable-length arrays.

In an extended template for a dialog box, the header uses the **DLGTEMPLATEEX** format and the control definitions use the **DLGITEMTEMPLATEEX** format.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *IParam* parameter of the **WM_INITDIALOG** message.

Return Values

If the function succeeds, the return value is the *nResult* parameter specified in the call to the **EndDialog** function that was used to terminate the dialog box.

If the function fails because the *hWndParent* parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is -1 . To get extended error information, call **GetLastError**.

Remarks

The **DialogBoxIndirectParam** function uses the **CreateWindowEx** function to create the dialog box. **DialogBoxIndirectParam** then sends a **WM_INITDIALOG** message to the dialog box procedure. If the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style, the function also sends a **WM_SETFONT** message to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the **WS_VISIBLE** style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the **EndDialog** function, **DialogBoxIndirectParam** destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the *nResult* parameter specified by the dialog box procedure when it called **EndDialog**.

In a standard dialog box template, the **DLGTEMPLATE** structure and each of the **DLGITEMTEMPLATE** structures must be aligned on **DWORD** boundaries. The creation data array that follows a **DLGITEMTEMPLATE** structure must also be aligned on a **DWORD** boundary. All of the other variable-length arrays in the template must be aligned on **WORD** boundaries.

In an extended dialog box template, the **DLGTEMPLATEEX** header and each of the **DLGITEMTEMPLATEEX** control definitions must be aligned on **DWORD** boundaries. The creation data array, if any, that follows a **DLGITEMTEMPLATEEX** structure must also be aligned on a **DWORD** boundary. All of the other variable-length arrays in the template must be aligned on **WORD** boundaries.

All character strings in the dialog box template, such as titles for the dialog box and buttons, must be Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateWindowEx**, **DialogBox**, **DialogBoxIndirect**, **DialogBoxParam**, **DialogProc**, **DLGITEMTEMPLATE**, **DLGITEMTEMPLATEEX**, **DLGTEMPLATE**, **DLGTEMPLATEEX**, **EndDialog**, **MultiByteToWideChar**, **WM_INITDIALOG**, **WM_SETFONT**

DialogBoxParam

The **DialogBoxParam** function creates a modal dialog box from a dialog box template resource. Before displaying the dialog box, the function passes an application-defined value to the dialog box procedure as the *IParam* parameter of the **WM_INITDIALOG** message. An application can use this value to initialize dialog box controls.

```
INT_PTR DialogBoxParam(
    HINSTANCE hInstance,      // handle to module
    LPCTSTR lpTemplateName,  // dialog box template
    HWND hWndParent,         // handle to owner window
    DLGPROC lpDialogFunc,    // dialog box procedure
    LPARAM dwInitParam       // initialization value
);
```

Parameters

hInstance

[in] Handle to the module whose executable file contains the dialog box template.

lpTemplateName

[in] Specifies the dialog box template. This parameter is either the pointer to a null-terminated character string that specifies the name of the dialog box template or an integer value that specifies the resource identifier of the dialog box template. If the parameter specifies a resource identifier, its high-order word must be zero and its low-order word must contain the identifier. You can use the **MAKEINTRESOURCE** macro to create this value.

hWndParent

[in] Handle to the window that owns the dialog box.

lpDialogFunc

[in] Pointer to the dialog box procedure. For more information about the dialog box procedure, see **DialogProc**.

dwInitParam

[in] Specifies the value to pass to the dialog box in the *IParam* parameter of the **WM_INITDIALOG** message.

Return Values

If the function succeeds, the return value is the value of the *nResult* parameter specified in the call to the **EndDialog** function used to terminate the dialog box.

If the function fails because the *hWndParent* parameter is invalid, the return value is zero. The function returns zero in this case for compatibility with previous versions of Windows. If the function fails for any other reason, the return value is -1 . To get extended error information, call **GetLastError**.

Remarks

The **DialogBoxParam** function uses the **CreateWindowEx** function to create the dialog box. **DialogBoxParam** then sends a **WM_INITDIALOG** message (and a **WM_SETFONT** message, if the template specifies the **DS_SETFONT** or **DS_SHELLFONT** style) to the dialog box procedure. The function displays the dialog box (regardless of whether the template specifies the **WS_VISIBLE** style), disables the owner window, and starts its own message loop to retrieve and dispatch messages for the dialog box.

When the dialog box procedure calls the **EndDialog** function, **DialogBoxParam** destroys the dialog box, ends the message loop, enables the owner window (if previously enabled), and returns the *nResult* parameter specified by the dialog box procedure when it called **EndDialog**.

Windows 95/98: The system can support a maximum of 255 controls per dialog box template. To place more than 255 controls in a dialog box, create the controls in the **WM_INITDIALOG** message handler, instead of placing them in the template.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateWindowEx**, **DialogBox**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DialogProc**, **EndDialog**, **MAKEINTRESOURCE**, **WM_INITDIALOG**, **WM_SETFONT**

DialogProc

The **DialogProc** function is an application-defined callback function used with the **DialogBox** function. It processes messages sent to a modal or modeless dialog box. The **DLGPROC** type defines a pointer to this callback function. **DialogProc** is a placeholder for the application-defined function name.

```
INT_PTR CALLBACK DialogProc(
    HWND hwndDlg, // handle to dialog box
```

(continued)

(continued)

```

UINT uMsg, // message
LPARAM lParam, // first message parameter
LPARAM lParam // second message parameter
);

```

Parameters

hwndDlg

[in] Handle to the dialog box.

uMsg

[in] Specifies the message.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

Typically, the dialog box procedure should return TRUE if it processed the message, and FALSE if it did not. If the dialog box procedure returns FALSE, the dialog box manager performs the default dialog box operation in response to the message.

If the dialog box procedure processes a message that requires a specific return value, the dialog box procedure should set the desired return value by calling **SetWindowLong**(*hwndDlg*, *DWL_MSGRESULT*, *lResult*) immediately before returning TRUE. Note that you must call **SetWindowLong** immediately before returning TRUE; doing so earlier might result in the *DWL_MSGRESULT* value being overwritten by a nested dialog box message.

The following messages are exceptions to the general rules stated above. Consult the documentation for the specific message for details on the semantics of the return value:

| | |
|--------------------|----------------------|
| WM_CHARTOITEM | WM_CTLCOLORSCROLLBAR |
| WM_COMPAREITEM | WM_CTLCOLORSTATIC |
| WM_CTLCOLORBTN | WM_INITDIALOG |
| WM_CTLCOLORDLG | WM_QUERYDRAGICON |
| WM_CTLCOLOREDIT | WM_VKEYTOITEM |
| WM_CTLCOLORLISTBOX | |

Remarks

You should use the dialog box procedure only if you use the dialog box class for the dialog box. This is the default class and is used when no explicit class is specified in the dialog box template. Although the dialog box procedure is similar to a window procedure, it must not call the **DefWindowProc** function to process unwanted messages. Unwanted messages are processed internally by the dialog box window procedure.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialog**, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **CreateDialogParam**, **DefWindowProc**, **DialogBox**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DialogBoxParam**, **SetFocus**, **WM_INITDIALOG**

EndDialog

The **EndDialog** function destroys a modal dialog box, causing the system to end any processing for the dialog box.

```
BOOL EndDialog(  
    HWND hDlg,           // handle to dialog box  
    INT_PTR nResult     // value to return  
);
```

Parameters

hDlg

[in] Handle to the dialog box to be destroyed.

nResult

[in] Specifies the value to be returned to the application from the function that created the dialog box.

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

Dialog boxes created by the **DialogBox**, **DialogBoxParam**, **DialogBoxIndirect**, and **DialogBoxIndirectParam** functions must be destroyed using the **EndDialog** function. An application calls **EndDialog** from within the dialog box procedure; the function must not be used for any other purpose.

A dialog box procedure can call **EndDialog** at any time, even during the processing of the **WM_INITDIALOG** message. If your application calls the function while

WM_INITDIALOG is being processed, the dialog box is destroyed before it is shown and before the input focus is set.

EndDialog does not destroy the dialog box immediately. Instead, it sets a flag and allows the dialog box procedure to return control to the system. The system checks the flag before attempting to retrieve the next message from the application queue. If the flag is set, the system ends the message loop, destroys the dialog box, and uses the value in *nResult* as the return value from the function that created the dialog box.

! 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

Dialog Boxes Overview, Dialog Box Functions, **DialogBox**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DialogBoxParam**, **WM_INITDIALOG**

GetDialogBaseUnits

The **GetDialogBaseUnits** function retrieves the system's dialog box base units, which are the average width and height of characters in the system font. For dialog boxes that use the system font, you can use these values to convert between dialog box template units, as specified in dialog box templates, and pixels. For dialog boxes that do not use the system font, the conversion from dialog template units to pixels depends on the font used by the dialog box.

For either type of dialog box, it is easier to use the **MapDialogRect** function to perform the conversion. **MapDialogRect** takes the font into account and correctly converts a rectangle from dialog box template units into pixels.

```
LONG GetDialogBaseUnits(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is a 32-bit value that contains the dialog box base units. The low-order word of the return value contains the horizontal dialog box base unit, and the high-order word contains the vertical dialog box base unit.

Remarks

The horizontal base unit returned by **GetDialogBaseUnits** is equal to the average width, in pixels, of the characters in the system font; the vertical base unit is equal to the height, in pixels, of the font.

For a dialog box that does not use the system font, the base units are the average width and height, in pixels, of the characters in the font of the dialog box. You can use the **GetTextMetrics** and **GetTextExtentPoint32** functions to calculate these values for a selected font. However, by using the **MapDialogRect** function, you can avoid errors that might result if your calculations differ from those performed by the system.

Each horizontal base unit is equal to 4 horizontal dialog box template units; each vertical base unit is equal to 8 vertical dialog box template units. Therefore, to convert dialog box template units to pixels, use the following formulas:

```
pixelX = (templateunitX * baseunitX) / 4  
pixelY = (templateunitY * baseunitY) / 8
```

Similarly, to convert from pixels to dialog box template units, use the following formulas:

```
templateunitX = (pixelX * 4) / baseunitX  
templateunitY = (pixelY * 8) / baseunitY
```

The multiplication is performed before the division to avoid rounding problems, if base units are not divisible by 4 or 8.

! 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

Dialog Boxes Overview, Dialog Box Functions, **MapDialogRect**

GetDlgCtrlID

The **GetDlgCtrlID** function returns the identifier of the specified control.

```
int GetDlgCtrlID(  
    HWND hwndCtl // handle to control  
);
```

Parameters

hwndCtl

[in] Handle to the control.

Return Values

If the function succeeds, the return value is the identifier of the control.

If the function fails, the return value is zero. An invalid value for the *hwndCtl* parameter, for example, will cause the function to fail. To get extended error information, call **GetLastError**.

Remarks

GetDlgCtrlID accepts child window handles as well as handles of controls in dialog boxes. An application sets the identifier for a child window when it creates the window by assigning the identifier value to the *hmenu* parameter when calling the **CreateWindow** or **CreateWindowEx** function.

Although **GetDlgCtrlID** may return a value if *hwndCtl* is a handle to a top-level window, top-level windows cannot have identifiers, and such a return value is never valid.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateWindow**, **CreateWindowEx**, **GetDlgItem**

GetDlgItem

The **GetDlgItem** function retrieves a handle to a control in the specified dialog box.

```
HWND GetDlgItem(  
    HWND hDlg,          // handle to dialog box  
    int nIDDlgItem     // control identifier  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the identifier of the control to be retrieved.

Return Values

If the function succeeds, the return value is the window handle of the specified control.

If the function fails, the return value is NULL, indicating an invalid dialog box handle or a nonexistent control. To get extended error information, call **GetLastError**.

Remarks

You can use the **GetDlgItem** function with any parent-child window pair, not just with dialog boxes. As long as the *hDlg* parameter specifies a parent window and the child window has a unique identifier (as specified by the *hMenu* parameter in the **CreateWindow** or **CreateWindowEx** function that created the child window), **GetDlgItem** returns a valid handle to the child window.

! 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

Dialog Boxes Overview, Dialog Box Functions, **CreateWindow**, **CreateWindowEx**, **GetDlgItemInt**, **GetDlgItemText**

GetDlgItemInt

The **GetDlgItemInt** function translates the text of a specified control in a dialog box into an integer value.

```
UINT GetDlgItemInt(  
    HWND hDlg,           // handle to dialog box  
    int nIDDlgItem,     // control identifier  
    BOOL *lpTranslated, // success state  
    BOOL bSigned        // signed or unsigned value  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control of interest.

nIDDlgItem

[in] Specifies the identifier of the control whose text is to be translated.

lpTranslated

[out] Pointer to a variable that receives a success or failure value (TRUE indicates success, FALSE indicates failure).

If this parameter is NULL, the function returns no information about success or failure.

bSigned

[in] Specifies whether the function should examine the text for a minus sign at the beginning, and return a signed integer value if it finds one (TRUE specifies this should be done, FALSE specifies that it should not).

Return Values

If the function succeeds, the variable pointed to by *lpTranslated* is set to TRUE, and the return value is the translated value of the control text.

If the function fails, the variable pointed to by *lpTranslated* is set to FALSE, and the return value is zero. Note that, since zero is a possible translated value, a return value of zero does not indicate failure by itself .

If *lpTranslated* is NULL, the function returns no information about success or failure.

If the *bSigned* parameter is TRUE, specifying that the value to be retrieved is a signed integer value, cast the return value to an **int** type. To get extended error information, call **GetLastError**.

Remarks

The **GetDlgItemInt** function retrieves the text of the specified control by sending the control a **WM_GETTEXT** message. The function translates the retrieved text by stripping any extra spaces at the beginning of the text and then converting the decimal digits. The function stops translating when it reaches the end of the text or encounters a nonnumeric character.

If the *bSigned* parameter is TRUE, the **GetDlgItemInt** function checks for a minus sign (–) at the beginning of the text and translates the text into a signed integer value. Otherwise, the function creates an unsigned integer value.

The **GetDlgItemInt** function returns zero if the translated value is greater than **INT_MAX** (for signed numbers) or **UINT_MAX** (for unsigned numbers).



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

Dialog Boxes Overview, Dialog Box Functions, **GetDlgCtrlID**, **GetDlgItem**, **GetDlgItemText**, **SetDlgItemInt**

GetDlgItemText

The **GetDlgItemText** function retrieves the title or text associated with a control in a dialog box.

```
UINT GetDlgItemText(  
    HWND hDlg,           // handle to dialog box  
    int nIDDlgItem,     // control identifier  
    LPTSTR lpString,    // pointer to buffer for text  
    int nMaxCount       // maximum size of string  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the identifier of the control whose title or text is to be retrieved.

lpString

[out] Pointer to the buffer to receive the title or text.

nMaxCount

[in] Specifies the maximum length, in characters, of the string to be copied to the buffer pointed to by *lpString*. If the length of the string exceeds the limit, the string is truncated.

Return Values

If the function succeeds, the return value specifies the number of characters copied to the buffer, not including the terminating null character.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **GetDlgItemText** function sends a **WM_GETTEXT** message to the control.

 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

Dialog Boxes Overview, Dialog Box Functions, **GetDlgItemInt**, **SetDlgItemInt**, **SetDlgItemText**, **WM_GETTEXT**

GetNextDlgGroupItem

The **GetNextDlgGroupItem** function retrieves a handle to the first control in a group of controls that precedes (or follows) the specified control in a dialog box.

```
HWND GetNextDlgGroupItem(  
    HWND hDlg,           // handle to dialog box  
    HWND hCtl,          // handle to control  
    BOOL bPrevious      // direction flag  
);
```

Parameters

hDlg

[in] Handle to the dialog box being searched.

hCtl

[in] Handle to the control to be used as the starting point for the search. If this parameter is NULL, the function uses the last (or first) control in the dialog box as the starting point for the search.

bPrevious

[in] Specifies how the function is to search the group of controls in the dialog box. If this parameter is TRUE, the function searches for the previous control in the group. If it is FALSE, the function searches for the next control in the group.

Return Values

If **GetNextDlgGroupItem** succeeds, the return value is a handle to the previous (or next) control in the group of controls.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **GetNextDlgGroupItem** function searches controls in the order (or reverse order) they were created in the dialog box template. The first control in the group must have the **WS_GROUP** style; all other controls in the group must have been created consecutively and *not* have the **WS_GROUP** style.

When searching for the previous control, the function returns the first control it locates that is visible and not disabled. If the control specified by *hCtl* has the `WS_GROUP` style, the function temporarily reverses the search to locate the first control having the `WS_GROUP` style, then resumes the search in the original direction, returning the first control it locates that is visible and not disabled, or returning *hwndCtrl* if no such control is found.

When searching for the next control, the function returns the first control it locates that is visible and not disabled, and does *not* have the `WS_GROUP` style. If it encounters a control having the `WS_GROUP` style, the function reverses the search, locates the first control having the `WS_GROUP` style, and returns this control if it is visible and not disabled. Otherwise, the function resumes the search in the original direction, and returns the first control it locates that is visible and not disabled, or returns *hCtl* if no such control is found.

If the search for the next control in the group encounters a window with the `WS_EX_CONTROLPARENT` style, the system recursively searches the window's children.

! 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

Dialog Boxes Overview, Dialog Box Functions, **GetNextDlgTabItem**

GetNextDlgTabItem

The **GetNextDlgTabItem** function retrieves a handle to the first control that has the `WS_TABSTOP` style that precedes (or follows) the specified control.

```
HWND GetNextDlgTabItem(
    HWND hDlg,           // handle to dialog box
    HWND hCtl,          // handle to known control
    BOOL bPrevious      // direction flag
);
```

Parameters

hDlg

[in] Handle to the dialog box to be searched.

hCtl

[in] Handle to the control to be used as the starting point for the search. If this parameter is NULL, the function uses the last (or first) control in the dialog box as the starting point for the search.

bPrevious

[in] Specifies how the function is to search the dialog box. If this parameter is TRUE, the function searches for the previous control in the dialog box. If this parameter is FALSE, the function searches for the next control in the dialog box.

Return Values

If the function succeeds, the return value is the window handle of the previous (or next) control that has the WS_TABSTOP style set.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The **GetNextDlgTabItem** function searches controls in the order (or reverse order) they were created in the dialog box template. The function returns the first control it locates that is visible and not disabled, and has the WS_TABSTOP style. If no such control exists, the function returns *hCtl*.

If the search for the next control with the WS_TABSTOP style encounters a window with the WS_EX_CONTROLPARENT style, the system recursively searches the window's children.

! 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

Dialog Boxes Overview, Dialog Box Functions, **GetDlgItem**, **GetNextDlgGroupItem**

IsDialogMessage

The **IsDialogMessage** function determines whether a message is intended for the specified dialog box and, if it is, processes the message.

```

BOOL IsDialogMessage(
    HWND hDlg, // handle to dialog box
    LPMSG lpMsg // message to be checked
);

```

Parameters

hDlg

[in] Handle to the dialog box.

lpMsg

[in] Pointer to an **MSG** structure that contains the message to be checked.

Return Values

If the message has been processed, the return value is nonzero.

If the message has not been processed, the return value is zero.

Remarks

Although the **IsDialogMessage** function is intended for modeless dialog boxes, you can use it with any window that contains controls, enabling the windows to provide the same keyboard selection as is used in a dialog box.

When **IsDialogMessage** processes a message, it checks for keyboard messages and converts them into selections for the corresponding dialog box. For example, the TAB key, when pressed, selects the next control or group of controls, and the DOWN ARROW key, when pressed, selects the next control in a group.

Because the **IsDialogMessage** function performs all necessary translating and dispatching of messages, a message processed by **IsDialogMessage** must not be passed to the **TranslateMessage** or **DispatchMessage** function.

IsDialogMessage sends **WM_GETDLGCODE** messages to the dialog box procedure to determine which keys should be processed.

IsDialogMessage can send **DM_GETDEFID** and **DM_SETDEFID** messages to the window. These messages are defined in the winuser.h header file as **WM_USER** and **WM_USER + 1**, so conflicts are possible with application-defined messages having the same 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 winuser.h; include windows.h.

Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Dialog Boxes Overview, Dialog Box Functions, **DispatchMessage**, **DM_GETDEFID**, **DM_SETDEFID**, **MSG**, **TranslateMessage**, **WM_GETDLGCODE**, **WM_USER**

MapDialogRect

The **MapDialogRect** function converts the specified dialog box units to screen units (pixels). The function replaces the coordinates in the specified **RECT** structure with the converted coordinates, which allows the structure to be used to create a dialog box or position a control within a dialog box.

```
BOOL MapDialogRect(
    HWND hDlg,        // handle to dialog box
    LPRECT lpRect     // dialog box coordinates
);
```

Parameters

hDlg

[in] Handle to a dialog box. This function accepts only handles returned by one of the dialog box creation functions; handles for other windows are not valid.

lpRect

[in/out] Pointer to a **RECT** structure that contains the dialog box coordinates to be converted.

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 **MapDialogRect** function assumes that the initial coordinates in the **RECT** structure represent dialog box units. To convert these coordinates from dialog box units to pixels, the function retrieves the current horizontal and vertical base units for the dialog box, then applies the following formulas:

```
left   = (left   * baseunitX) / 4
right  = (right  * baseunitX) / 4
top    = (top    * baseunitY) / 8
bottom = (bottom * baseunitY) / 8
```

In most cases, the base units for the dialog box are the same as those retrieved by using the **GetDialogBaseUnits** function. If the dialog box template has the DS_SETFONT or DS_SHELLFONT style, the base units are the average width and height, in pixels, of the characters in the font specified by the template.

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

Dialog Boxes Overview, Dialog Box Functions, **CreateDialog**, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **CreateDialogParam**, **DialogBox**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DialogBoxParam**, **GetDialogBaseUnits**, **RECT**

MessageBox

The **MessageBox** function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons.

To specify a language identifier, use the **MessageBoxEx** function.

```
int MessageBox(
    HWND hWnd,           // handle to owner window
    LPCTSTR lpText,     // text in message box
    LPCTSTR lpCaption,  // message box title
    UINT uType          // message box style
);
```

Parameters

hWnd

[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption

[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title **Error** is used.

uType

[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values:

| Value | Meaning |
|---------------------|--|
| MB_ABORTRETRYIGNORE | The message box contains three push buttons: Abort , Retry , and Ignore . |

(continued)

(continued)

| Value | Meaning |
|----------------------|---|
| MB_CANCELTRYCONTINUE | Windows 2000: The message box contains three push buttons: Cancel , Try Again , and Continue . Use this message box type instead of MB_ABORTRETRYIGNORE. |
| MB_HELP | Windows 95/98, Windows NT 4.0 and later: Adds a Help button to the message box. When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner. |
| MB_OK | The message box contains one push button: OK . This is the default. |
| MB_OKCANCEL | The message box contains two push buttons: OK and Cancel . |
| MB_RETRYCANCEL | The message box contains two push buttons: Retry and Cancel . |
| MB_YESNO | The message box contains two push buttons: Yes and No . |
| MB_YESNOCANCEL | The message box contains three push buttons: Yes , No , and Cancel . |

To display an icon in the message box, specify one of the following values:

| Value | Meaning |
|--|---|
| MB_ICONEXCLAMATION, MB_ICONWARNING | An exclamation-point icon appears in the message box. |
| MB_ICONASTERISK, MB_ICONINFORMATION | An icon consisting of a lowercase letter <i>i</i> in a circle appears in the message box. |
| MB_ICONQUESTION | A question-mark icon appears in the message box. |
| MB_ICONERROR, MB_ICONHAND, MB_ICONSTOP | A stop-sign icon appears in the message box. |

To indicate the default button, specify one of the following values:

| Value | Meaning |
|---------------|---|
| MB_DEFBUTTON1 | The first button is the default button. MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified. |
| MB_DEFBUTTON2 | The second button is the default button. |
| MB_DEFBUTTON3 | The third button is the default button. |
| MB_DEFBUTTON4 | The fourth button is the default button. |

To indicate the modality of the dialog box, specify one of the following values:

| Value | Meaning |
|----------------|---|
| MB_APPLMODAL | <p>The user must respond to the message box before continuing work in the window identified by the <i>hWnd</i> parameter. However, the user can move to the windows of other threads and work in those windows.</p> <p>Depending on the hierarchy of windows in the application, the user might be able to move to other windows within the thread. All child windows of the parent of the message box are automatically disabled, but popup windows are not.</p> <p>MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.</p> |
| MB_SYSTEMMODAL | <p>Same as MB_APPLMODAL, except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with <i>hWnd</i>.</p> |
| MB_TASKMODAL | <p>Same as MB_APPLMODAL, except that all the top-level windows belonging to the current thread are disabled if the <i>hWnd</i> parameter is NULL. Use this flag when the calling application or library does not have a window handle available, but still needs to prevent input to other windows in the calling thread without suspending other threads.</p> |

To specify other options, use one or more of the following values:

| Value | Meaning |
|-------------------------|---|
| MB_DEFAULT_DESKTOP_ONLY | <p>Windows NT/2000: Same as MB_SERVICE_NOTIFICATION, except that the system will display the message box only on the default desktop of the interactive window station. For more information, see <i>Window Stations and Desktops</i>.</p> <p>Windows NT 4.0 and earlier: If the current input desktop is not the default desktop, MessageBox fails.</p> |

(continued)

(continued)

| Value | Meaning |
|------------------------------|---|
| | <p>Windows 2000: If the current input desktop is not the default desktop, MessageBox does not return until the user switches to the default desktop.</p> <p>Windows 95/98: This flag has no effect.</p> |
| MB_RIGHT | The text is right-justified. |
| MB_RTREADING | Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems. |
| MB_SERVICE_NOTIFICATION | <p>Windows NT/2000: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer.</p> <p>Terminal Services: If the calling thread has an impersonation token, the function directs the message box to the session specified in the impersonation token.</p> <p>If this flag is set, the <i>hWnd</i> parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the <i>hWnd</i>. For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.</p> |
| MB_SERVICE_NOTIFICATION_NT3X | <p>Windows NT/2000: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51.</p> <p>For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks.</p> |
| MB_SETFOREGROUND | <p>The message box becomes the foreground window. Internally, the system calls the SetForegroundWindow function for the message box.</p> |
| MB_TOPMOST | The message box is created with the WS_EX_TOPMOST window style. |

Return Values

If the function succeeds, the return value is one of the following menu-item values:

| Value | Meaning |
|----------|------------------------------------|
| IDABORT | Abort button was selected. |
| IDCANCEL | Cancel button was selected. |

| Value | Meaning |
|------------|---------------------------------------|
| IDCONTINUE | Continue button was selected. |
| IDIGNORE | Ignore button was selected. |
| IDNO | No button was selected. |
| IDOK | OK button was selected. |
| IDRETRY | Retry button was selected. |
| IDTRYAGAIN | Try Again button was selected. |
| IDYES | Yes button was selected. |

If a message box has a **Cancel** button, the function returns the IDCANCEL value if either the ESC key is pressed or the **Cancel** button is selected. If the message box has no **Cancel** button, pressing ESC has no effect.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the *lpText* and *lpCaption* parameters should not be taken from a resource file, because an attempt to load the resource might fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the *hWnd* parameter. The *hWnd* parameter should not identify a child window, such as a control in a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

Windows NT/2000: The value of MB_SERVICE_NOTIFICATION changed, starting with Windows NT 4.0. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of **MessageBox**. This mapping is only done for executables that have a version number less than 4.0, as set by the linker.

To build a service that uses MB_SERVICE_NOTIFICATION, and can run on both Windows NT 3.x and Windows NT 4.0, you can do one of the following:

- At link-time, specify a version number less than 4.0
- At link-time, specify version 4.0. At run time, use the **GetVersionEx** function to check the system version. Then, when running on Windows NT 3.x, use MB_SERVICE_NOTIFICATION_NT3X and, on Windows NT 4.0, use MB_SERVICE_NOTIFICATION.

! 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 and Windows NT/2000.

 See Also

Dialog Boxes Overview, Dialog Box Functions, **FlashWindow**, **MessageBeep**, **MessageBoxEx**, **MessageBoxIndirect**, **SetForegroundWindow**

MessageBoxEx

The **MessageBoxEx** function creates, displays, and operates a message box. The message box contains an application-defined message and title, plus any combination of predefined icons and push buttons. The *wLanguageId* parameter specifies which set of language resources is used for the predefined push buttons.

```
int MessageBoxEx(
    HWND hWnd,           // handle to owner window
    LPCTSTR lpText,     // text in message box
    LPCTSTR lpCaption,  // message box title
    UINT uType,         // message box style
    WORD wLanguageId   // language identifier
);
```

Parameters

hWnd

[in] Handle to the owner window of the message box to be created. If this parameter is NULL, the message box has no owner window.

lpText

[in] Pointer to a null-terminated string that contains the message to be displayed.

lpCaption

[in] Pointer to a null-terminated string that contains the dialog box title. If this parameter is NULL, the default title **Error** is used.

uType

[in] Specifies the contents and behavior of the dialog box. This parameter can be a combination of flags from the following groups of flags.

To indicate the buttons displayed in the message box, specify one of the following values:

| Value | Meaning |
|---------------------|--|
| MB_ABORTRETRYIGNORE | The message box contains three push buttons: Abort , Retry , and Ignore . |

| Value | Meaning |
|----------------------|--|
| MB_CANCELTRYCONTINUE | Windows 2000: The message box contains three push buttons: Cancel , Try Again , and Continue . Use this message box type instead of MB_ABORTRETRYIGNORE. |
| MB_HELP | Windows 95/98, Windows NT 4.0 and later: Adds a Help button to the message box. When the user clicks the Help button or presses F1, the system sends a WM_HELP message to the owner. |
| MB_OK | The message box contains one push button: OK . This is the default. |
| MB_OKCANCEL | The message box contains two push buttons: OK and Cancel . |
| MB_RETRYCANCEL | The message box contains two push buttons: Retry and Cancel . |
| MB_YESNO | The message box contains two push buttons: Yes and No . |
| MB_YESNOCANCEL | The message box contains three push buttons: Yes , No , and Cancel . |

To display an icon in the message box, specify one of the following values:

| Value | Meaning |
|--|---|
| MB_ICONEXCLAMATION, MB_ICONWARNING | An exclamation-point icon appears in the message box. |
| MB_ICONASTERISK, MB_ICONINFORMATION | An icon consisting of a lowercase letter <i>i</i> in a circle appears in the message box. |
| MB_ICONQUESTION | A question-mark icon appears in the message box. |
| MB_ICONERROR, MB_ICONHAND, MB_ICONSTOP | A stop-sign icon appears in the message box. |

To indicate the default button, specify one of the following values:

| Value | Meaning |
|---------------|---|
| MB_DEFBUTTON1 | The first button is the default button. MB_DEFBUTTON1 is the default unless MB_DEFBUTTON2, MB_DEFBUTTON3, or MB_DEFBUTTON4 is specified. |
| MB_DEFBUTTON2 | The second button is the default button. |

(continued)

(continued)

| Value | Meaning |
|---------------|--|
| MB_DEFBUTTON3 | The third button is the default button. |
| MB_DEFBUTTON4 | The fourth button is the default button. |

To indicate the modality of the dialog box, specify one of the following values:

| Value | Meaning |
|----------------|---|
| MB_APPLMODAL | <p>The user must respond to the message box before continuing work in the window identified by the <i>hWnd</i> parameter. However, the user can move to the windows of other threads and work in those windows.</p> <p>Depending on the hierarchy of windows in the application, the user might be able to move to other windows within the thread. All child windows of the parent of the message box are automatically disabled, but popup windows are not.</p> <p>MB_APPLMODAL is the default if neither MB_SYSTEMMODAL nor MB_TASKMODAL is specified.</p> |
| MB_SYSTEMMODAL | <p>Same as MB_APPLMODAL, except that the message box has the WS_EX_TOPMOST style. Use system-modal message boxes to notify the user of serious, potentially damaging errors that require immediate attention (for example, running out of memory). This flag has no effect on the user's ability to interact with windows other than those associated with <i>hWnd</i>.</p> |
| MB_TASKMODAL | <p>Same as MB_APPLMODAL, except that all the top-level windows belonging to the current thread are disabled if the <i>hWnd</i> parameter is NULL. Use this flag when the calling application or library does not have a window handle available, but still needs to prevent input to other windows in the calling thread without suspending other threads.</p> |

To specify other options, use one or more of the following values:

| Values | Meaning |
|-------------------------|---|
| MB_DEFAULT_DESKTOP_ONLY | <p>Windows NT/2000: Same as MB_SERVICE_NOTIFICATION, except that the system will display the message box only on the default desktop of the interactive window station. For more information, see <i>Window Stations and Desktops</i>.</p> |

| Values | Meaning |
|------------------------------|--|
| MB_RIGHT | Windows NT 4.0 and earlier: If the current input desktop is not the default desktop, MessageBoxEx fails. |
| MBRTLREADING | Windows 2000: If the current input desktop is not the default desktop, MessageBoxEx does not return until the user switches to the default desktop. |
| MB_SERVICE_NOTIFICATION | Windows 95/98: This flag has no effect. The text is right-justified. Displays message and caption text using right-to-left reading order on Hebrew and Arabic systems. Windows NT/2000: The caller is a service notifying the user of an event. The function displays a message box on the current active desktop, even if there is no user logged on to the computer. |
| MB_SERVICE_NOTIFICATION_NT3X | Terminal Services: If the calling thread has an impersonation token, the function directs the message box to the session specified in the impersonation token. If this flag is set, the <i>hWnd</i> parameter must be NULL. This is so the message box can appear on a desktop other than the desktop corresponding to the <i>hWnd</i> . For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks. |
| MB_SETFOREGROUND | Windows NT/2000: This value corresponds to the value defined for MB_SERVICE_NOTIFICATION for Windows NT version 3.51. For more information on the changes between Windows NT 3.51 and Windows NT 4.0, see Remarks. The message box becomes the foreground window. Internally, the system calls the SetForegroundWindow function for the message box. |
| MB_TOPMOST | The message box is created with the WS_EX_TOPMOST window style. |

wLanguageId

[in] Specifies the language in which to display the text contained in the predefined push buttons. This value must be in the form returned by the **MAKELANGID** macro. For a list of the language identifiers supported by Win32, see *Language Identifiers*.

Note that each localized release of Windows 95/98 and Windows NT/Windows 2000 typically contains resources only for a limited set of languages. Thus, for example, the U.S. version offers LANG_ENGLISH, the French version offers LANG_FRENCH, the German version offers LANG_GERMAN, and the Japanese version offers LANG_JAPANESE. Each version offers LANG_NEUTRAL. This limits the set of values that can be used with the *wLanguageId* parameter. Before specifying a language identifier, you should enumerate the locales that are installed on a system.

Return Values

If the function succeeds, the return value is one of the following menu-item values:

| Value | Meaning |
|------------|---------------------------------------|
| IDABORT | Abort button was selected. |
| IDCANCEL | Cancel button was selected. |
| IDCONTINUE | Continue button was selected. |
| IDIGNORE | Ignore button was selected. |
| IDNO | No button was selected. |
| IDOK | OK button was selected. |
| IDRETRY | Retry button was selected. |
| IDTRYAGAIN | Try Again button was selected. |
| IDYES | Yes button was selected. |

If a message box has a **Cancel** button, the function returns the IDCANCEL value when either the ESC key or **Cancel** button is pressed. If the message box has no **Cancel** button, pressing the ESC key has no effect.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

When you create a system-modal message box to indicate that the system is low on memory, the strings passed as the *lpText* and *lpCaption* parameters should not be taken from a resource file, because an attempt to load the resource might fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the *hWnd* parameter. The *hWnd* parameter should not identify a child window, such as a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

Windows NT/2000: The value of `MB_SERVICE_NOTIFICATION` changed, starting with Windows NT 4.0. Windows NT 4.0 provides backward compatibility for pre-existing services by mapping the old value to the new value in the implementation of **MessageBoxEx**. This mapping is only done for executables that have a version number less than 4.0, as set by the linker.

To build a service that uses `MB_SERVICE_NOTIFICATION`, and can run on both Windows NT 3.x and Windows NT 4.0, you can do one of the following:

- At link-time, specify a version number less than 4.0
- At link-time, specify version 4.0. At run time, use the **GetVersionEx** function to check the system version. Then, when running on Windows NT 3.x, use `MB_SERVICE_NOTIFICATION_NT3X` and, on Windows NT 4.0, use `MB_SERVICE_NOTIFICATION`.

! 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 and Windows NT/2000.

+ See Also

Dialog Boxes Overview, Dialog Box Functions, **MAKELANGID**, **MessageBeep**, **MessageBox**, **MessageBoxIndirect**, **SetForegroundWindow**

MessageBoxIndirect

The **MessageBoxIndirect** function creates, displays, and operates a message box. The message box contains application-defined message text and title, any icon, and any combination of predefined push buttons.

```
int MessageBoxIndirect(
    CONST LPMSGBOXPARAMS lpMsgBoxParams // message box parameters
);
```

Parameters

lpMsgBoxParams

[in] Pointer to a **MSGBOXPARAMS** structure that contains information used to display the message box.

Return Values

If the function succeeds, the return value is one of the following menu-item values:

| Value | Meaning |
|------------|---------------------------------------|
| IDABORT | Abort button was selected. |
| IDCANCEL | Cancel button was selected. |
| IDCONTINUE | Continue button was selected. |
| IDIGNORE | Ignore button was selected. |
| IDNO | No button was selected. |
| IDOK | OK button was selected. |
| IDRETRY | Retry button was selected. |
| IDTRYAGAIN | Try Again button was selected. |
| IDYES | Yes button was selected. |

If a message box has a **Cancel** button, the function returns the IDCANCEL value if either the ESC key is pressed or the **Cancel** button is selected. If the message box has no **Cancel** button, pressing ESC has no effect.

If there is not enough memory to create the message box, the return value is zero.

Remarks

When you use a system-modal message box to indicate that the system is low on memory, the strings pointed to by the **lpszText** and **lpszCaption** members of the **MSGBOXPARAMS** structure should not be taken from a resource file, because an attempt to load the resource might fail.

If you create a message box while a dialog box is present, use a handle to the dialog box as the *hWnd* parameter. The *hWnd* parameter should not identify a child window, such as a control in a dialog box.

Windows 95: The system can support a maximum of 16,364 window handles.

! 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

Dialog Boxes Overview, Dialog Box Functions, **MessageBox**, **MessageBoxEx**, **MSGBOXPARAMS**

SendDlgItemMessage

The **SendDlgItemMessage** function sends a message to the specified control in a dialog box.

```
LRESULT SendDlgItemMessage(  
    HWND hDlg,           // handle to dialog box  
    int nIDDlgItem,     // control identifier  
    UINT Msg,           // message to send  
    WPARAM wParam,     // first message parameter  
    LPARAM lParam       // second message parameter  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the identifier of the control that receives the message.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message sent.

Remarks

The **SendDlgItemMessage** function does not return until the message has been processed.

Using **SendDlgItemMessage** is identical to retrieving a handle to the specified control and calling the **SendMessage** 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.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

 See Also

Dialog Boxes Overview, Dialog Box Functions, **SendMessage**

SetDlgItemInt

The **SetDlgItemInt** function sets the text of a control in a dialog box to the string representation of a specified integer value.

```
BOOL SetDlgItemInt(  
    HWND hDlg,          // handle to dialog box  
    int nIDDlgItem,    // control identifier  
    UINT uValue,       // value to set  
    BOOL bSigned       // signed or unsigned indicator  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the control to be changed.

uValue

[in] Specifies the integer value used to generate the item text.

bSigned

[in] Specifies whether the *uValue* parameter is signed or unsigned. If this parameter is TRUE, *uValue* is signed. If this parameter is TRUE and *uValue* is less than zero, a minus sign is placed before the first digit in the string. If this parameter is FALSE, *uValue* is unsigned.

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

To set the new text, this function sends a **WM_SETTEXT** message to the specified control.

 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

Dialog Boxes Overview, Dialog Box Functions, **GetDlgItemInt**, **SetDlgItemText**, **WM_SETTEXT**

SetDlgItemText

The **SetDlgItemText** function sets the title or text of a control in a dialog box.

```
BOOL SetDlgItemText(  
    HWND hDlg,           // handle to dialog box  
    int nIDDlgItem,     // control identifier  
    LPCTSTR lpString    // text to set  
);
```

Parameters

hDlg

[in] Handle to the dialog box that contains the control.

nIDDlgItem

[in] Specifies the control with a title or text to be set.

lpString

[in] Pointer to the null-terminated string that contains the text to be copied to the control.

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 **SetDlgItemText** function sends a **WM_SETTEXT** message to the specified control.

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

Dialog Boxes Overview, Dialog Box Functions, **GetDlgItemInt**, **GetDlgItemText**, **SetDlgItemInt**, **WM_SETTEXT**

Dialog Box Structures

DLGITEMTEMPLATE

The **DLGITEMTEMPLATE** structure defines the dimensions and style of a control in a dialog box. One or more of these structures are combined with a **DLGTEMPLATE** structure to form a standard template for a dialog box.

```
typedef struct {
    DWORD style;
    DWORD dwExtendedStyle;
    short x;
    short y;
    short cx;
    short cy;
    WORD id;           // on Windows 95/98, this is a byte.
} DLGITEMTEMPLATE, *PDLGITEMTEMPLATE;
```

Members

style

Specifies the style of the control. This member can be a combination of window style values (such as **WS_BORDER**) and one or more of the control style values (such as **BS_PUSHBUTTON** or **ES_LEFT**).

dwExtendedStyle

Specifies extended styles for a window. This member is not used to create controls in dialog boxes, but applications that use dialog box templates can use it to create other types of windows.

x

Specifies the x-coordinate, in dialog box units, of the upper-left corner of the control. This coordinate is always relative to the upper-left corner of the dialog box's client area.

y

Specifies the y-coordinate, in dialog box units, of the upper-left corner of the control. This coordinate is always relative to the upper-left corner of the dialog box's client area.

cx

Specifies the width, in dialog box units, of the control.

cy

Specifies the height, in dialog box units, of the control.

id

Specifies the control identifier.

Windows 95/98: Only the first byte is used (that is, 255 is the maximum).

Remarks

In a standard template for a dialog box, the **DLGITEMTEMPLATE** structure is always immediately followed by three variable-length arrays specifying the class, title, and creation data for the control. Each array consists of one or more 16-bit elements.

Each **DLGITEMTEMPLATE** structure in the template must be aligned on a **DWORD** boundary. The class and title arrays must be aligned on **WORD** boundaries. The creation data array must be aligned on a **WORD** boundary.

Immediately following each **DLGITEMTEMPLATE** structure is a class array that specifies the window class of the control. If the first element of this array is any value other than 0xFFFF, the system treats the array as a null-terminated Unicode string that specifies the name of a registered window class. If the first element is 0xFFFF, the array has one additional element that specifies the ordinal value of a predefined system class. The ordinal can be one of the following atom values:

| Value | Meaning |
|--------------|----------------|
| 0x0080 | Button |
| 0x0081 | Edit |
| 0x0082 | Static |
| 0x0083 | List box |
| 0x0084 | Scroll bar |
| 0x0085 | Combo box |

Following the class array is a title array that contains the initial text or resource identifier of the control. If the first element of this array is 0xFFFF, the array has one additional element that specifies an ordinal value of a resource, such as an icon, in an executable file. You can use a resource identifier for controls, such as static icon controls, that load and display an icon or other resource rather than text. If the first element is any value other than 0xFFFF, the system treats the array as a null-terminated Unicode string that specifies the initial text.

The creation data array begins at the next **WORD** boundary after the title array. This creation data can be of any size and format. If the first word of the creation data array is nonzero, it indicates the size, in bytes, of the creation data (including the size word). The control's window procedure must be able to interpret the data. When the system creates the control, it passes a pointer to this data in the *IParam* parameter of the **WM_CREATE** message that it sends to the control.

If you specify character strings in the class and title arrays, you must use Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

The **x**, **y**, **cx**, and **cy** members specify values in dialog box units. You can convert these values to screen units (pixels) by using the **MapDialogRect** 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.

+ See Also

Dialog Boxes Overview, Dialog Box Structures, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **CreateWindowEx**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DLGITEMTEMPLATEEX**, **DLGTEMPLATE**, **DLGTEMPLATEEX**, **MapDialogRect**, **MultiByteToWideChar**, **WM_CREATE**

DLGITEMTEMPLATEEX

The **DLGITEMTEMPLATEEX** structure is not defined in any standard header file. The structure definition is provided here to explain the format of an extended template for a dialog box.

For each control in a dialog box, an extended dialog box template has a block of data that uses the **DLGITEMTEMPLATEEX** format to describe the control. For a description of the format of an extended dialog box template, see **DLGTEMPLATEEX**.

```
typedef struct {
    DWORD    helpID;
    DWORD    exStyle;
    DWORD    style;
    short    x;
    short    y;
    short    cx;
    short    cy;
    WORD     id;
    sz_0r_0rd windowClass;
    sz_0r_0rd title;
    WORD     extraCount;
} DLGITEMTEMPLATEEX;
```

Members

helpID

Specifies the help context identifier for the control. When the system sends a **WM_HELP** message, it passes the **helpID** value in the **dwContextId** member of the **HELPINFO** structure.

exStyle

Specifies extended styles for a window. This member is not used to create controls in dialog boxes, but applications that use dialog box templates can use it to create other types of windows.

style

Specifies the style of the control. This member can be a combination of window style values (such as `WS_BORDER`) and one or more of the control style values (such as `BS_PUSHBUTTON` or `ES_LEFT`).

x

Specifies the x-coordinate, in dialog box units, of the upper-left corner of the control. This coordinate is always relative to the upper-left corner of the dialog box's client area.

y

Specifies the y-coordinate, in dialog box units, of the upper-left corner of the control. This coordinate is always relative to the upper-left corner of the dialog box's client area.

cx

Specifies the width, in dialog box units, of the control.

cy

Specifies the height, in dialog box units, of the control.

id

Specifies the control identifier.

windowClass

Specifies a variable-length array of 16-bit elements that specifies the window class of the control. If the first element of this array is any value other than `0xFFFF`, the system treats the array as a null-terminated Unicode string that specifies the name of a registered window class.

If the first element is `0xFFFF`, the array has one additional element that specifies the ordinal value of a predefined system class. The ordinal can be one of the following atom values:

| Value | Meaning |
|--------|------------|
| 0x0080 | Button |
| 0x0081 | Edit |
| 0x0082 | Static |
| 0x0083 | List box |
| 0x0084 | Scroll bar |
| 0x0085 | Combo box |

title

Specifies a variable-length array of 16-bit elements that contains the initial text or resource identifier of the control. If the first element of this array is `0xFFFF`, the array has one additional element that specifies the ordinal value of a resource, such as an

icon, in an executable file. You can use a resource identifier for controls, such as static icon controls, that load and display an icon or other resource rather than text. If the first element is any value other than 0xFFFF, the system treats the array as a null-terminated Unicode string that specifies the initial text.

extraCount

Specifies the number of bytes of creation data that follow this member. If this value is greater than zero, the creation data begins at the next **WORD** boundary. This creation data can be of any size and format. The control's window procedure must be able to interpret the data. When the system creates the control, it passes a pointer to this data in the *IParam* parameter of the **WM_CREATE** message that it sends to the control.

Remarks

An extended template for a dialog box consists of a **DLGTEMPLATEEX** header, followed by a **DLGITEMTEMPLATEEX** structure for each control in the dialog box.

Each **DLGITEMTEMPLATEEX** structure must be aligned on a **DWORD** boundary. The variable-length **windowClass** and **title** arrays must be aligned on **WORD** boundaries. The creation data array, if any, must be aligned on a **WORD** boundary.

If you specify character strings in the **windowClass** and **title** arrays, you must use Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

The **x**, **y**, **cx**, and **cy** members specify values in dialog box units. You can convert these values to screen units (pixels) by using the **MapDialogRect** function.

! Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.

+ See Also

Dialog Boxes Overview, *Dialog Box Structures*, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **CreateWindowEx**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DLGTEMPLATEEX**, **MapDialogRect**, **MultiByteToWideChar**, **WM_CREATE**, **WM_HELP**

DLGTEMPLATE

The **DLGTEMPLATE** structure defines the dimensions and style of a dialog box. This structure, which is always the first in a standard template for a dialog box, also specifies

the number of controls in the dialog box and, therefore, specifies the number of subsequent **DLGITEMTEMPLATE** structures in the template.

```
typedef struct {  
    DWORD style;  
    DWORD dwExtendedStyle;  
    WORD cdit;  
    short x;  
    short y;  
    short cx;  
    short cy;  
} DLGTEMPLATE, *LPDLGTEMPLATE;
```

Members

style

Specifies the style of the dialog box. This member can be a combination of window style values (such as **WS_CAPTION** or **WS_SYSMENU**) and dialog box style values (such as **DS_CENTER**).

For a list of window styles, see *CreateWindow*. For a list of dialog box styles, see *Dialog Box Template Styles*.

If the style member includes the **DS_SETFONT** style, the header of the dialog box template contains additional data specifying the font to use for text in the client area and controls of the dialog box. The font data begins on the **WORD** boundary that follows the title array. The font data specifies a 16-bit point size value and a Unicode font name string. If possible, the system creates a font according to the specified values. Then, the system sends a **WM_SETFONT** message to the dialog box and to each control to provide a handle to the font. If **DS_SETFONT** is not specified, the dialog box template does not include the font data.

Windows 2000: The **DS_SHELLFONT** style is not supported in the **DLGTEMPLATE** header.

dwExtendedStyle

Specifies extended styles for a window. This member is not used to create dialog boxes, but applications that use dialog box templates can use it to create other types of windows.

cdit

Specifies the number of items in the dialog box.

x

Specifies the x-coordinate, in dialog box units, of the upper-left corner of the dialog box.

y

Specifies the y-coordinate, in dialog box units, of the upper-left corner of the dialog box.

cx

Specifies the width, in dialog box units, of the dialog box.

cy

Specifies the height, in dialog box units, of the dialog box.

Remarks

In a standard template for a dialog box, the **DLGTEMPLATE** structure is always immediately followed by three variable-length arrays that specify the menu, class, and title for the dialog box. When the **DS_SETFONT** style is specified, these arrays are also followed by a 16-bit value specifying point size and another variable-length array specifying a typeface name. Each array consists of one or more 16-bit elements. The menu, class, title, and font arrays must be aligned on **WORD** boundaries.

Immediately following the **DLGTEMPLATE** structure is a menu array that identifies a menu resource for the dialog box. If the first element of this array is 0x0000, the dialog box has no menu and the array has no other elements. If the first element is 0xFFFF, the array has one additional element that specifies the ordinal value of a menu resource in an executable file. If the first element has any other value, the system treats the array as a null-terminated Unicode string that specifies the name of a menu resource in an executable file.

Following the menu array is a class array that identifies the window class of the control. If the first element of the array is 0x0000, the system uses the predefined dialog box class for the dialog box and the array has no other elements. If the first element is 0xFFFF, the array has one additional element that specifies the ordinal value of a predefined system window class. If the first element has any other value, the system treats the array as a null-terminated Unicode string that specifies the name of a registered window class.

Following the class array is a title array that specifies a null-terminated Unicode string that contains the title of the dialog box. If the first element of this array is 0x0000, the dialog box has no title and the array has no other elements.

The 16-bit point size value and the typeface array follow the title array, but only if the **style** member specifies the **DS_SETFONT** style. The point-size value specifies the point size of the font to use for the text in the dialog box and its controls. The typeface array is a null-terminated Unicode string specifying the name of the typeface for the font. When these values are specified, the system creates a font having the specified size and typeface (if possible), and sends a **WM_SETFONT** message to the dialog box procedure and the control window procedures as it creates the dialog box and controls.

Following the **DLGTEMPLATE** header in a standard dialog box template are one or more **DLGITEMTEMPLATE** structures that define the dimensions and style of the controls in the dialog box. The **cdit** member specifies the number of **DLGITEMTEMPLATE** structures in the template. These **DLGITEMTEMPLATE** structures must be aligned on **DWORD** boundaries.

If you specify character strings in the menu, class, title, or typeface arrays, you must use Unicode strings. To create code that works on both Windows 95/98 and

Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

The **x**, **y**, **cx**, and **cy** members specify values in dialog box units. You can convert these values to screen units (pixels) by using the **MapDialogRect** 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`.

+ See Also

Dialog Boxes Overview, Dialog Box Structures, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DLGITEMTEMPLATE**, **DLGITEMTEMPLATEEX**, **DLGTEMPLATEEX**, **MapDialogRect**, **MultiByteToWideChar**

DLGTEMPLATEEX

The **DLGTEMPLATEEX** structure is not defined in any standard header file. The structure definition is provided here to explain the format of an extended template for a dialog box.

An extended dialog box template begins with a **DLGTEMPLATEEX** header that describes the dialog box and specifies the number of controls in the dialog box. For each control in a dialog box, an extended dialog box template has a block of data that uses the **DLGITEMTEMPLATEEX** format to describe the control.

```
typedef struct {
    WORD        dlgVer;
    WORD        signature;
    DWORD       helpID;
    DWORD       exStyle;
    DWORD       style;
    WORD        cDlgItems;
    short       x;
    short       y;
    short       cx;
    short       cy;
    sz_Or_Ord  menu;
    sz_Or_Ord  windowClass;
    WCHAR       title[titleLen];
    // The following members exist only if the style member is
```

(continued)

(continued)

```
// set to DS_SETFONT or DS_SHELLFONT.
WORD    fontsize;
WORD    weight;
BYTE    italic;
BYTE    charset;
WCHAR   typeface[stringLen];
} DLGTEMPLATEEX;
```

Members

dlgVer

Specifies the version number of the extended dialog box template; this member must be 1.

signature

Indicates whether a template is an extended dialog box template. If **signature** is 0xFFFF, this is an extended dialog box template. In this case, the **dlgVer** member specifies the template version number.

If **signature** is any value other than 0xFFFF, this is a standard dialog box template that uses the **DLGTEMPLATE** and **DLGITEMTEMPLATE** structures.

helpID

Specifies the help context identifier for the dialog box window. When the system sends a **WM_HELP** message, it passes this value in the **dwContextId** member of the **HELPINFO** structure.

exStyle

Specifies extended windows styles. This member is not used when creating dialog boxes, but applications that use dialog box templates can use it to create other types of windows.

For a list of extended window styles, see **CreateWindowEx**.

style

Specifies the style of the dialog box. This member can be a combination of window style values and dialog box style values. For a list of window styles, see **CreateWindow**. For a list of dialog box styles, see *Dialog Box Template Styles*.

If **style** includes the DS_SETFONT or DS_SHELLFONT dialog box style, the **DLGTEMPLATEEX** header of the extended dialog box template contains four additional members (**fontsize**, **weight**, **italic**, and **typeface**) that describe the font to use for the text in the client area and controls of the dialog box. If possible, the system creates a font according to the values specified in these members. Then, the system sends a **WM_SETFONT** message to the dialog box and to each control to provide a handle to the font.

For more information, see *Dialog Box Fonts*.

cdlItems

Specifies the number of controls in the dialog box.

x

Specifies the x-coordinate, in dialog box units, of the upper-left corner of the dialog box.

y

Specifies the y-coordinate, in dialog box units, of the upper-left corner of the dialog box.

cx

Specifies the width, in dialog box units, of the dialog box.

cy

Specifies the height, in dialog box units, of the dialog box.

menu

Specifies a variable-length array of 16-bit elements that identifies a menu resource for the dialog box. If the first element of this array is 0x0000, the dialog box has no menu and the array has no other elements. If the first element is 0xFFFF, the array has one additional element that specifies the ordinal value of a menu resource in an executable file. If the first element has any other value, the system treats the array as a null-terminated Unicode string that specifies the name of a menu resource in an executable file.

windowClass

Specifies a variable-length array of 16-bit elements that identifies the window class of the dialog box. If the first element of the array is 0x0000, the system uses the predefined dialog box class for the dialog box and the array has no other elements. If the first element is 0xFFFF, the array has one additional element that specifies the ordinal value of a predefined system window class. If the first element has any other value, the system treats the array as a null-terminated Unicode string that specifies the name of a registered window class.

title

Specifies a null-terminated Unicode string that contains the title of the dialog box. If the first element of this array is 0x0000, the dialog box has no title and the array has no other elements.

pointsize

Specifies the point size of the font to use for the text in the dialog box and its controls.

This member is present only if the **style** member specifies DS_SETFONT or DS_SHELLFONT.

weight

Specifies the weight of the font in the range 0 through 1000. This can be any of the values listed for the **IfWeight** member of the **LOGFONT** structure.

This member is present only if the **style** member specifies DS_SETFONT or DS_SHELLFONT.

italic

Indicates whether the font is italic. If this value is TRUE, the font is italic.

This member is present only if the **style** member specifies DS_SETFONT or DS_SHELLFONT.

charset

Indicates the character set to use. For more information, see the **lfcharset** member of **LOGFONT**.

This member is present only if the **style** member specifies **DS_SETFONT** or **DS_SHELLFONT**.

typeface

Specifies a null-terminated Unicode string that contains the name of the typeface for the font.

This member is present only if the **style** member specifies **DS_SETFONT** or **DS_SHELLFONT**.

Remarks

You can use an extended dialog box template, instead of a standard dialog box template, in the **CreateDialogIndirectParam** and **DialogBoxIndirectParam** functions and in the **CreateDialogIndirect** and **DialogBoxIndirect** macros.

Following the **DLGTEMPLATEEX** header in an extended dialog box template is one or more **DLGITEMTEMPLATEEX** structures that describe the controls of the dialog box. The **cdlItems** member of the **DLGITEMTEMPLATEEX** structure specifies the number of **DLGITEMTEMPLATEEX** structures that follow in the template.

Each **DLGITEMTEMPLATEEX** structure in the template must be aligned on a **DWORD** boundary. If the **style** member specifies the **DS_SETFONT** or **DS_SHELLFONT** style, the first **DLGITEMTEMPLATEEX** structure begins on the first **DWORD** boundary after the **typeface** string. If these styles are not specified, the first structure begins on the first **DWORD** boundary after the **title** string.

The **menu**, **windowClass**, **title**, and **typeface** arrays must be aligned on **WORD** boundaries.

If you specify character strings in the **menu**, **windowClass**, **title**, and **typeface** arrays, you must use Unicode strings. To create code that works on both Windows 95/98 and Windows NT/Windows 2000, use the **MultiByteToWideChar** function to generate these Unicode strings.

The **x**, **y**, **cx**, and **cy** members specify values in dialog box units. You can convert these values to screen units (pixels) by using the **MapDialogRect** function.

**Requirements**

Windows NT/2000: Requires Windows NT 3.51 or later.

Windows 95/98: Requires Windows 95 or later.

Windows CE: Unsupported.

+ See Also

Dialog Boxes Overview, Dialog Box Structures, **CreateDialogIndirect**, **CreateDialogIndirectParam**, **DialogBoxIndirect**, **DialogBoxIndirectParam**, **DLGITEMTEMPLATEEX**, **LOGFONT**, **MapDialogRect**, **MultiByteToWideChar**, **WM_SETFONT**

MSGBOXPARAMS

The **MSGBOXPARAMS** structure contains information used to display a message box. The **MessageBoxIndirect** function uses this structure.

```
typedef struct {
    UINT        cbSize;
    HWND        hwndOwner;
    HINSTANCE   hInstance;
    LPCTSTR     lpszText;
    LPCTSTR     lpszCaption;
    DWORD       dwStyle;
    LPCTSTR     lpszIcon;
    DWORD_PTR   dwContextHelpId;
    MSGBOXCALLBACK lpfnMsgBoxCallback;
    DWORD       dwLanguageId;
} MSGBOXPARAMS, *PMSGBOXPARAMS;
```

Members

cbSize

Specifies the structure size, in bytes.

hwndOwner

Handle to the owner window. This member can be NULL.

hInstance

Handle to the module that contains the icon resource identified by the **lpszIcon** member, and the string resource identified by the **lpszText** or **lpszCaption** member.

lpszText

Pointer to a null-terminated string, or the identifier of a string resource, that contains the message to be displayed.

lpszCaption

Pointer to a null-terminated string, or the identifier of a string resource, that contains the message box title. If this member is NULL, the default title **Error** is used.

dwStyle

Specifies the contents and behavior of the dialog box. This member can be a combination of flags described for the *uType* parameter of the **MessageBoxEx** function.

In addition, you can specify the **MB_USERICON** flag, if you want the message box to display the icon specified by the **lpszIcon** member.

lpszIcon

Identifies an icon resource. This parameter can be either a null-terminated string or an integer resource identifier passed to the **MAKEINTRESOURCE** macro.

To load one of the standard system-defined icons, set the **hInstance** member to **NULL** and the **lpszIcon** member to one of the values listed with the **LoadIcon** function.

This member is ignored if the **dwStyle** member does not specify the **MB_USERICON** flag.

dwContextHelpId

Identifies a help context. If a help event occurs, this value is specified in the **HELPINFO** structure that the message box sends to the owner window or callback function.

lpfnMsgBoxCallback

Pointer to the callback function that processes help events for the message box. The callback function has the following form:

```
VOID CALLBACK MsgBoxCallback(LPHELPINFO lpHelpInfo);
```

If this member is **NULL**, the message box sends **WM_HELP** messages to the owner window when help events occur.

dwLanguageId

Specifies the language in which to display the text contained in the predefined push buttons. This value must be in the form returned by the **MAKELANGID** macro.

For a list of supported language identifiers, see *Language Identifiers*. Note that each localized release of Windows 95/98 and Windows NT/Windows 2000 typically contains resources only for a limited set of languages. Thus, for example, the U.S. version offers **LANG_ENGLISH**, the French version offers **LANG_FRENCH**, the German version offers **LANG_GERMAN**, and the Japanese version offers **LANG_JAPANESE**. Each version offers **LANG_NEUTRAL**. This limits the set of values that can be used with the *wLanguageId* parameter. Before specifying a language identifier, you should enumerate the locales that are installed on a system.

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`.

Unicode: Declared as Unicode and ANSI structures.

+ See Also

Dialog Boxes Overview, Dialog Box Structures, **HELPINFO**, **LoadIcon**, **MAKEINTRESOURCE**, **MAKELANGID**, **MessageBoxEx**, **MessageBoxIndirect**, **WM_HELP**

Dialog Box Messages

The following messages are used to create and manage dialog boxes and controls within dialog boxes:

DM_GETDEFID

An application sends a **DM_GETDEFID** message to retrieve the identifier of the default push-button control for a dialog box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    DM_GETDEFID,         // message to send  
    (WPARAM) wParam,     // not used; must be zero  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

If a default push button exists, the high-order word of the return value contains the value **DC_HASDEFID** and the low-order word contains the control identifier. Otherwise, the return value is zero.

Remarks

The **DefDlgProc** function processes this message.

! 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

Dialog Boxes Overview, Dialog Box Messages, **DefDlgProc**, **DM_SETDEFID**

DM_REPOSITION

The **DM_REPOSITION** message repositions a top-level dialog box, so that it fits within the desktop area. An application can send this message to a dialog box after resizing it to ensure that the entire dialog box remains visible.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    DM_REPOSITION,       // message to send  
    (LPARAM) wParam,     // not used; must be zero  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

This message has no return value.

Remarks

This message has no effect if the dialog box is a child window.

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

Dialog Boxes Overview, Dialog Box Messages

DM_SETDEFID

An application sends a **DM_SETDEFID** message to change the identifier of the default push button for a dialog box.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    DM_SETDEFID,         // message to send  
    (LPARAM) wParam;     // button identifier  
);
```

```
(LPARAM) lParam;           // not used; must be zero
);
```

Parameters

wParam

Specifies the identifier of a push-button control that will become the default.

lParam

This parameter is not used.

Remarks

This message is processed by the **DefDlgProc** function. To set the default push button, the function can send **WM_GETDLGCODE** and **BM_SETSTYLE** messages to both the specified control and the current default push button.

Using the **DM_SETDEFID** message can result in more than one button appearing to have the default push-button state. When the system brings up a dialog box, it draws the first push button in the dialog box template with the default state border. Sending a **DM_SETDEFID** message to change the default button will not always remove the default state border from the first push button. In these cases, the application should send a **BM_SETSTYLE** message to change the first push-button border style.

Return Values

The return value is always TRUE.

! 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

Dialog Boxes Overview, Dialog Box Messages, **BM_SETSTYLE**, **DefDlgProc**, **DM_GETDEFID**, **EM_SETLIMITTEXT**, **WM_GETDLGCODE**

WM_CTLCOLORDLG

The **WM_CTLCOLORDLG** message is sent to a dialog box before the system draws the dialog box. By responding to this message, the dialog box can set its text and background colors using the specified display device context handle.

A window receives this message through its **WindowProc** function.

```
LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
```

(continued)

(continued)

```
UINT uMsg,          // WM_CTLCOLORDLG
WPARAM wParam,     // handle to DC (HDC)
LPARAM lParam      // handle to dialog box (HWND)
);
```

Parameters

wParam

Handle to the device context for the dialog box.

lParam

Handle to the dialog box.

Return Values

If an application processes this message, it must return a handle to a brush. The system uses the brush to paint the background of the dialog box.

Remarks

By default, the **DefWindowProc** function selects the default system colors for the dialog box.

The system does not automatically destroy the returned brush. It is the application's responsibility to destroy the brush when it is no longer needed.

The **WM_CTLCOLORDLG** message is never sent between threads. It is sent within one thread only .

Note that the **WM_CTLCOLORDLG** message is sent to the dialog box itself; all of the other **WM_CTLCOLOR*** messages are sent to the owner of the control.

If a dialog box procedure handles this message, it should cast the desired return value to a **BOOL** and return the value directly. If the dialog box procedure returns **FALSE**, then default message handling is performed. The **DWL_MSGRESULT** value set by the **SetWindowLong** function is ignored.

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

Dialog Boxes Overview, Dialog Box Messages, **DefWindowProc**, **RealizePalette**, **SelectPalette**, **SetWindowLong**

WM_ENTERIDLE

The **WM_ENTERIDLE** message is sent to the owner window of a modal dialog box or menu that is entering an idle state. A modal dialog box or menu enters an idle state when no messages are waiting in its queue, after it has processed one or more previous messages.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,         // WM_ENTERIDLE
    WPARAM wParam,     // idle reason
    LPARAM lParam      // handle to dialog box or owner (HWND)
);

```

Parameters

wParam

Specifies whether the message is the result of a dialog box or a menu being displayed. This parameter can be one of the following values:

| Value | Meaning |
|----------------|---|
| MSGF_DIALOGBOX | The system is idle because a dialog box is displayed. |
| MSGF_MENU | The system is idle because a menu is displayed. |

lParam

Handle to the dialog box (if *wParam* is MSGF_DIALOGBOX) or window containing the displayed menu (if *wParam* is MSGF_MENU).

Return Values

An application should return zero if it processes this message.

Remarks

You can suppress the **WM_ENTERIDLE** message for a dialog box by creating the dialog box with the DS_NOIDLEMSG style.

! 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

Dialog Boxes Overview, Dialog Box Messages, **DefWindowProc**

WM_GETDLGCODE

The **WM_GETDLGCODE** message is sent to the window procedure associated with a control. By default, the system handles all keyboard input to the control; the system interprets certain types of keyboard input as dialog box navigation keys. To override this default behavior, the control can respond to the **WM_GETDLGCODE** message to indicate the types of input it wants to process itself.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,          // handle to window
    UINT uMsg,          // WM_GETDLGCODE
    WPARAM wParam,      // not used
    LPARAM lParam       // message information
);

```

Parameters

wParam

This parameter is not used.

lParam

Pointer to an **MSG** structure (or NULL, if the system is performing a query).

Return Values

The return value is one or more of the following values, indicating which type of input the application processes:

| Value | Meaning |
|----------------------|---|
| DLGC_BUTTON | Button |
| DLGC_DEFPUSHBUTTON | Default push button |
| DLGC_HASSETSEL | EM_SETSEL messages |
| DLGC_RADIOBUTTON | Radio button |
| DLGC_STATIC | Static control |
| DLGC_UNDEFPUSHBUTTON | Non-default push button |
| DLGC_WANTALLKEYS | All keyboard input |
| DLGC_WANTARROWS | Direction keys |
| DLGC_WANTCHARS | WM_CHAR messages |
| DLGC_WANTMESSAGE | All keyboard input (the application passes this message in the MSG structure to the control) |
| DLGC_WANTTAB | TAB key |

Remarks

Although the **DefWindowProc** function always returns zero in response to the **WM_GETDLGCODE** message, the window procedure for the predefined control classes returns a code appropriate for each class.

The **WM_GETDLGCODE** message and the returned values are useful only with user-defined dialog box controls or standard controls modified by subclassing.

! 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

Dialog Boxes Overview, Dialog Box Messages, **DefWindowProc**, **MSG**, **EM_SETSEL**, **WM_CHAR**

WM_INITDIALOG

The **WM_INITDIALOG** message is sent to the dialog box procedure immediately before a dialog box is displayed. Dialog box procedures typically use this message to initialize controls and carry out any other initialization tasks that affect the appearance of the dialog box.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_INITDIALOG
    WPARAM wParam,      // handle to control (HWND)
    LPARAM lParam       // initialization parameter
);

```

Parameters

wParam

Handle to the control to receive the default keyboard focus. The system assigns the default keyboard focus only if the dialog box procedure returns **TRUE**.

lParam

Specifies additional initialization data. This data is passed to the system as the *lParam* parameter in a call to the **CreateDialogIndirectParam**, **CreateDialogParam**, **DialogBoxIndirectParam**, or **DialogBoxParam** function used to create the dialog box. For property sheets, this parameter is a pointer to the **PROPSHEETPAGE** structure used to create the page. This parameter is zero if any other dialog box creation function is used.

Return Values

The dialog box procedure should return `TRUE` to direct the system to set the keyboard focus to the control specified by *wParam*. Otherwise, it should return `FALSE` to prevent the system from setting the default keyboard focus.

The dialog box procedure should return the value directly. The `DWL_MSGRESULT` value set by the **SetWindowLong** function is ignored.

Remarks

The control to receive the default keyboard focus is always the first control in the dialog box that is visible and not disabled, and that has the `WS_TABSTOP` style. When the dialog box procedure returns `TRUE`, the system checks the control to ensure that the procedure has not disabled it. If it has been disabled, the system sets the keyboard focus to the next control that is visible and not disabled, and that has the `WS_TABSTOP`.

An application can return `FALSE` only if it has set the keyboard focus to one of the controls of the dialog box.

! 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

Dialog Boxes Overview, Dialog Box Messages, **CreateDialogIndirectParam**, **CreateDialogParam**, **DialogBoxIndirectParam**, **DialogBoxParam**, **PROPSHEETPAGE**, **SetFocus**

WM_NEXTDLGCTL

The `WM_NEXTDLGCTL` message is sent to a dialog box procedure to set the keyboard focus to a different control in the dialog box.

A window receives this message through its **WindowProc** function.

```

LRESULT CALLBACK WindowProc(
    HWND hwnd,           // handle to window
    UINT uMsg,          // WM_NEXTDLGCTL
    WPARAM wParam,      // control identifier
    LPARAM lParam       // wParam usage
);

```

Parameters

wParam

If *IParam* is TRUE, this parameter identifies the control that receives the focus. If *IParam* is FALSE, this parameter indicates whether the next or previous control with the WS_TABSTOP style receives the focus. If *wParam* is zero, the next control receives the focus; otherwise, the previous control with the WS_TABSTOP style receives the focus.

IParam

The low-order word indicates how the system uses *wParam*. If the low-order word is TRUE, *wParam* is a handle associated with the control that receives the focus; otherwise, *wParam* is a flag that indicates whether the next or previous control with the WS_TABSTOP style receives the focus.

Return Values

An application should return zero if it processes this message.

Remarks

This message performs additional dialog box management operations beyond those performed by the **SetFocus** function. **WM_NEXTDLGCTL** updates the default push-button border, sets the default control identifier, and automatically selects the text of an edit control (if the target window is an edit control).

Do not use the **SendMessage** function to send a **WM_NEXTDLGCTL** message, if your application will concurrently process other messages that set the focus. Use the **PostMessage** function, instead.

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

Dialog Boxes Overview, Dialog Box Messages, **PostMessage**, **SendMessage**, **SetFocus**

Messages and Message Queues

This overview describes messages and message queues, and how to use them in your Win32-based applications.

About Messages and Message Queues

Unlike MS-DOS-based applications, Win32-based applications are event-driven. They do not make explicit function calls (such as C run-time library calls) to obtain input. Instead, they wait for the system to pass input to them.

The system passes all input for an application to the various windows in the application. Each window has a function, called a *window procedure*, that the system calls whenever it has input for the window. The window procedure processes the input and returns control to the system. For more information about window procedures, see *Window Procedures*.

Win32 Messages

The system passes input to a window procedure in the form of *messages*. Messages are generated by both the system and applications. The system generates a message at each input event—for example, when the user types, moves the mouse, or clicks a control such as a scroll bar. The system also generates messages in response to changes in the system brought about by an application, such as when an application changes the pool of system font resources or resizes one of its windows. An application can generate messages to direct its own windows to perform tasks or communicate with windows in other applications.

The system sends a message to a window procedure with a set of four parameters: a window handle, a message identifier, and two values called message parameters. The *window handle* identifies the window for which the message is intended. The system uses it to determine which window procedure should receive the message.

A *message identifier* is a named constant that identifies the purpose of a message. When a window procedure receives a message, it uses a message identifier to determine how to process the message. For example, the message identifier **WM_PAINT** tells the window procedure that the window's client area has changed and must be repainted.

Message parameters specify data or the location of data used by a window procedure when processing a message. The meaning and value of the message parameters depend on the message. A message parameter can contain an integer, packed bit flags, a pointer to a structure containing additional data, and so on. When a message does not use message parameters, these typically are set to NULL. A window procedure must check the message identifier to determine how to interpret the message parameters.

Message Types

This section describes the two types of messages:

- *system-defined messages*
- *application-defined messages*

System-Defined Messages

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

Each system-defined message has a unique message identifier and a corresponding symbolic constant (defined in the SDK header files) that states the purpose of the message. For example, the **WM_PAINT** constant requests that a window paint its contents.

Symbolic constants specify the category to which system-defined messages belong. The prefix of the constant identifies the type of window that can interpret and process the message. Following are the prefixes and their related message categories:

| Prefix | Message category |
|--------|------------------------------|
| ABM | Application desktop toolbar |
| BM | Button control |
| CB | Combo-box control |
| CBEM | Extended combo-box control |
| CDM | Common dialog box |
| DBT | Device |
| DL | Drag-list box |
| DM | Default push-button control |
| DTM | Date and time picker control |
| EM | Edit control |
| HDM | Header control |
| HKM | Hot-key control |
| IPM | IP address control |
| LB | List-box control |
| LVM | List-view control |
| MCM | Month calendar control |
| PBM | Progress bar |
| PGM | Pager control |
| PSM | Property sheet |
| RB | Rebar control |
| SB | Status bar window |
| SBM | Scroll-bar control |
| STM | Static control |
| TB | Toolbar |

(continued)

(continued)

| Prefix | Message category |
|--------|-------------------|
| TBM | Trackbar |
| TCM | Tab control |
| TTM | Tooltip control |
| TVM | Tree-view control |
| UDM | Up-down control |
| WM | General window |

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

Application-Defined Messages

An application can create messages to be used by its own windows to communicate with windows in other processes. If an application creates its own messages, the window procedure that receives them must interpret the messages and provide appropriate processing.

Message-identifier values are used as follows:

- The system reserves message-identifier values in the range 0x0000 through 0x03FF (the value of **WM_USER** - 1) for system-defined messages. Applications cannot use these values for private messages.
- Values in the range 0x0400 (the value of **WM_USER**) through 0x7FFF are available for message identifiers for private window classes.
- If your application is marked version 4.0, you can use message-identifier values in the range 0x8000 (**WM_APP**) through 0xBFFF for private messages.
- The system returns a message identifier in the range 0xC000 through 0xFFFF when an application calls the **RegisterWindowMessage** function to register a message. The message identifier returned by this function is guaranteed to be unique throughout the system. Use of this function prevents conflicts that can arise if other applications use the same message identifier for different purposes.

Message Routing

The system uses two methods to route messages to a window procedure: posting messages to a first-in, first-out (FIFO) queue called a *message queue*, a system-defined memory object that temporarily stores messages; and sending messages directly to a window procedure.

Messages posted to a message queue are called *queued messages*. They are primarily the result of user input entered through the mouse or keyboard, such as **WM_MOUSEMOVE**, **WM_LBUTTONDOWN**, **WM_KEYDOWN**, and **WM_CHAR** messages. Other queued messages include the timer, paint, and quit messages: **WM_TIMER**, **WM_PAINT**, and **WM_QUIT**. Most other messages, which are sent directly to a window procedure, are called *nonqueued messages*.

- *Queued Messages*
- *Nonqueued Messages*

Queued Messages

The system can display any number of windows at a time. To route mouse and keyboard input to the appropriate window, the system uses message queues.

The system maintains a single system message queue and one thread-specific message queue for each GUI thread. To avoid the overhead of creating a message queue for non-GUI threads, all threads are created initially without a message queue. The system creates a thread-specific message queue only when the thread makes its first call to one of the Win32 User or GDI functions.

Whenever the user moves the mouse, clicks the mouse buttons, or types on the keyboard, the device driver for the mouse or keyboard converts the input into messages and places them in the system message queue. The system removes the messages, one at a time, from the system message queue, examines them to determine the destination window, and then posts them to the message queue of the thread that created the destination window. A thread's message queue receives all mouse and keyboard messages for the windows created by the thread. The thread removes messages from its queue and directs the system to send them to the appropriate window procedure for processing.

With the exception of the **WM_PAINT** message, the system always posts messages at the end of a message queue. This ensures that a window receives its input messages in the proper FIFO sequence. The **WM_PAINT** message, however, is kept in the queue and forwarded to the window procedure only when the queue contains no other messages. Multiple **WM_PAINT** messages for the same window are combined into a single **WM_PAINT** message, consolidating all invalid parts of the client area into a single area. Combining **WM_PAINT** messages reduces the number of times a window must redraw the contents of its client area.

The system posts a message to a thread's message queue by filling an **MSG** structure and, then, copying it to the message queue. Information in **MSG** includes the handle of the window for which the message is intended, the message identifier, the two message parameters, the time the message was posted, and the mouse cursor position. A thread can post a message to its own message queue, or to the queue of another thread, by using the **PostMessage** or **PostThreadMessage** function.

An application can remove a message from its queue by using the **GetMessage** function. To examine a message without removing it from its queue, an application can use the **PeekMessage** function. This function fills **MSG** with information about the message.

After removing a message from its queue, an application can use the **DispatchMessage** function to direct the system to send the message to a window procedure for processing. **DispatchMessage** takes a pointer to **MSG** that was filled by a previous call to the **GetMessage** or **PeekMessage** function. **DispatchMessage** passes the window handle,

the message identifier, and the two message parameters to the window procedure, but it does not pass the time the message was posted or the mouse cursor position. An application can retrieve this information by calling the **GetMessageTime** and **GetMessagePos** functions while processing a message.

A thread can use the **WaitMessage** function to yield control to other threads when it has no messages in its message queue. The function suspends the thread and does not return until a new message is placed in the thread's message queue.

You can call the **SetMessageExtraInfo** function to associate a value with the current thread's message queue. Then, call the **GetMessageExtraInfo** function to get the value associated with the last message retrieved by the **GetMessage** or **PeekMessage** function.

Nonqueued Messages

Nonqueued messages are sent immediately to the destination window procedure, bypassing the system message queue and thread message queue. The system typically sends nonqueued messages to notify a window of events that affect it. For example, when the user activates a new application window, the system sends the window a series of messages, including **WM_ACTIVATE**, **WM_SETFOCUS**, and **WM_SETCURSOR**. These messages notify the window that it has been activated, that keyboard input is being directed to the window, and that the mouse cursor has been moved within the borders of the window. Nonqueued messages can also result when an application calls certain system functions. For example, the system sends the **WM_WINDOWPOSCHANGED** message after an application uses the **SetWindowPos** function to move a window.

Message Handling

An application must remove and process messages posted to the message queues of its threads. A single-threaded application usually uses a message loop in its **WinMain** function to remove and send messages to the appropriate window procedures for processing. Applications with multiple threads can include a message loop in each thread that creates a window. The following sections describe how a message loop works, and explain the role of a window procedure.

Message Loop

A simple message loop consists of one function call to each of these three functions: **GetMessage**, **TranslateMessage**, and **DispatchMessage**.

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

The **GetMessage** function retrieves a message from the queue and copies it to a structure of type **MSG**. It returns a nonzero value, unless it encounters the **WM_QUIT** message, in which case it returns **FALSE** and ends the loop. In a single-threaded application, ending the message loop is often the first step in closing the application. An application can end its own loop by using the **PostQuitMessage** function, typically in response to the **WM_DESTROY** message in the window procedure of the application's main window.

If you specify a window handle as the second parameter of **GetMessage**, only messages for the specified window are retrieved from the queue. **GetMessage** can also filter messages in the queue, retrieving only those that fall within a specified range. For more information about filtering messages, see *Message Filtering*.

A thread's message loop must include **TranslateMessage** if the thread is to receive character input from the keyboard. The system generates virtual-key messages (**WM_KEYDOWN** and **WM_KEYUP**) each time the user presses a key. A virtual-key message contains a virtual-key code that identifies which key was pressed, but not its character value. To retrieve this value, the message loop must contain **TranslateMessage**, which translates the virtual-key message into a character message (**WM_CHAR**) and places it back into the application message queue. The character message, then, can be removed upon a subsequent iteration of the message loop, and dispatched to a window procedure.

The **DispatchMessage** function sends a message to the window procedure associated with the window handle specified in the **MSG** structure. If the window handle is **HWND_TOPMOST**, **DispatchMessage** sends the message to the window procedures of all top-level windows in the system. If the window handle is **NULL**, **DispatchMessage** does nothing with the message.

An application's main thread starts its message loop after initializing the application and creating at least one window. Once started, the message loop continues to retrieve messages from the thread's message queue and dispatch them to the appropriate windows. The message loop ends when the **GetMessage** function removes the **WM_QUIT** message from the message queue.

Only one message loop is needed for a message queue, even if an application contains many windows. **DispatchMessage** always dispatches the message to the proper window; this is because each message in the queue is an **MSG** structure that contains the handle of the window to which the message belongs.

You can modify a message loop in a variety of ways. For example, you can retrieve messages from the queue without dispatching them to a window. This is useful for applications that post messages not specifying a window. You can also direct **GetMessage** to search for specific messages, leaving other messages in the queue. This is useful if you must temporarily bypass the usual FIFO order of the message queue.

An application that uses accelerator keys must be able to translate keyboard messages into command messages. To do this, the application's message loop must include a call

to the **TranslateAccelerator** function. For more information about accelerator keys, see *Keyboard Accelerators*.

If a thread uses a modeless dialog box, the message loop must include the **IsDialogMessage** function, so that the dialog box can receive keyboard input.

Window Procedure

A window procedure is a function that receives and processes all messages sent to the window. Every window class has a window procedure, and every window created with that class uses that same window procedure to respond to messages.

The system sends a message to a window procedure by passing the message data as arguments to the procedure. Then the window procedure performs an appropriate action for the message; it checks the message identifier and, while processing the message, uses the information specified by the message parameters.

A window procedure usually does not ignore a message. If it does not process a message, it must send the message back to the system for default processing. The window procedure does this by calling the **DefWindowProc** function, which performs a default action and returns a message result. Then the window procedure must return this value as its own message result. Most window procedures process just a few messages and pass the others on to the system by calling **DefWindowProc**.

Because a window procedure is shared by all windows belonging to the same class, it can process messages for several different windows. To identify the specific window affected by the message, a window procedure can examine the window handle passed with a message. For more information about window procedures, see *Window Procedures*.

Message Filtering

An application can choose specific messages to retrieve from the message queue (while ignoring other messages) by using the **GetMessage** or **PeekMessage** function to specify a message filter. The filter is a range of message identifiers (specified by a first identifier and last identifier), a window handle, or both. **GetMessage** and **PeekMessage** use a message filter to select which messages to retrieve from the queue. Message filtering is useful if an application must search the message queue for messages that have arrived later in the queue. It is also useful if an application must process input (hardware) messages before processing posted messages.

The **WM_KEYFIRST** and **WM_KEYLAST** constants can be used as filter values to retrieve all keyboard messages; the **WM_MOUSEFIRST** and **WM_MOUSELAST** constants can be used to retrieve all mouse messages.

Any application that filters messages must ensure that a message satisfying the message filter can be posted. For example, if an application filters for a **WM_CHAR** message in a window that does not receive keyboard input, the **GetMessage** function does not return. This effectively “hangs” the application.

Posting and Sending Messages

Any application can post and send messages. Like the system, an application posts a message by copying it to a message queue, and sends a message by passing the message data as arguments to a window procedure. To post messages, an application uses the **PostMessage** function. An application can send a message by calling the **SendMessage**, **BroadcastSystemMessage**, **SendMessageCallback**, **SendMessageTimeout**, **SendNotifyMessage**, or **SendDlgItemMessage** function.

Posting Messages

An application typically posts a message to notify a specific window to perform a task. **PostMessage** creates an **MSG** structure for the message and copies the message to the message queue. The application's message loop eventually retrieves the message and dispatches it to the appropriate window procedure.

An application can post a message without specifying a window. If the application supplies a NULL window handle when calling **PostMessage**, the message is posted to the queue associated with the current thread. Because no window handle is specified, the application must process the message in the message loop. This is one way to create a message that applies to the entire application, instead of to a specific window.

Occasionally, you might want to post a message to all top-level windows in the system. An application can post a message to all top-level windows by calling **PostMessage** and specifying **HWND_TOPMOST** in the *hwnd* parameter.

A common programming error is to assume that the **PostMessage** function always posts a message. This is not true when the message queue is full. An application should check the return value of the **PostMessage** function to determine whether the message has been posted and, if it has not been, to repost it.

Sending Messages

An application typically sends a message to notify a window procedure to perform a task immediately. The **SendMessage** function sends the message to the window procedure corresponding to the given window. The function waits until the window procedure completes processing and, then, returns the message result. Parent and child windows often communicate by sending messages to each other. For example, a parent window that has an edit control as its child window can set the text of the control by sending a message to it. The control can notify the parent window of changes to the text that are carried out by the user by sending messages back to the parent.

The **SendMessageCallback** function also sends a message to the window procedure corresponding to the given window. However, this function returns immediately. After the window procedure processes the message, the system calls the specified callback function. For more information about the callback function, see the **SendAsyncProc** function.

Occasionally, you might want to send a message to all top-level windows in the system. For example, if the application changes the system time, it must notify all top-level

windows about the change by sending a **WM_TIMECHANGE** message. An application can send a message to all top-level windows by calling **SendMessage** and specifying **HWND_TOPMOST** in the *hwnd* parameter. You also can broadcast a message to all applications by calling the **BroadcastSystemMessage** function and specifying **BSM_APPLICATIONS** in the *lpdwRecipients* parameter.

By using the **InSendMessage** or **InSendMessageEx** function, a window procedure can determine whether it is processing a message sent by another thread. This capability is useful when message processing depends on the origin of the message.

Message Deadlocks

A thread that calls the **SendMessage** function to send a message to another thread cannot continue executing until the window procedure that receives the message returns. If the receiving thread yields control while processing the message, the sending thread cannot continue executing, because it is waiting for **SendMessage** to return. If the receiving thread then sends a message to the calling thread while it is blocked, it can cause an application deadlock to occur.

Note that the receiving thread need not yield control explicitly; calling any of the following functions can cause a thread to yield control implicitly:

- DialogBox**
- DialogBoxIndirect**
- DialogBoxIndirectParam**
- DialogBoxParam**
- GetMessage**
- MessageBox**
- PeekMessage**
- SendMessage**

To avoid potential deadlocks in your application, consider using the **SendNotifyMessage** or **SendMessageTimeout** function. Otherwise, a window procedure can determine whether a message it has received was sent by another thread by calling the **InSendMessage** function. Before calling any of the functions in the preceding list while processing a message, the window procedure first should call **InSendMessage**. If this function returns **TRUE**, the window procedure must call the **ReplyMessage** function before any function that causes the thread to yield control.

Broadcasting Messages

Each message consists of a message identifier and two parameters, *wParam* and *lParam*. The message identifier is a unique value that specifies the message purpose. The parameters provide additional information that is message-specific, but the *wParam* parameter is generally a type value that provides more information about the message.

A *message broadcast* is simply the sending of a message to components in the system. To broadcast a message from an application, use the **BroadcastSystemMessage** function, specifying the recipients of the message. Instead of specifying individual

recipients, you must specify one or more types of recipients. These types are applications, installable drivers, network drivers, and system-level device drivers.

BroadcastSystemMessage sends messages to all members of each specified type.

The system typically broadcasts messages in response to changes that take place within system-level device drivers or related components. The driver or related component broadcasts the message to applications and other components to notify them of the change. For example, the component responsible for disk drives broadcasts a message whenever the device driver for the floppy-disk drive detects a change of media, such as when the user inserts a disk in the drive.

The **BroadcastSystemMessage** function sends messages to recipients in this order: system-level device drivers, network drivers, installable drivers, and applications. This means that system-level device drivers, if chosen as recipients, always get the first opportunity to respond to a message. Within a given recipient type, no driver is guaranteed to receive a given message before any other driver. This means that a message intended for a specific driver must have a globally-unique message identifier, so that no other driver unintentionally processes it.

Applications receive messages through the window procedure of their top-level windows. Messages are not sent to child windows. Services can receive messages through a window procedure or their service control handlers.

Note System-level device drivers use a related, system-level function to broadcast system messages.

Query Messages

You can create your own custom messages and use them to coordinate activities between your applications and other components in the system. This is especially useful if you have created your own installable drivers or system-level device drivers. Your custom messages can carry information to and from your driver and the applications that use the driver.

To poll recipients for permission to carry out a given action, use a *query message*. You can generate your own query messages by setting the `BSF_QUERY` value in the `dwFlags` parameter when calling **BroadcastSystemMessage**. Each recipient of the query message must return `TRUE` for the function to send the message to the next recipient. If any recipient returns `BROADCAST_QUERY_DENY`, the broadcast ends immediately and the function returns a zero.

Windows 95/98: You can create *installable drivers* that broadcast and process messages. An installable driver is a dynamic-link library (DLL) that exports a **DriverProc** function. The driver receives messages through its **DriverProc** function, and can broadcast messages using **BroadcastSystemMessage**. Installable drivers are typically used to support multimedia devices, such as sound boards, but can be used for other devices and purposes, too.

Windows 95/98: *Network drivers* are DLLs that provide the underlying support for applications that use the network functions to connect to and browse network resources. System-level device drivers are system-specific executable components that provide direct access to (and management of the hardware devices of) the computer. The details regarding how these components process system messages are beyond the scope of this overview.

Message and Message Queue Reference

Message and Message Queue Functions

BroadcastSystemMessage

The **BroadcastSystemMessage** function sends a message to the specified recipients. The recipients can be applications, installable drivers, network drivers, system-level device drivers, or any combination of these system components.

```

long BroadcastSystemMessage(
    DWORD dwFlags,           // broadcast option
    LPDWORD lpdwRecipients, // recipients
    UINT uiMessage,         // message
    WPARAM wParam,         // first message parameter
    LPARAM lParam           // second message parameter
);

```

Parameters

dwFlags

[in] Specifies the broadcast option. This parameter can be one or more of the following values:

| Value | Meaning |
|-----------------------|--|
| BSF_ALLOWSFWM | Windows 2000: Enables the recipient to set the foreground window while processing the message. |
| BSF_FLUSHDISK | Flushes the disk after each recipient processes the message. |
| BSF_FORCEIFHUNG | Continues to broadcast the message, even if the time-out period elapses or one of the recipients has stopped responding. |
| BSF_IGNORECURRENTTASK | Does not send the message to windows that belong to the current task. This prevents an application from receiving its own message. |

| Value | Meaning |
|------------------------|---|
| BSF_NOHANG | Forces an unresponsive application to time out. If one of the recipients times out, do not continue broadcasting the message. |
| BSF_NOTIMEOUTIFNOTHUNG | Waits for a response to the message, as long as the recipient is not unresponsive. Do not time out. |
| BSF_POSTMESSAGE | Posts the message. Do not use in combination with BSF_QUERY. |
| BSF_QUERY | Sends the message to one recipient at a time, sending to a subsequent recipient only if the current recipient returns TRUE. |
| BSF_SENDNOTIFYMESSAGE | Windows 2000: Sends the message using SendNotifyMessage function. Do not use in combination with BSF_QUERY. |

lpdwRecipients

[in] Pointer to a variable that contains and receives information about the recipients of the message. This parameter can be one or more of the following values:

| Value | Meaning |
|------------------------|--|
| BSM_ALLCOMPONENTS | Broadcast to all system components. |
| BSM_ALLDESKTOPS | Windows NT/2000: Broadcast to all desktops. Requires the SE_TCB_NAME privilege. |
| BSM_APPLICATIONS | Broadcast to applications. |
| BSM_INSTALLABLEDRIVERS | Windows 95/98: Broadcast to installable drivers. |
| BSM_NETDRIVER | Windows 95/98: Broadcast to network drivers. |
| BSM_VXDS | Windows 95/98: Broadcast to all system-level device drivers. |

When the function returns, this variable receives a combination of these values identifying which recipients actually received the message.

If this parameter is NULL, the function broadcasts to all components.

uiMessage

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

If the function succeeds, the return value is a positive value.

If the function is unable to broadcast the message, the return value is -1.

If the *dwFlags* parameter is `BSF_QUERY`, and at least one recipient returned `BROADCAST_QUERY_DENY` to the corresponding message, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If `BSF_QUERY` is not specified, the function sends the specified message to all requested recipients, ignoring values returned by those recipients.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **SendNotifyMessage**

DispatchMessage

The **DispatchMessage** function dispatches a message to a window procedure. It is used typically to dispatch a message retrieved by the **GetMessage** function.

```

LRESULT DispatchMessage(
    CONST MSG *lpmsg // message information
);

```

Parameters

lpmsg

[in] Pointer to an **MSG** structure that contains the message.

Return Values

The return value specifies the value returned by the window procedure. Although its meaning depends on the message being dispatched, the return value generally is ignored.

Remarks

The **MSG** structure must contain valid message values. If the *lpmsg* parameter points to a **WM_TIMER** message, and the *lParam* parameter of the **WM_TIMER** message is not `NULL`, *lParam* points to a function that is called instead of the window procedure.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **MSG**, **PeekMessage**, **TranslateMessage**, **WM_TIMER**

GetInputState

The **GetInputState** function determines whether there are mouse-button or keyboard messages in the calling thread's message queue.

BOOL **GetInputState**(**VOID**);

Parameters

This function has no parameters.

Return Values

If the queue contains one or more new mouse-button or keyboard messages, the return value is nonzero.

If there are no new mouse-button or keyboard messages in the queue, 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 winuser.h; include windows.h.

Library: Use user32.lib.

+ See Also

Messages and Message Queues Overview, Message and Message Queue Functions, **GetQueueStatus**

GetMessage

The **GetMessage** function retrieves a message from the calling thread's message queue and places it in the specified structure. The function dispatches incoming sent messages until a posted message is available.

This function can retrieve both messages associated with a specified window and thread messages posted using the **PostThreadMessage** function. The function retrieves messages that lie within a specified range of message values. **GetMessage** does not retrieve messages for windows that belong to other threads or applications.

```

BOOL GetMessage(
    LPMSG lpMsg,          // message information
    HWND hWnd,           // handle to window
    UINT wMsgFilterMin,  // first message
    UINT wMsgFilterMax   // last message
);

```

Parameters

lpMsg

[in] Pointer to an **MSG** structure that receives message information from the thread's message queue.

hWnd

[in] Handle to the window whose messages are to be retrieved. The following value has a special meaning:

| Value | Meaning |
|-------|---|
| NULL | GetMessage retrieves messages for any window that belongs to the calling thread, and thread messages posted to the calling thread via PostThreadMessage . |

wMsgFilterMin

[in] Specifies the integer value of the lowest message value to be retrieved.

wMsgFilterMax

[in] Specifies the integer value of the highest message value to be retrieved.

Return Values

If the function retrieves a message other than **WM_QUIT**, the return value is nonzero.

If the function retrieves the **WM_QUIT** message, the return value is zero.

If there is an error, the return value is -1 . For example, the function fails if *hWnd* is an invalid window handle or *lpMsg* is an invalid pointer. To get extended error information, call **GetLastError**.

Warning Because the return value can be nonzero, zero, or -1 , avoid code like this:

```
while (GetMessage(&lpMsg, hWnd, 0, 0)) ...
```

The possibility of a -1 return value means that such code can lead to fatal application errors.

Remarks

An application typically uses the return value to determine whether to end the main message loop and exit the program.

The **GetMessage** function only retrieves messages associated with the window identified by the *hWnd* parameter or any of its children, as specified by the **IsChild** function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. If *hWnd* is NULL, **GetMessage** retrieves messages for any window that belongs to the calling thread, and thread messages posted to the calling thread via **PostThreadMessage**. **GetMessage** does not retrieve messages for windows that belong to other threads or for threads other than the calling thread, even if *hWnd* is not NULL. Thread messages, posted by the **PostThreadMessage** function, have a message *hWnd* value of NULL. If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **GetMessage** returns all available messages (that is, no range filtering is performed). Note that **GetMessage** will always retrieve **WM_QUIT** messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

The **WM_KEYFIRST** and **WM_KEYLAST** constants can be used as filter values to retrieve all messages related to keyboard input; the **WM_MOUSEFIRST** and **WM_MOUSELAST** constants can be used to retrieve all mouse messages. If the *wMsgFilterMin* and *wMsgFilterMax* parameters are both zero, the **GetMessage** function returns all available messages (that is, without performing any filtering).

During this call, the system delivers pending messages that were sent to windows owned by the calling thread using the **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, or **SendNotifyMessage** function. The system can also process internal events. Messages are processed in the following order:

1. Sent messages
2. Posted messages
3. Input (hardware) messages and system internal events
4. Sent messages (again)
5. **WM_PAINT** messages
6. **WM_TIMER** messages

To retrieve input messages before posted messages, use the *wMsgFilterMin* and *wMsgFilterMax* parameters.

GetMessage does not remove **WM_PAINT** messages from the queue. The messages remain in the queue until they are processed.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **IsChild**, **MSG**, **PeekMessage**, **PostMessage**, **PostThreadMessage**, **WaitMessage**

GetMessageExtraInfo

The **GetMessageExtraInfo** function gets the extra message information for the current thread. Extra message information is an application-defined or driver-defined value associated with the current thread's message queue. You can use the **SetMessageExtraInfo** function to set a thread's extra message information, which will remain until the next call to **GetMessage** or **PeekMessage**.

LPARAM GetMessageExtraInfo(VOID):

Parameters

This function has no parameters.

Return Values

The return value specifies the extra information. The meaning of the extra information is device-specific.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **PeekMessage**, **SetMessageExtraInfo**

GetMessagePos

The **GetMessagePos** function returns a long value that gives the cursor position in screen coordinates. This position is the point occupied by the cursor when the last message retrieved by the **GetMessage** function occurred.

```
DWORD GetMessagePos(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value specifies the x-coordinate and y-coordinate of the cursor position. The x-coordinate is the low order **int** and the y-coordinate is the high-order **int**.

Remarks

As noted above, the x-coordinate is in the low-order **int** of the return value; the y-coordinate is in the high-order **int** (both represent *signed* values because they can take negative values on systems with multiple monitors). If the return value is assigned to a variable, you can use the **MAKEPOINTS** macro to obtain a **POINTS** structure from the return value. You can also use the **GET_X_LPARAM** or **GET_Y_LPARAM** macro to extract the x-coordinate and y-coordinate.

To determine the current position of the cursor instead of the position when the last message occurred, use the **GetCursorPos** function.

Important Do not use the **LOWORD** or **HIWORD** macros to extract x-coordinate and y-coordinate of the cursor position, because these macros return incorrect results on systems with multiple monitors. Systems with multiple monitor systems can have negative x-coordinates and y-coordinates, and **LOWORD** and **HIWORD** treat the coordinates as unsigned quantities.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetCursorPos**, **GetMessage**, **GetMessageTime**, **HIWORD**, **LOWORD**, **MAKEPOINTS**, **POINTS**

GetMessageTime

The **GetMessageTime** function returns the message time for the last message retrieved by the **GetMessage** function from the current thread's message queue. The time is a long integer that specifies the elapsed time, in milliseconds, from the time the system was started to the time the message was created (that is, placed in the thread's message queue).

```
LONG GetMessageTime(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value specifies the message time.

Remarks

The return value from the **GetMessageTime** function does not necessarily increase between subsequent messages, because the value wraps to zero if the timer count exceeds the maximum value for a long integer.

To calculate time delays between messages, verify that the time of the second message is greater than the time of the first message; then, subtract the time of the first message from the time of the second 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **GetMessagePos**

GetQueueStatus

The **GetQueueStatus** function returns flags that indicate the type of messages found in the calling thread's message queue.

```
DWORD GetQueueStatus(  
    UINT flags // message types  
);
```

Parameters

flags

[in] Specifies the types of messages for which to check. This parameter can be one or more of the following values:

| Value | Meaning |
|-------------------|---|
| QS_ALLEVENTS | An input, WM_TIMER , WM_PAINT , WM_HOTKEY , or posted message is in the queue. |
| QS_ALLINPUT | Any message is in the queue. |
| QS_ALLPOSTMESSAGE | A posted message (other than those listed here) is in the queue. |
| QS_HOTKEY | A WM_HOTKEY message is in the queue. |
| QS_INPUT | An input message is in the queue. |
| QS_KEY | A WM_KEYUP , WM_KEYDOWN , WM_SYSKEYUP , or WM_SYSKEYDOWN message is in the queue. |
| QS_MOUSE | A WM_MOUSEMOVE message or mouse-button message (WM_LBUTTONDOWN , WM_RBUTTONDOWN , and so on) is in the queue. |
| QS_MOUSEBUTTON | A mouse-button message (WM_LBUTTONDOWN , WM_RBUTTONDOWN , and so on) is in the queue. |
| QS_MOUSEMOVE | A WM_MOUSEMOVE message is in the queue. |
| QS_PAINT | A WM_PAINT message is in the queue. |
| QS_POSTMESSAGE | A posted message (other than those listed here) is in the queue. |
| QS_SENDMESSAGE | A message sent by another thread or application is in the queue. |
| QS_TIMER | A WM_TIMER message is in the queue. |

Return Values

The high-order word of the return value indicates the types of messages currently in the queue. The low-order word indicates the types of messages that have been added to the queue and that are still in the queue since the last call to the **GetQueueStatus**, **GetMessage**, or **PeekMessage** function.

Remarks

The presence of a QS_ flag in the return value does not guarantee that a subsequent call to the **GetMessage** or **PeekMessage** function will return a message. **GetMessage** and **PeekMessage** perform some internal filtering that can cause the message to be processed internally. For this reason, the return value from **GetQueueStatus** should be considered only a hint as to whether **GetMessage** or **PeekMessage** should be called.

The `QS_ALLPOSTMESSAGE` and `QS_POSTMESSAGE` flags differ in when they are cleared. `QS_POSTMESSAGE` is cleared when you call **GetMessage** or **PeekMessage**, whether or not you are filtering messages. `QS_ALLPOSTMESSAGE` is cleared when you call **GetMessage** or **PeekMessage** without filtering messages (*wMsgFilterMin* and *wMsgFilterMax* are 0). This can be useful when you call **PeekMessage** multiple times to get messages in different ranges.

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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetInputState**, **GetMessage**, **PeekMessage**

InSendMessage

The **InSendMessage** function determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process) by a call to the **SendMessage** function.

To obtain additional information about how the message was sent, use the **InSendMessageEx** function.

```
BOOL InSendMessage(VOID);
```

Parameters

This function has no parameters.

Return Values

If the window procedure is processing a message sent to it from another thread using the **SendMessage** function, the return value is nonzero.

If the window procedure is not processing a message sent to it from another thread using the **SendMessage** function, 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 winuser.h; include windows.h.

Library: Use user32.lib.

See Also

Messages and Message Queues Overview, Message and Message Queue Functions, **InSendMessageEx**, **SendMessage**

InSendMessageEx

The **InSendMessageEx** function determines whether the current window procedure is processing a message that was sent from another thread (in the same process or a different process).

```
DWORD InSendMessageEx(
    LPVOID lpReserved // not used; must be NULL
);
```

Parameters

lpReserved

Reserved; must be NULL.

Return Values

If the message was not sent, the return value is ISMEX_NOSEND. Otherwise, the return value is one or more of the following values:

| Value | Meaning |
|----------------|--|
| ISMEX_CALLBACK | The message was sent using the SendMessageCallback function. The thread that sent the message is not blocked. |
| ISMEX_NOTIFY | The message was sent using the SendNotifyMessage function. The thread that sent the message is not blocked. |
| ISMEX_REPLIED | The window procedure has processed the message. The thread that sent the message is no longer blocked. |
| ISMEX_SEND | The message was sent using the SendMessage or SendMessageTimeout function. If ISMEX_REPLIED is not set, the thread that sent the message is blocked. |

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.

+ See Also

Messages and Message Queues Overview, Message and Message Queue Functions, **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, **SendNotifyMessage**

PeekMessage

The **PeekMessage** function dispatches incoming sent messages, then checks a thread message queue for a message and places the message (if any) in the specified structure.

```

BOOL PeekMessage(
    LPMSG lpMsg,           // message information
    HWND hWnd,           // handle to window
    UINT wMsgFilterMin,   // first message
    UINT wMsgFilterMax,   // last message
    UINT wRemoveMsg       // removal options
);

```

Parameters

lpMsg

[out] Pointer to an **MSG** structure that receives message information.

hWnd

[in] Handle to the window whose messages are to be examined.

wMsgFilterMin

[in] Specifies the value of the first message in the range of messages to be examined.

wMsgFilterMax

[in] Specifies the value of the last message in the range of messages to be examined.

wRemoveMsg

[in] Specifies how messages are handled. This parameter can be one of the following values:

| Value | Meaning |
|-------------|--|
| PM_NOREMOVE | Messages are not removed from the queue after processing by PeekMessage . |
| PM_REMOVE | Messages are removed from the queue after processing by PeekMessage . |

Optionally, you can combine the value **PM_NOYIELD** with either **PM_NOREMOVE** or **PM_REMOVE**. This flag prevents the system from releasing any thread that is waiting for the caller to go idle (see **WaitForInputIdle**).

By default, all message types are processed. To specify that only certain message should be processed, specify one of more of the following values:

| Value | Meaning |
|-------------------|--|
| PM_QS_INPUT | Windows 98, Windows 2000: Process mouse and keyboard messages. |
| PM_QS_PAINT | Windows 98, Windows 2000: Process paint messages. |
| PM_QS_POSTMESSAGE | Windows 98, Windows 2000: Process all posted messages, including timers and hot keys. |
| PM_QS_SENDMESSAGE | Windows 98, Windows 2000: Process all sent messages. |

Return Values

If a message is available, the return value is nonzero.

If no messages are available, the return value is zero.

Remarks

Unlike the **GetMessage** function, the **PeekMessage** function does not wait for a message to be placed in the queue before returning.

PeekMessage retrieves only messages associated with the window identified by the *hWnd* parameter or any of its children as specified by the **IsChild** function, and within the range of message values given by the *wMsgFilterMin* and *wMsgFilterMax* parameters. If *hWnd* is NULL, **PeekMessage** retrieves messages for any window that belongs to the current thread making the call. (**PeekMessage** does not retrieve messages for windows that belong to other threads.) If *hWnd* is -1, **PeekMessage** returns only messages with a *hWnd* value of NULL, as posted by the **PostThreadMessage** function. If *wMsgFilterMin* and *wMsgFilterMax* are both zero, **PeekMessage** returns all available messages (that is, no range filtering is performed). Note that **GetMessage** will always retrieve **WM_QUIT** messages, no matter which values you specify for *wMsgFilterMin* and *wMsgFilterMax*.

The **WM_KEYFIRST** and **WM_KEYLAST** constants can be used as filter values to retrieve all keyboard messages; the **WM_MOUSEFIRST** and **WM_MOUSELAST** constants can be used to retrieve all mouse messages.

During this call, the system delivers pending messages that were sent to windows owned by the calling thread using the **SendMessage**, **SendMessageCallback**, **SendMessageTimeout**, or **SendNotifyMessage** function. The system can also process internal events. Messages are processed in the following order:

1. Sent messages
2. Posted messages
3. Input (hardware) messages and system internal events
4. Sent messages (again)
5. **WM_PAINT** messages
6. **WM_TIMER** messages

To retrieve input messages before posted messages, use the *wMsgFilterMin* and *wMsgFilterMax* parameters.

The **PeekMessage** function normally does not remove **WM_PAINT** messages from the queue. **WM_PAINT** messages remain in the queue until they are processed. However, if a **WM_PAINT** message has a null update region, **PeekMessage** does remove it from the queue.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **IsChild**, **MSG**, **WaitForInputIdle**, **WaitMessage**

PostMessage

The **PostMessage** function places (posts) a message in the message queue associated with the thread that created the specified window, and then returns without waiting for the thread to process the message.

To post a message in the message queue associate with a thread, use the **PostThreadMessage** function.

```

BOOL PostMessage(
    HWND hWnd,           // handle to destination window
    UINT Msg,           // message
    WPARAM wParam,      // first message parameter
    LPARAM lParam       // second message parameter
);

```

Parameters

hWnd

[in] Handle to the window whose window procedure is to receive the message. The following values have special meanings:

| Value | Meaning |
|----------------|--|
| HWND_BROADCAST | The message is posted to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows. The message is not posted to child windows. |
| NULL | The function behaves like a call to PostThreadMessage with the <i>dwThreadId</i> parameter set to the identifier of the current thread. |

Msg

[in] Specifies the message to be posted.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

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

Messages in a message queue are retrieved by calls to the **GetMessage** or **PeekMessage** function.

Applications that need to communicate using HWND_BROADCAST should use the **RegisterWindowMessage** function to obtain a unique message for interapplication communication.

If you send a message in the range below **WM_USER** to the asynchronous message functions (**PostMessage**, **SendNotifyMessage**, and **SendMessageCallback**), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message, and the sender will free the memory before it is used.

Do not post the **WM_QUIT** message using **PostMessage**; use the **PostQuitMessage** 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.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **PeekMessage**, **PostQuitMessage**, **PostThreadMessage**, **RegisterWindowMessage**, **SendMessageCallback**, **SendNotifyMessage**

PostQuitMessage

The **PostQuitMessage** function indicates to the system that a thread has made a request to terminate (quit). It is used typically in response to a **WM_DESTROY** message.

```
VOID PostQuitMessage(  
    int nExitCode // exit code  
);
```

Parameters

nExitCode

[in] Specifies an application exit code. This value is used as the *wParam* parameter of the **WM_QUIT** message.

Return Values

This function does not return a value.

Remarks

The **PostQuitMessage** function posts a **WM_QUIT** message to the thread's message queue and returns immediately; the function indicates to the system that the thread is requesting to quit at some time in the future.

When the thread retrieves the **WM_QUIT** message from its message queue, it should exit its message loop and return control to the system. The exit value returned to the system must be the *wParam* parameter of the **WM_QUIT** message.

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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **PeekMessage**, **PostMessage**, **WM_DESTROY**, **WM_QUIT**

PostThreadMessage

The **PostThreadMessage** function places (posts) a message in the message queue of the specified thread, and then returns without waiting for the thread to process the message.

```
BOOL PostThreadMessage(  
    DWORD idThread, // thread identifier  
    UINT Msg,       // message  
    WPARAM wParam, // first message parameter  
    LPARAM lParam  // second message parameter  
);
```

Parameters

idThread

[in] Identifier of the thread to which the message will be posted.

The function fails if the specified thread does not have a message queue. The system creates a thread's message queue when the thread makes its first call to one of the Win32 USER or GDI functions. For more information, see the Remarks section.

Msg

[in] Specifies the type of message to be posted.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

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**. **GetLastError** returns **ERROR_INVALID_THREAD_ID**, if *idThread* is not a valid thread identifier, or if the thread specified by *idThread* does not have a message queue.

Remarks

The thread to which the message is posted must have created a message queue, or else the call to **PostThreadMessage** fails. Use one of the following methods to handle this situation:

- Call **PostThreadMessage**. If it fails, call the **Sleep** function, and then call **PostThreadMessage** again. Repeat until **PostThreadMessage** succeeds.
- Create an event object, then create the thread. Use the **WaitForSingleObject** function to wait for the event to be set to the signaled state before calling **PostThreadMessage**. In the thread to which the message will be posted, call **PeekMessage(&msg, NULL, WM_USER, WM_USER, PM_NOREMOVE)** to force the system to create the message queue. Set the event, to indicate that the thread is ready to receive posted messages.

The thread to which the message is posted retrieves the message by calling the **GetMessage** or **PeekMessage** function. The **hwnd** member of the returned **MSG** structure is **NULL**.

Messages sent by **PostThreadMessage** are not associated with a window. Messages that are not associated with a window cannot be dispatched by the **DispatchMessage** function. Therefore, if the recipient thread is in a modal loop (as used by **MessageBox** or **DialogBox**), the messages will be lost. To intercept thread messages while in a modal loop, use a thread-specific hook.

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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetCurrentThreadId**, **GetMessage**, **GetWindowThreadProcessId**, **MSG**, **PeekMessage**, **PostMessage**, **Sleep**, **WaitForSingleObject**

RegisterWindowMessage

The **RegisterWindowMessage** function defines a new window message that is guaranteed to be unique throughout the system. The returned message value can be used when calling the **SendMessage** or **PostMessage** function.

```
UINT RegisterWindowMessage(  
    LPCTSTR lpString // message string  
);
```

Parameters

lpString

[in] Pointer to a null-terminated string that specifies the message to be registered.

Return Values

If the message is registered successfully, the return value is a message identifier in the range 0xC000 through 0xFFFF.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

The **RegisterWindowMessage** function is used typically to register messages for communicating between two cooperating applications.

If two different applications register the same message string, the applications return the same message value. The message remains registered until the session ends.

Only use **RegisterWindowMessage** when more than one application must process the same message. For sending private messages within a window class, an application can use any integer in the range **WM_USER** through 0x7FFF. (Messages in this range are private to a window class, not to an application. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** may use values in this range.)

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

Messages and Message Queues Overview, Message and Message Queue Functions, **PostMessage**, **SendMessage**

ReplyMessage

The **ReplyMessage** function is used to reply to a message sent through the **SendMessage** function without returning control to the function that called **SendMessage**.

```
BOOL ReplyMessage(  
    LRESULT lResult // message-specific reply  
);
```

Parameters

lResult

[in] Specifies the result of the message processing. The possible values are based on the message sent.

Return Values

If the calling thread was processing a message sent from another thread or process, the return value is nonzero.

If the calling thread was not processing a message sent from another thread or process, the return value is zero.

Remarks

By calling this function, the window procedure that receives the message allows the thread that called **SendMessage** to continue to run as though the thread receiving the message had returned control. The thread that calls the **ReplyMessage** function also continues to run.

If the message was not sent through **SendMessage**, or if the message was sent by the same thread, **ReplyMessage** has no effect.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **InSendMessage**, **SendMessage**

SendAsyncProc

The **SendAsyncProc** function is an application-defined callback function used with the **SendMessageCallback** function. The system passes the message to the callback function after passing the message to the destination window procedure. The **SENDASYNCPROC** type defines a pointer to this callback function. **SendAsyncProc** is a placeholder for the application-defined function name.

```
VOID CALLBACK SendAsyncProc(  
    HWND hwnd, // handle to destination window
```

```
UINT uMsg,           // message
ULONG_PTR dwData,  // application-defined value
LRESULT lResult    // result of message processing
);
```

Parameters

hwnd

[in] Handle to the window whose window procedure received the message.

If the **SendMessageCallback** function was called with its *hwnd* parameter set to **HWND_BROADCAST**, the system calls the **SendAsyncProc** function once for each top-level window.

uMsg

[in] Specifies the message.

dwData

[in] Specifies an application-defined value sent from the **SendMessageCallback** function.

lResult

[in] Specifies the result of the message processing. This value depends on the message.

Return Values

This callback function does not return a value.

Remarks

You install a **SendAsyncProc** application-defined callback function by passing a **SENDASYNCPROC** pointer to the **SendMessageCallback** function.

The callback function is called only when the thread that called **SendMessageCallback** calls **GetMessage**, **PeekMessage**, or **WaitMessage**.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **PeekMessage**, **SendMessageCallback**, **WaitMessage**

SendMessage

The **SendMessage** function sends the specified message to a window or windows. It calls the window procedure for the specified window, and does not return until the window procedure has processed the message.

To send a message and return immediately, use the **SendMessageCallback** or **SendNotifyMessage** function. To post a message to a thread's message queue and return immediately, use the **PostMessage** or **PostThreadMessage** function.

```
LRESULT SendMessage(  
    HWND hWnd,        // handle to destination window  
    UINT Msg,         // message  
    WPARAM wParam,    // first message parameter  
    LPARAM lParam     // second message parameter  
);
```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is `HWND_BROADCAST`, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; however, the message is not sent to child windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing; it depends on the message sent.

Remarks

Applications that need to communicate using `HWND_BROADCAST` should use the **RegisterWindowMessage** function to obtain a unique message for interapplication communication.

If the specified window was created by the calling thread, the window procedure is called immediately as a subroutine. If the specified window was created by a different thread, the system switches to that thread and calls the appropriate window procedure. Messages sent between threads are processed only when the receiving thread executes message retrieval code. The sending thread is blocked until the receiving thread processes the message.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **InSendMessage**, **PostMessage**, **PostThreadMessage**, **RegisterWindowMessage**, **SendDlgItemMessage**, **SendMessageCallback**, **SendNotifyMessage**

SendMessageCallback

The **SendMessageCallback** function sends the specified message to a window or windows. It calls the window procedure for the specified window, and returns immediately. After the window procedure processes the message, the system calls the specified callback function, passing the result of the message processing and an application-defined value to the callback function.

```

BOOL SendMessageCallback(
    HWND hWnd,           // handle to window
    UINT Msg,           // message
    WPARAM wParam,      // first message parameter
    LPARAM lParam,      // second message parameter
    SENDASYNCPROC lpCallback, // callback function
    ULONG_PTR dwData    // application-defined value
);

```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is `HWND_BROADCAST`, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; however, the message is not sent to child windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

lpCallback

[in] Pointer to a callback function that the system calls after the window procedure processes the message. For more information, see **SendAsyncProc**.

If *hWnd* is `HWND_BROADCAST`, the system calls the **SendAsyncProc** callback function once for each top-level window.

dwData

[in] Specifies an application-defined value to be sent to the callback function pointed to by the *lpCallback* parameter.

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

If you send a message in the range below **WM_USER** to the asynchronous message functions (**PostMessage**, **SendNotifyMessage**, and **SendMessageCallback**), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message, and the sender will free the memory before it is used.

Applications that need to communicate using `HWND_BROADCAST` should use the **RegisterWindowMessage** function to obtain a unique message for interapplication communication.

The callback function is called only when the thread that called **SendMessageCallback** also calls **GetMessage**, **PeekMessage**, or **WaitMessage**.

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

Messages and Message Queues Overview, Message and Message Queue Functions, **PostMessage**, **RegisterWindowMessage**, **SendAsyncProc**, **SendMessageCallback**, **SendNotifyMessage**

SendMessageTimeout

The **SendMessageTimeout** function sends the specified message to a window or windows. The function calls the window procedure for the specified window and, if the specified window belongs to a different thread, does not return until the window procedure has processed the message or the specified time-out period has elapsed. If the window receiving the message belongs to the same queue as the current thread, the window procedure is called directly—the time-out value is ignored.

```

LRESULT SendMessageTimeout(
    HWND hWnd,           // handle to window
    UINT Msg,           // message
    WPARAM wParam,      // first message parameter
    LPARAM lParam,      // second message parameter
    UINT fuFlags,        // send options
    UINT uTimeout,      // time-out duration
    PDWORD_PTR lpdwResult // return value for synchronous call
);

```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is `HWND_BROADCAST`, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

fuFlags

[in] Specifies how to send the message. This parameter can be one or more of the following values:

| Value | Meaning |
|-------------------------------|--|
| <code>SMTO_ABORTIFHUNG</code> | Returns without waiting for the time-out period to elapse if the receiving process appears to be in a “hung” (unresponsive) state. |
| <code>SMTO_BLOCK</code> | Prevents the calling thread from processing any other requests until the function returns. |
| <code>SMTO_NORMAL</code> | The calling thread is not prevented from processing other requests while waiting for the function to return. |

(continued)

(continued)

| Value | Meaning |
|-------------------------|--|
| SMTO_NOTIMEOUTIFNOTHUNG | Windows 2000: Does not return when the time-out period elapses if the receiving thread is not “hung”. |

uTimeout

[in] Specifies the duration, in milliseconds, of the time-out period. If the message is a broadcast message, each window can use the full time-out period. For example, if you specify a 5-second time-out period and there are three top-level windows that fail to process the message, you could have up to a 15-second delay.

lpdwResult

[in] Receives the result of the message processing. This value depends on the message sent.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails or times out, the return value is zero. To get extended error information, call **GetLastError**. If **GetLastError** returns zero, then the function timed out.

SendMessageTimeout does not provide information about individual windows timing out if `HWND_BROADCAST` is 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Messages and Message Queues Overview, Message and Message Queue Functions, **InSendMessage**, **PostMessage**, **SendDlgItemMessage**, **SendMessage**, **SendMessageCallback**, **SendNotifyMessage**

SendNotifyMessage

The **SendNotifyMessage** function sends the specified message to a window. If the window was created by the calling thread, **SendNotifyMessage** calls the window procedure for the window, and does not return until the window procedure has processed the message. If the window was created by a different thread, **SendNotifyMessage** passes the message to the window procedure and returns immediately; it does not wait for the window procedure to finish processing the message.

```
BOOL SendNotifyMessage(  
    HWND hWnd,        // handle to window  
    UINT Msg,         // message  
    WPARAM wParam,    // first message parameter  
    LPARAM lParam     // second message parameter  
);
```

Parameters

hWnd

[in] Handle to the window whose window procedure will receive the message. If this parameter is `HWND_BROADCAST`, the message is sent to all top-level windows in the system, including disabled or invisible unowned windows, overlapped windows, and pop-up windows; however, the message is not sent to child windows.

Msg

[in] Specifies the message to be sent.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

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

If you send a message in the range below **WM_USER** to the asynchronous message functions (**PostMessage**, **SendNotifyMessage**, and **SendMessageCallback**), its message parameters cannot include pointers. Otherwise, the operation will fail. The functions will return before the receiving thread has had a chance to process the message, and the sender will free the memory before it is used.

Applications that need to communicate using `HWND_BROADCAST` should use the **RegisterWindowMessage** function to obtain a unique message for interapplication communication.

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

Messages and Message Queues Overview, Message and Message Queue Functions, **PostMessage**, **PostThreadMessage**, **RegisterWindowMessage**, **SendMessage**, **SendMessageCallback**, **SendNotifyMessage**

SendMessageExtraInfo

The **SendMessageExtraInfo** function sets the extra message information for the current thread. Extra message information is an application-defined or driver-defined value associated with the current thread's message queue. An application can use the **GetMessageExtraInfo** function to retrieve a thread's extra message information.

```
LPARAM SendMessageExtraInfo(  
    LPARAM lParam // application-defined value  
);
```

Parameters

lParam

[in] Specifies the value to associate with the current thread.

Return Values

The return value is the previous value associated with the current thread.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessageExtraInfo**

TranslateMessage

The **TranslateMessage** function translates virtual-key messages into character messages. The character messages are posted to the calling thread's message queue, to be read the next time the thread calls the **GetMessage** or **PeekMessage** function.

```
BOOL TranslateMessage(  
    CONST MSG *lpMsg // message information  
);
```

Parameters

lpMsg

[in] Pointer to an **MSG** structure that contains message information retrieved from the calling thread's message queue by using the **GetMessage** or **PeekMessage** function.

Return Values

If the message is translated (that is, a character message is posted to the thread's message queue), the return value is nonzero.

If the message is **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, or **WM_SYSKEYUP**, the return value is nonzero, regardless of the translation.

If the message is not translated (that is, a character message is not posted to the thread's message queue), the return value is zero.

Remarks

The **TranslateMessage** function does not modify the message pointed to by the *lpMsg* parameter.

WM_KEYDOWN and **WM_KEYUP** combinations produce a **WM_CHAR** or **WM_DEADCHAR** message. **WM_SYSKEYDOWN** and **WM_SYSKEYUP** combinations produce a **WM_SYSCHAR** or **WM_SYSDEADCHAR** message.

TranslateMessage produces **WM_CHAR** messages only for keys that are mapped to ASCII characters by the keyboard driver.

If applications process virtual-key messages for some other purpose, they should not call **TranslateMessage**. For instance, an application should not call **TranslateMessage** if the **TranslateAccelerator** function returns a nonzero value.

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

Messages and Message Queues Overview, *Message and Message Queue Functions*, **GetMessage**, **PeekMessage**, **TranslateAccelerator**, **WM_CHAR**, **WM_DEADCHAR**, **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSCHAR**, **WM_SYSDEADCHAR**, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**

WaitMessage

The **WaitMessage** function yields control to other threads when a thread has no other messages in its message queue. The **WaitMessage** function suspends the thread and does not return until a new message is placed in the thread's message queue.

BOOL WaitMessage(VOID);

Parameters

This function has no parameters.

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

Note that **WaitMessage** does not return if there is unread input in the message queue after the thread has called a function to check the queue. This is because functions such as **GetMessage**, **GetQueueStatus**, **MsgWaitForMultipleObjects**, **MsgWaitForMultipleObjectsEx**, **PeekMessage**, and **WaitMessage**, check the queue and then change the state information for the queue, so that the input is no longer considered new. A subsequent call to **WaitMessage** will not return until new input of the specified type arrives. The existing unread input (received prior to the last time the thread checked the queue) is ignored.

! 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

Messages and Message Queues Overview, Message and Message Queue Functions, **GetMessage**, **PeekMessage**

Message and Message Queue Structures

MSG

The **MSG** structure contains message information from a thread's message queue.

```
typedef struct tagMSG {
    HWND    hwnd;
    UINT    message;
    WPARAM wParam;
    LPARAM lParam;
    DWORD   time;
    POINT   pt;
} MSG, *PMSG;
```

Members

hwnd

Handle to the window whose window procedure receives the message.

message

Specifies the message identifier.

wParam

Specifies additional information about the message. The exact meaning depends on the value of the **message** member.

lParam

Specifies additional information about the message. The exact meaning depends on the value of the **message** member.

time

Specifies the time when the message was posted.

pt

Specifies the cursor position, in screen coordinates, at which the message was posted.

! 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

Messages and Message Queues Overview, Message and Message Queue Structures, GetMessage, PeekMessage

Message and Message Queue Messages

WM_APP

The **WM_APP** constant is used by applications to help define private messages, usually of the form **WM_APP+X**, where X is an integer value.

```
#define WM_APP 0x8000
```

Remarks

The **WM_APP** constant is used to distinguish between message values that are reserved for use by the system and values that can be used by an application to send messages within a private window class. The following are the ranges of message numbers available:

| Range | Meaning |
|-------------------------------|--|
| 0 through WM_USER –1 | Messages reserved for use by the system |
| WM_USER through 0x7FFF | Integer messages for use by private window classes |
| WM_APP through 0xBFFF | Messages available for use by applications |
| 0xC000 through 0xFFFF | String messages for use by applications |
| Greater than 0xFFFF | Reserved by the system for future use |

Message numbers in the first range (0 through **WM_USER** –1) are defined by the system. Values in this range that are not explicitly defined are reserved for future use by the system.

Message numbers in the second range (**WM_USER** through 0x7FFF) can be defined and used by an application to send messages within a private window class. These values cannot be used to define messages that are meaningful throughout an application, because some predefined window classes already define values in this range. For example, predefined control classes such as **BUTTON**, **EDIT**, **LISTBOX**, and **COMBOBOX** can use these values. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are available for application to use as private messages. Message in this range do not conflict with system messages.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the **RegisterWindowMessage** function to retrieve a message number for a string. All applications that register the same string can use the associated message number for exchanging messages. The actual message number, however, is not a constant, and cannot be assumed to be the same between different sessions.

Message numbers in the fifth range (greater than 0xFFFF) are reserved for future use by the system.

! Requirements

Windows NT/2000: Requires Windows NT 3.51 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

Messages and Message Queues Overview, Message and Message Queue Messages, RegisterWindowMessage, WM_USER

WM_USER

The **WM_USER** constant is used by applications to help define private messages, usually of the form **WM_USER+X**, where X is an integer value.

```
#define WM_USER 0x0400
```

Remarks

The following are the ranges of message numbers:

| Range | Meaning |
|-------------------------------|--|
| 0 through WM_USER -1 | Messages reserved for use by the system |
| WM_USER through 0x7FFF | Integer messages for use by private window classes |
| WM_APP through 0xBFFF | Messages available for use by applications |
| 0xC000 through 0xFFFF | String messages for use by applications |
| Greater than 0xFFFF | Reserved by the system for future use |

Message numbers in the first range (0 through **WM_USER** -1) are defined by the system. Values in this range that are not explicitly defined are reserved for future use by the system.

Message numbers in the second range (**WM_USER** through 0x7FFF) can be defined and used by an application to send messages within a private window class. These values cannot be used to define messages that are meaningful throughout an application, because some predefined window classes already define values in this range. For example, predefined control classes such as BUTTON, EDIT, LISTBOX, and COMBOBOX can use these values. Messages in this range should not be sent to other applications unless the applications have been designed to exchange messages and to attach the same meaning to the message numbers.

Message numbers in the third range (0x8000 through 0xBFFF) are available for application to use as private messages. Message in this range do not conflict with system messages.

Message numbers in the fourth range (0xC000 through 0xFFFF) are defined at run time when an application calls the **RegisterWindowMessage** function to retrieve a message number for a string. All applications that register the same string can use the associated message number for exchanging messages. The actual message number, however, is not a constant, and cannot be assumed to be the same between different sessions.

Message numbers in the fifth range (greater than 0xFFFF) are reserved for future use 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

Messages and Message Queues Overview, Message and Message Queue Messages, **RegisterWindowMessage**, **WM_APP**

Multiple Document Interface

The multiple document interface (MDI) is a specification that defines a user interface for applications that enable the user to work with more than one document at the same time.

Note MDI is an application-oriented model. Many new and intermediate users find it difficult to learn to use MDI applications. Therefore, many applications are switching to a document-oriented model. Therefore, you might want to consider other models for your user interface. However, you can use MDI for applications which do not fit easily into an existing model until a more suitable model is introduced.

About the Multiple Document Interface

Each document in an MDI application is displayed in a separate child window within the client area of the application's main window. Typical MDI applications include word-processing applications that allow the user to work with multiple text documents, and spreadsheet applications that allow the user to work with multiple charts and spreadsheets.

Frame, Client, and Child Windows

An MDI application has three kinds of windows: a frame window, an MDI client window, and a number of child windows. The *frame window* is like the main window of the application: it has a sizing border, a title bar, a window menu, a minimize button, and a maximize button. The application must register a window class for the frame window and provide a window procedure to support it.

An MDI application does not display output in the client area of the frame window. Instead, it displays the MDI client window. An *MDI client window* is a special type of child window belonging to the preregistered window class MDICLIENT. The client window is a child of the frame window; it serves as the background for child windows. It also provides support for creating and manipulating child windows. For example, an MDI application can create, activate, or maximize child windows by sending messages to the MDI client window.

When the user opens or creates a document, the client window creates a child window for the document. The client window is the parent window of all MDI child windows in the application. Each child window has a sizing border, a title bar, a window menu, a minimize button, and a maximize button. Because a child window is clipped, it is confined to the client window and cannot appear outside it.

An MDI application can support more than one kind of document. For example, a typical spreadsheet application enables the user to work with both charts and spreadsheets. For each type of document that it supports, an MDI application must register a child window class and provide a window procedure to support the windows belonging to that class. For more information about window classes, see *Window Classes*. For more information about window procedures, see *Window Procedures*.

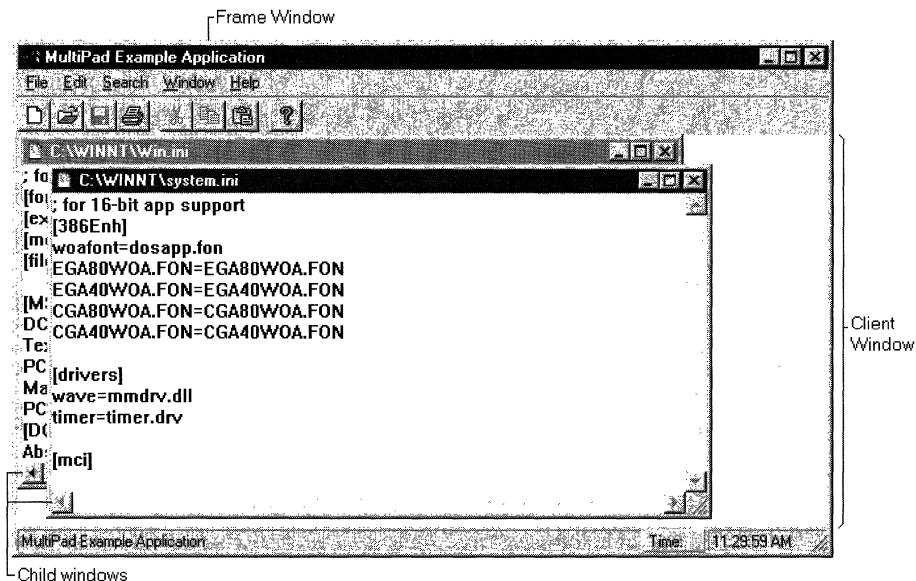


Figure 9-1: A typical MDI application.

Child Window Creation

To create a child window, an MDI application either calls the **CreateMDIWindow** function or sends the **WM_MDICREATE** message to the MDI client window. A more efficient way to create an MDI child window is to call the **CreateWindowEx** function, specifying the **WS_EX_MDICHILD** extended style. A thread in an MDI application can use **CreateMDIWindow** or **CreateWindowEx** to create a child window in a different thread. The **WM_MDICREATE** message is used only in the context of the same thread.

To destroy a child window, an MDI application sends a **WM_MDIDESTROY** message to the MDI client window.

Child Window Activation

Any number of child windows can appear in the client window at any one time, but only one can be active. The active child window is positioned in front of all other child windows, and its border is highlighted.

The user can activate an inactive child window by clicking it. An MDI application activates a child window by sending a **WM_MDIACTIVATE** message to the MDI client window. As the client window processes this message, it sends a **WM_MDIACTIVATE** message to the window procedure of the child window to be activated and to the window procedure of the child window being deactivated.

To prevent a child window from activating, handle the **WM_NCACTIVATE** message to the child window by returning **FALSE**.

The system keeps track of each child window's position in the stack of overlapping windows. This stacking is known as the *Z order*. The user can activate the next child window in the Z order by clicking **Next** from the window menu in the active window. An application activates the next (or previous) child window in the Z order by sending a **WM_MDINEXT** message to the client window.

To retrieve the handle to the active child window, the MDI application sends a **WM_MDIGETACTIVE** message to the client window.

Multiple Document Menus

The frame window of an MDI application should include a menu bar with a window menu. The window menu should include items that either arrange the child windows within the client window or close all child windows. The window menu of a typical MDI application might include the items in the following table:

| Menu item | Purpose |
|----------------|--|
| Tile | Arranges child windows in a tile format, so that each appears in its entirety in the client window. |
| Cascade | Arranges child windows in a cascade format. The child windows overlap one another, but the title bar of each is visible. |

| Menu item | Purpose |
|----------------------|--|
| Arrange Icons | Arranges the icons of minimized child windows along the bottom of the client window. |
| Close All | Closes all child windows. |

Whenever a child window is created, the system automatically appends a new menu item to the window menu. The text of the menu item is the same as the text on the menu bar of the new child window. By clicking the menu item, the user can activate the corresponding child window. When a child window is destroyed, the system automatically removes the corresponding menu item from the window menu.

The system can add up to 10 menu items to the window menu. When the tenth child window is created, the system adds the **More Windows** item to the window menu. Clicking this item displays the **Select Window** dialog box. The dialog box contains a list box with the titles of all MDI child windows currently available. The user can activate a child window by clicking its title in the list box.

If your MDI application supports several types of child windows, tailor the menu bar to reflect the operations associated with the active window. To do this, provide separate menu resources for each type of child window the application supports. When a new type of child window is activated, the application should send a **WM_MDISETMENU** message to the client window, passing to it the handle to the corresponding menu.

When no child window exists, the menu bar should contain only items used to create or open a document.

When the user is moving through an MDI application's menus by using cursor keys, the keys behave differently than when the user is moving through a typical application's menus. In an MDI application, control passes from the application's window menu to the window menu of the active child window, and then to the first item on the menu bar.

Multiple Document Accelerators

To receive and process accelerator keys for its child windows, an MDI application must include the **TranslateMDISysAccel** function in its message loop. The loop must call **TranslateMDISysAccel** before calling the **TranslateAccelerator** or **DispatchMessage** function.

Accelerator keys on the window menu for an MDI child window are different from those for a non-MDI child window. In an MDI child window, the ALT+ – (minus) key combination opens the window menu, the CTRL+F4 key combination closes the active child window, and the CTRL+F6 key combination activates the next child window.

Child Window Size and Arrangement

An MDI application controls the size and position of its child windows by sending messages to the MDI client window. To maximize the active child window, the application sends the **WM_MDIMAXIMIZE** message to the client window. When a child window is maximized, its client area completely fills the MDI client window. In addition, the system automatically hides the child window's title bar, and adds the child window's

window-menu icon and **Restore** button to the MDI application's menu bar. The application can restore the client window to its original (premaximized) size and position by sending the client window a **WM_MDIRESTORE** message.

An MDI application can arrange its child windows in either a cascade or tile format. When the child windows are cascaded, the windows appear in a stack. The window on the bottom of the stack occupies the upper-left corner of the screen, and the remaining windows are offset vertically and horizontally, so that the left border and title bar of each child window is visible. To arrange child windows in the cascade format, an MDI application sends the **WM_MDICASCADE** message. Typically, the application sends this message when the user clicks **Cascade** on the window menu.

When the child windows are tiled, the system displays each child window in its entirety, overlapping none of the windows. All of the windows are sized, as necessary, to fit within the client window. To arrange child windows in the tile format, an MDI application sends a **WM_MDITILE** message to the client window. Typically, the application sends this message when the user clicks **Tile** on the window menu.

An MDI application should provide a different icon for each type of child window it supports. The application specifies an icon when registering the child window class. The system automatically displays a child window's icon in the lower portion of the client window when the child window is minimized. An MDI application directs the system to arrange child window icons by sending a **WM_MDIICONARRANGE** message to the client window. Typically, the application sends this message when the user clicks **Arrange Icons** on the window menu.

Icon Title Windows

Because MDI child windows can be minimized, an MDI application must avoid manipulating icon title windows as if they were normal MDI child windows. Icon title windows appear when the application enumerates child windows of the MDI client window. Icon title windows differ from other child windows, however, in that they are owned by an MDI child window.

To determine whether a child window is an icon title window, use the **GetWindow** function with the **GW_OWNER** index. Non-title windows return **NULL**. Note that this test is insufficient for top-level windows, because menus and dialog boxes are owned windows.

Child Window Data

Because the number of child windows varies depending on how many documents the user opens, an MDI application must be able to associate data (for example, the name of the current file) with each child window. There are two ways to do this:

- Store child window data in the window structure.
- Use window properties.

Window Structure

When an MDI application registers a window class, it can reserve extra space in the window structure for application data specific to this particular class of windows. To store and retrieve data in this extra space, the application uses the **GetWindowWord**, **SetWindowWord**, **GetWindowLong**, and **SetWindowLong** functions.

To maintain a large amount of data for a child window, an application can allocate memory for a data structure, and then store the handle to the memory containing the structure in the extra space associated with the child window.

Window Properties

An MDI application also can store per-document data by using window properties. *Per-document data* is data specific to the type of document contained in a particular child window. Properties are different from extra space in the window structure in that you do not need to allocate extra space when registering the window class. A window can have any number of properties. Also, where offsets are used to access the extra space in window structures, properties are referred to by string names. For more information about window properties, see *Window Properties*.

Multiple Document Interface Reference

Multiple Document Interface Functions

CreateMDIWindow

The **CreateMDIWindow** function creates a multiple document interface (MDI) child window.

```
HWND CreateMDIWindow(  
    LPCTSTR lpClassName, // registered child class name  
    LPCTSTR lpWindowName, // window name  
    DWORD dwStyle, // window style  
    int X, // horizontal position of window  
    int Y, // vertical position of window  
    int nWidth, // window width  
    int nHeight, // window height  
    HWND hWndParent, // handle to parent window  
    HINSTANCE hInstance, // handle to application instance  
    LPARAM lParam // application-defined value  
);
```

Parameters

lpClassName

[in] Pointer to a null-terminated string specifying the window class of the MDI child window. The class name must have been registered by a call to the **RegisterClassEx** function.

lpWindowName

[in] Pointer to a null-terminated string that represents the window name. The system displays the name in the title bar of the child window.

dwStyle

[in] Specifies the style of the MDI child window. If the MDI client window is created with the MDIS_ALLCHILDSTYLES window style, this parameter can be any combination of the window styles listed in the description of the **CreateWindow** function. Otherwise, this parameter can be one or more of the following values:

| Value | Meaning |
|-------------|---|
| WS_HSCROLL | Creates an MDI child window that has a horizontal scroll bar. |
| WS_MAXIMIZE | Creates an MDI child window that is initially maximized. |
| WS_MINIMIZE | Creates an MDI child window that is initially minimized. |
| WS_VSCROLL | Creates an MDI child window that has a vertical scroll bar. |

X

[in] Specifies the initial horizontal position, in client coordinates, of the MDI child window. If this parameter is CW_USEDEFAULT, the MDI child window is assigned the default horizontal position.

Y

[in] Specifies the initial vertical position, in client coordinates, of the MDI child window. If this parameter is CW_USEDEFAULT, the MDI child window is assigned the default vertical position.

nWidth

[in] Specifies the initial width, in device units, of the MDI child window. If this parameter is CW_USEDEFAULT, the MDI child window is assigned the default width.

nHeight

[in] Specifies the initial height, in device units, of the MDI child window. If this parameter is set to CW_USEDEFAULT, the MDI child window is assigned the default height.

hWndParent

[in] Handle to the MDI client window that will be the parent of the new MDI child window.

hInstance

[in] Handle to the instance of the application creating the MDI child window.

lParam

[in] Specifies an application-defined value.

Return Values

If the function succeeds, the return value is the handle to the created window.

If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

Using the **CreateMDIWindow** function is similar to sending the **WM_MDICREATE** message to an MDI client window, except that the function can create an MDI child window in a different thread, while the message cannot.

Windows 95: The system can support a maximum of 16,364 window 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 *winuser.h*; include *windows.h*.

Library: Use *user32.lib*.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Multiple Document Interface Overview, Multiple Document Interface Functions, **CreateWindow**, **RegisterClassEx**, **WM_MDICREATE**

DefFrameProc

The **DefFrameProc** function provides default processing for any window messages that the window procedure of a multiple document interface (MDI) frame window does not process. All window messages that are not explicitly processed by the window procedure must be passed to the **DefFrameProc** function, not to the **DefWindowProc** function.

```
LRESULT DefFrameProc(  
    HWND hWnd,           // handle to MDI frame window  
    HWND hWndMDIClient, // handle to MDI client window  
    UINT uMsg,          // message  
    WPARAM wParam,     // first message parameter  
    LPARAM lParam       // second message parameter  
);
```

Parameters

hWnd

[in] Handle to the MDI frame window.

hWndMDIClient

[in] Handle to the MDI client window.

uMsg

[in] Specifies the message to be processed.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message. If the *hWndMDIClient* parameter is NULL, the return value is the same as for the **DefWindowProc** function.

Remarks

When an application's window procedure does not handle a message, it typically passes the message to the **DefWindowProc** function to process the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions, instead of **DefWindowProc**, to provide default message processing. All messages that an application would usually pass to **DefWindowProc** (such as nonclient messages and the **WM_SETTEXT** message) should be passed to **DefFrameProc**, instead. The **DefFrameProc** function also handles the following messages:

| Message | Response |
|--------------------|---|
| WM_COMMAND | Activates the MDI child window that the user chooses. This message is sent when the user chooses an MDI child window from the window menu of the MDI frame window. The window identifier accompanying this message identifies the MDI child window to be activated. |
| WM_MENUCHAR | Opens the window menu of the active MDI child window when the user presses the ALT+ – (minus) key combination. |
| WM_SETFOCUS | Passes the keyboard focus to the MDI client window, which, in turn, passes it to the active MDI child window. |
| WM_SIZE | Resizes the MDI client window to fit in the new frame window's client area. If the frame window procedure sizes the MDI client window to a different size, it should not pass the message to the DefWindowProc 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 winuser.h; include windows.h.

Library: Use user32.lib.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Multiple Document Interface Overview, Multiple Document Interface Functions, **DefMDIChildProc**, **DefWindowProc**, **WM_SETTEXT**

DefMDIChildProc

The **DefMDIChildProc** function provides default processing for any window message that the window procedure of a multiple document interface (MDI) child window does not process. A window message not processed by the window procedure must be passed to the **DefMDIChildProc** function, not to the **DefWindowProc** function.

```
LRESULT DefMDIChildProc(  
    HWND hWnd, // handle to MDI child window  
    UINT uMsg, // message  
    WPARAM wParam, // first message parameter  
    LPARAM lParam // second message parameter  
);
```

Parameters

hWnd

[in] Handle to the MDI child window.

uMsg

[in] Specifies the message to be processed.

wParam

[in] Specifies additional message-specific information.

lParam

[in] Specifies additional message-specific information.

Return Values

The return value specifies the result of the message processing and depends on the message.

Remarks

The **DefMDIChildProc** function assumes that the parent window of the MDI child window identified by the *hWnd* parameter was created with the MDIClient class.

When an application's window procedure does not handle a message, it typically passes the message to the **DefWindowProc** function to process the message. MDI applications use the **DefFrameProc** and **DefMDIChildProc** functions, instead of **DefWindowProc**, to provide default message processing. All messages that an application would usually

pass to **DefWindowProc** (such as nonclient messages and the **WM_SETTEXT** message) should be passed to **DefMDIChildProc**, instead. In addition, **DefMDIChildProc** also handles the following messages:

| Message | Response |
|-------------------------|---|
| WM_CHILDACTIVATE | Performs activation processing when MDI child windows are sized, moved, or displayed. This message must be passed. |
| WM_GETMINMAXINFO | Calculates the size of a maximized MDI child window, based on the current size of the MDI client window. |
| WM_MENUCHAR | Passes the message to the MDI frame window. |
| WM_MOVE | Recalculates MDI client scroll bars, if they are present. |
| WM_SETFOCUS | Activates the child window, if it is not the active MDI child window. |
| WM_SIZE | Performs operations necessary for changing the size of a window, especially for maximizing or restoring an MDI child window. Failing to pass this message to the DefMDIChildProc function produces highly undesirable results. |
| WM_SYSCOMMAND | Handles window menu commands: SC_NEXTWINDOW , SC_PREVWINDOW , SC_MOVE , SC_SIZE , and SC_MAXIMIZE . |

! 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

Multiple Document Interface Overview, Multiple Document Interface Functions, **DefFrameProc**, **DefWindowProc**, **WM_CHILDACTIVATE**, **WM_GETMINMAXINFO**, **WM_MENUCHAR**, **WM_MOVE**, **WM_SETFOCUS**, **WM_SETTEXT**, **WM_SIZE**, **WM_SYSCOMMAND**

TranslateMDISysAccel

The **TranslateMDISysAccel** function processes accelerator keystrokes for window menu commands of the multiple document interface (MDI) child windows associated

with the specified MDI client window. The function translates **WM_KEYUP** and **WM_KEYDOWN** messages to **WM_SYSCOMMAND** messages, and sends them to the appropriate MDI child windows.

```

BOOL TranslateMDISysAccel(
    HWND hWndClient, // handle to MDI client window
    LPMSG lpMsg      // message data
);

```

Parameters

hWndClient

[in] Handle to the MDI client window.

lpMsg

[in] Pointer to a message retrieved by using the **GetMessage** or **PeekMessage** function. The message must be an **MSG** structure and contain message information from the application's message queue.

Return Values

If the message is translated into a system command, the return value is nonzero.

If the message is not translated into a system command, 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 `winuser.h`; include `windows.h`.

Library: Use `user32.lib`.

See Also

Multiple Document Interface Overview, Multiple Document Interface Functions, **GetMessage**, **PeekMessage**, **TranslateAccelerator**, **MSG**, **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSCOMMAND**

Multiple Document Interface Structures

MDICREATESTRUCT

The **MDICREATESTRUCT** structure contains information about the class, title, owner, location, and size of a multiple document interface (MDI) child window.

```

typedef struct tagMDICREATESTRUCT {
    LPCTSTR szClass;
    LPCTSTR szTitle;

```

(continued)

(continued)

```

HANDLE hOwner;
int x;
int y;
int cx;
int cy;
DWORD style;
LPARAM lParam;
} MDICREATESTRUCT, *LPMDCREATESTRUCT;

```

Members**szClass**

Pointer to a null-terminated string specifying the name of the window class of the MDI child window. The class name must have been registered by a previous call to the **RegisterClass** function.

szTitle

Pointer to a null-terminated string that represents the title of the MDI child window. The system displays the title in the child window's title bar.

hOwner

Handle to the instance of the application creating the MDI client window.

x

Specifies the initial horizontal position, in client coordinates, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default horizontal position.

y

Specifies the initial vertical position, in client coordinates, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default vertical position.

cx

Specifies the initial width, in device units, of the MDI child window. If this member is **CW_USEDEFAULT**, the MDI child window is assigned the default width.

cy

Specifies the initial height, in device units, of the MDI child window. If this member is set to **CW_USEDEFAULT**, the MDI child window is assigned the default height.

style

Specifies the style of the MDI child window. If the MDI client window was created with the **MDIS_ALLCHILDSTYLES** window style, this member can be any combination of the window styles listed in the description of the **CreateWindow** function. Otherwise, this member can be one or more of the following values:

| Value | Meaning |
|-------------|---|
| WS_HSCROLL | Creates an MDI child window that has a horizontal scroll bar. |
| WS_MAXIMIZE | Creates an MDI child window that is initially maximized. |

| Value | Meaning |
|-------------|---|
| WS_MINIMIZE | Creates an MDI child window that is initially minimized. |
| WS_VSCROLL | Creates an MDI child window that has a vertical scroll bar. |

IParam

Specifies an application-defined value.

Remarks

When the MDI child window is created, the system sends the **WM_CREATE** message to the window. The *IParam* parameter of **WM_CREATE** contains a pointer to a **CREATESTRUCT** structure. The **lpCreateParams** member of this structure contains a pointer to the **MDICREATESTRUCT** structure passed with the **WM_MDICREATE** message that created the MDI child window.

! 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*.

Unicode: Declared as Unicode and ANSI structures.

+ See Also

Multiple Document Interface Overview, Multiple Document Interface Structures, **CLIENTCREATESTRUCT**, **CREATESTRUCT**, **WM_CREATE**

Multiple Document Interface Messages**WM_MDIACTIVATE**

An application sends the **WM_MDIACTIVATE** message to a multiple document interface (MDI) client window to instruct the client window to activate a different MDI child window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    WM_MDIACTIVATE,       // message to send
    (LPARAM) wParam,      // handle to child window (HWND)
    (LPARAM) lParam);     // not used; must be zero
```

Parameters

wParam

Handle to the MDI child window to be activated.

lParam

This parameter is not used.

Return Values

If an application sends this message to an MDI client window, the return value is zero.

An MDI child window should return zero if it processes this message.

Remarks

As the client window processes this message, it sends **WM_MDIACTIVATE** to the child window being deactivated and to the child window being activated. The message parameters received by an MDI child window are as follows:

wParam

Handle to the MDI child window being deactivated.

lParam

Handle to the MDI child window being activated.

An MDI child window is activated independently of the MDI frame window. When the frame window becomes active, the child window last activated by using the **WM_MDIACTIVATE** message receives the **WM_NCACTIVATE** message to draw an active window frame and title bar; the child window does not receive another **WM_MDIACTIVATE** 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.

+ See Also

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDIGETACTIVE**, **WM_MDINEXT**, **WM_NCACTIVATE**

WM_MDICASCADE

An application sends the **WM_MDICASCADE** message to a multiple document interface (MDI) client window to arrange all its child windows in a cascade format.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_MDICASCADE,       // message to send
```

```
(WPARAM) wParam;           // cascade option
(LPARAM) lParam;           // not used; must be zero
);
```

Parameters

wParam

Specifies the cascade behavior. This parameter can be one or more of the following values:

| Value | Meaning |
|----------------------|--|
| MDITILE_SKIPDISABLED | Prevents disabled MDI child windows from being cascaded. |
| MDITILE_ZORDER | Windows 2000: Arranges the windows in Z order. |

lParam

This parameter is not used.

Return Values

If the message succeeds, the return value is TRUE.

If the message fails, the return value is 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDIICONARRANGE**, **WM_MDITILE**

WM_MDICREATE

An application sends the **WM_MDICREATE** message to a multiple document interface (MDI) client window to create an MDI child window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    WM_MDICREATE,         // message to send
    (WPARAM) wParam;     // not used; must be zero
    (LPARAM) lParam;     // creation data (LPMDICREATESTRUCT)
);
```


Parameters

wParam

This parameter is not used.

lParam

Pointer to an **MDICREATESTRUCT** structure containing information that the system uses to create the MDI child window.

Return Values

If the message succeeds, the return value is the handle to the new child window.

If the message fails, the return value is NULL.

Remarks

The MDI child window is created with the style bits **WS_CHILD**, **WS_CLIPSIBLINGS**, **WS_CLIPCHILDREN**, **WS_SYSMENU**, **WS_CAPTION**, **WS_THICKFRAME**, **WS_MINIMIZEBOX**, and **WS_MAXIMIZEBOX**, plus additional style bits specified in the **MDICREATESTRUCT** structure. The system adds the title of the new child window to the window menu of the frame window. An application should use this message to create all child windows of the client window.

If an MDI client window receives any message that changes the activation of its child windows while the active child window is maximized, the system restores the active child window and maximizes the newly activated child window.

When an MDI child window is created, the system sends the **WM_CREATE** message to the window. The *lParam* parameter of the **WM_CREATE** message contains a pointer to a **CREATESTRUCT** structure. The **lpCreateParams** member of this structure contains a pointer to the **MDICREATESTRUCT** structure passed with the **WM_MDICREATE** message that created the MDI child window.

An application should not send a second **WM_MDICREATE** message while a **WM_MDICREATE** message is still being processed. For example, it should not send a **WM_MDICREATE** message while an MDI child window is processing its **WM_MDICREATE** 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.

 See Also

Multiple Document Interface Overview, Multiple Document Interface Messages, **CreateMDIWindow**, **CREATESTRUCT**, **MDICREATESTRUCT**, **WM_CREATE**, **WM_MDIDESTROY**

WM_MDIDESTROY

An application sends the **WM_MDIDESTROY** message to a multiple document interface (MDI) client window to close an MDI child window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_MDIDESTROY,       // message to send  
    (WPARAM) wParam,     // handle to child window (HWND)  
    (LPARAM) lParam;     // not used; must be zero  
);
```

Parameters

wParam

Handle to the MDI child window to be closed.

lParam

This parameter is not used.

Return Values

This message always returns zero.

Remarks

This message removes the title of the MDI child window from the MDI frame window and deactivates the child window. An application should use this message to close all MDI child windows.

If an MDI client window receives a message that changes the activation of its child windows and the active MDI child window is maximized, the system restores the active child window and maximizes the newly activated child window.

 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDICREATE**

WM_MDIGETACTIVE

An application sends the **WM_MDIGETACTIVE** message to a multiple document interface (MDI) client window to retrieve the handle to the active MDI child window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    WM_MDIGETACTIVE,     // message to send  
    (WPARAM) wParam,     // not used; must be zero  
    (LPARAM) lParam,     // maximized state (LPBOOL)  
);
```

Parameters

wParam

This parameter is not used.

lParam

Specifies the maximized state. If this parameter is not NULL, it is a pointer to a value that indicates the maximized state of the MDI child window. If the value is TRUE, the window is maximized; a value of FALSE indicates that it is not. If this parameter is NULL, the parameter is ignored.

Return Values

The return value is the handle to the active MDI child window.

! 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

Multiple Document Interface Overview, Multiple Document Interface Messages

WM_MDIICONARRANGE

An application sends the **WM_MDIICONARRANGE** message to a multiple document interface (MDI) client window to arrange all minimized MDI child windows. It does not affect child windows that are not minimized.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_MDIICONARRANGE,    // message to send  
    (LPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

This message has no parameters.

! 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDICASCADE**, **WM_MDIITILE**

WM_MDIMAXIMIZE

An application sends the **WM_MDIMAXIMIZE** message to a multiple document interface (MDI) client window to maximize an MDI child window. The system resizes the child window to make its client area fill the client window. The system places the child window's window-menu icon in the rightmost position of the frame window's menu bar, and places the child window's restore icon in the leftmost position. The system also appends the title bar text of the child window to that of the frame window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_MDIMAXIMIZE,       // message to send  
    (HWND) childWindow,    // handle to child window (HWND)  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

Handle to the MDI child window to be maximized.

lParam

This parameter is not used.

Return Values

The return value is always zero.

Remarks

If an MDI client window receives any message that changes the activation of its child windows while the currently active MDI child window is maximized, the system restores the active child window and maximizes the newly activated child window.

! 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDIRESTORE**

WM_MDINEXT

An application sends the **WM_MDINEXT** message to a multiple document interface (MDI) client window to activate the next or previous child window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_MDINEXT,          // message to send  
    (WPARAM) wParam,     // handle to child window (HWND)  
    (LPARAM) lParam;     // next or previous indicator  
);
```

Parameters

wParam

Handle to the MDI child window. The system activates the child window that is immediately before or after the specified child window, depending on the value of the

fNext parameter. If the *hwndChild* parameter is NULL, the system activates the child window that is immediately before or after the currently active child window.

IParam

If this parameter is zero, the system activates the next MDI child window, and places the child window identified by the *hwndChild* parameter behind all other child windows. If this parameter is nonzero, the system activates the previous child window, placing it in front of the child window identified by *hwndChild*.

Return Values

The return value is always zero.

Remarks

If an MDI client window receives any message that changes the activation of its child windows while the active MDI child window is maximized, the system restores the active child window and maximizes the newly activated child window.

! 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDIACTIVATE**, **WM_MDIGETACTIVE**

WM_MDIREFRESHMENU

An application sends the **WM_MDIREFRESHMENU** message to a multiple document interface (MDI) client window to refresh the window menu of the MDI frame window.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hwnd,           // handle to destination window  
    WM_MDIREFRESHMENU,    // message to send  
    (WPARAM) wParam,      // not used; must be zero  
    (LPARAM) lParam,      // not used; must be zero  
);
```

Parameters

This message has no parameters.

Return Values

If the message succeeds, the return value is the handle to the frame window menu.

If the message fails, the return value is NULL.

Remarks

After sending this message, an application must call the **DrawMenuBar** function to update the menu bar.

! 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **DrawMenuBar**, **WM_MDISETMENU**

WM_MDIESTORE

An application sends the **WM_MDIESTORE** message to a multiple document interface (MDI) client window to restore an MDI child window from maximized size or minimized size.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(  
    (HWND) hWnd,           // handle to destination window  
    WM_MDIESTORE,         // message to send  
    (WPARAM) wParam,      // handle to child window (HWND)  
    (LPARAM) lParam;      // not used; must be zero  
);
```

Parameters

wParam

Handle to the MDI child window to be restored.

lParam

This parameter is not used.

Return Values

The return value is always 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 `winuser.h`; include `windows.h`.

+ See Also

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDIMAXIMIZE**

WM_MDISETMENU

An application sends the **WM_MDISETMENU** message to a multiple document interface (MDI) client window to replace the entire menu of an MDI frame window, to replace the window menu of the frame window, or both.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hWnd,           // handle to destination window
    WM_MDISETMENU,       // message to send
    (WPARAM) wParam,     // handle to menu (HMENU)
    (LPARAM) lParam;     // handle to window menu (HMENU)
);
```

Parameters

wParam

Handle to the new frame window menu. If this parameter is `NULL`, the frame window menu is not changed.

lParam

Handle to the new window menu. If this parameter is `NULL`, the window menu is not changed.

Return Values

If the message succeeds, the return value is the handle to the old frame window menu.

If the message fails, the return value is zero.

Remarks

After sending this message, an application must call the **DrawMenuBar** function to update the menu bar.

If this message replaces the window menu, the MDI child window menu items are removed from the previous window menu and added to the new window menu.

If an MDI child window is maximized and this message replaces the MDI frame window menu, the window-menu icon and restore icon are removed from the previous frame window menu and added to the new frame window menu.

! 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

Multiple Document Interface Overview, Multiple Document Interface Messages, `DrawMenuBar`, `WM_MDIREFRESHMENU`

WM_MDITILE

An application sends the **WM_MDITILE** message to a multiple document interface (MDI) client window to arrange all of its MDI child windows in a tile format.

To send this message, call the **SendMessage** function with the following parameters.

```
SendMessage(
    (HWND) hwnd,           // handle to destination window
    WM_MDITILE,          // message to send
    (WPARAM) wParam,     // tiling option
    (LPARAM) lParam;     // not used; must be zero
);
```

Parameters

wParam

Specifies the tiling option. This parameter can be one of the following values, optionally combined with `MDITILE_SKIPDISABLED` to prevent disabled MDI child windows from being tiled:

| Value | Meaning |
|---------------------------------|----------------------------|
| <code>MDITILE_HORIZONTAL</code> | Tiles windows horizontally |
| <code>MDITILE_VERTICAL</code> | Tiles windows vertically |

lParam

This parameter is not used.

Return Values

If the message succeeds, the return value is `TRUE`.

If the message fails, the return value is 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

Multiple Document Interface Overview, Multiple Document Interface Messages, **WM_MDICASCADE**, **WM_MDIICONARRANGE**

Timers

A *timer* is an internal routine that repeatedly measures a specified interval, in milliseconds.

About Timers

Each time the specified interval (or *time-out value*) for a timer elapses, the system notifies the window associated with the timer. Because the accuracy of a timer depends on the system clock rate, and on how often the application retrieves messages from the message queue, the time-out value is only approximate.

If you need a timer with higher precision, use the high-resolution timer. For more information, see *High-Resolution Timer*. If you need to be notified when a timer elapses, use the waitable timers. For more information, see *Waitable Timer Objects*.

Timer Operations

A Win32-based application creates a timer by using the **SetTimer** function. A new timer starts timing the interval as soon as it is created. An application can change a timer's time-out value by using **SetTimer** and destroy a timer by using the **KillTimer** function. To use system resources efficiently, applications should destroy timers that are no longer necessary.

Each timer has a unique identifier. When creating a timer, an application can either specify an identifier or have the system create a unique value. The first parameter of a **WM_TIMER** message contains the identifier of the timer that posted the message.

If you specify a window handle in the call to **SetTimer**, the application associates the timer with that window. Whenever the time-out value for the timer elapses, the system posts a **WM_TIMER** message to the window associated with the timer. If no window handle is specified in the call to **SetTimer**, the application that created the timer must monitor its message queue for **WM_TIMER** messages and dispatch them to the appropriate window. If you specify a **TimerProc** callback function, the default window

procedure calls the callback function when it processes **WM_TIMER**. Therefore, you need to dispatch messages in the calling thread, even when you use **TimerProc** instead of processing **WM_TIMER**.

High-Resolution Timer

A counter is a general term used in programming to refer to an incrementing variable. Some systems include a *high-resolution performance counter* that provides high-resolution elapsed times.

If a high-resolution performance counter exists on the system, the **QueryPerformanceFrequency** function can be used to express the frequency, in counts per second. The value of the count is processor-dependent. On some processors, for example, the count might be the cycle rate of the processor clock.

The **QueryPerformanceCounter** function retrieves the current value of the high-resolution performance counter (if one exists on the system). By calling this function at the beginning and end of a section of code, an application essentially uses the counter as a high-resolution timer. For example, suppose that **QueryPerformanceFrequency** indicates that the frequency of the high-resolution performance counter is 50,000 counts per second. If the application calls **QueryPerformanceCounter** immediately before and immediately after the section of code to be timed, the counter values might be 1500 counts and 3500 counts, respectively. These values would indicate that .04 seconds (2000 counts) elapsed while the code executed.

Timer Reference

Timer Functions

KillTimer

The **KillTimer** function destroys the specified timer.

```
BOOL KillTimer(  
    HWND hWnd,           // handle to window  
    UINT_PTR uIDEvent    // timer identifier  
);
```

Parameters

hWnd

[in] Handle to the window associated with the specified timer. This value must be the same as the *hWnd* value passed to the **SetTimer** function that created the timer.

uIDEvent

[in] Specifies the timer to be destroyed. If the window handle passed to **SetTimer** is valid, this parameter must be the same as the *uIDEvent* value passed to **SetTimer**.

If the application calls **SetTimer** with *hWnd* set to NULL, this parameter must be the timer identifier returned by **SetTimer**.

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 **KillTimer** function does not remove **WM_TIMER** messages that are already posted to the message queue.

! 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

Timers Overview, Timer Functions, **SetTimer**, **WM_TIMER**

QueryPerformanceCounter

The **QueryPerformanceCounter** function retrieves the current value of the high-resolution performance counter, if one exists.

```
BOOL QueryPerformanceCounter(  
    LARGE_INTEGER *lpPerformanceCount // counter value  
);
```

Parameters

lpPerformanceCount

[out] Pointer to a variable that receives the current performance-counter value, in counts. If the installed hardware does not support a high-resolution performance counter, this parameter can be zero.

Return Values

If the installed hardware supports a high-resolution performance counter, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. For example, if the installed hardware does not support a high-resolution performance counter, the function fails.

Remarks

On a multiprocessor machine, it should not matter which processor is called. However, you can get different results on different processors due to bugs in the basic input/output system (BIOS) or the hardware abstraction layer (HAL). To specify processor affinity for a thread, use the **SetThreadAffinityMask** function.

! 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 winbase.h; include windows.h.

Library: Use kernel32.lib.

+ See Also

Timers Overview, Timer Functions, **QueryPerformanceFrequency**

QueryPerformanceFrequency

The **QueryPerformanceFrequency** function retrieves the frequency of the high-resolution performance counter, if one exists. The frequency cannot change while the system is running.

```
BOOL QueryPerformanceFrequency(  
    LARGE_INTEGER *lpFrequency // current frequency  
);
```

Parameters

lpFrequency

[out] Pointer to a variable that receives the current performance-counter frequency, in counts per second. If the installed hardware does not support a high-resolution performance counter, this parameter can be zero.

Return Values

If the installed hardware supports a high-resolution performance counter, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. For example, if the installed hardware does not support a high-resolution performance counter, the function fails.

! 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 `winbase.h`; include `windows.h`.

Library: Use `kernel32.lib`.

+ See Also

Timers Overview, Timer Functions, **QueryPerformanceCounter**

SetTimer

The **SetTimer** function creates a timer with the specified time-out value.

```
UINT_PTR SetTimer(  
    HWND hwnd,           // handle to window  
    UINT_PTR nIDEvent,   // timer identifier  
    UINT uElapse,        // time-out value  
    TIMERPROC lpTimerFunc // timer procedure  
);
```

Parameters

hwnd

[in] Handle to the window to be associated with the timer. This window must be owned by the calling thread. If this parameter is `NULL`, no window is associated with the timer, and the *nIDEvent* parameter is ignored.

nIDEvent

[in] Specifies a nonzero timer identifier. If the *hwnd* parameter is `NULL`, this parameter is ignored. If the *hwnd* parameter is not `NULL` and the window specified by *hwnd* already has a timer with the value *nIDEvent*, then the existing timer is replaced by the new timer.

uElapse

[in] Specifies the time-out value, in milliseconds.

lpTimerFunc

[in] Pointer to the function to be notified when the time-out value elapses. For more information about the function, see **TimerProc**.

If *lpTimerFunc* is `NULL`, the system posts a **WM_TIMER** message to the application queue. The **hwnd** member of the message's **MSG** structure contains the value of the *hwnd* parameter.

Return Values

If the function succeeds, the return value is an integer identifying the new timer. An application can pass this value, or the string identifier, if it exists, to the **KillTimer** function to destroy the timer.

If the function fails to create a timer, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

An application can process **WM_TIMER** messages by including a **WM_TIMER** case statement in the window procedure or by specifying a **TimerProc** callback function when creating the timer. When you specify a **TimerProc** callback function, the default window procedure calls the callback function when it processes **WM_TIMER**. Therefore, you need to dispatch messages in the calling thread, even when you use **TimerProc** instead of processing **WM_TIMER**.

The *wParam* parameter of the **WM_TIMER** message contains the value of the *nIDEvent* parameter.

The timer identifier, *nIDEvent*, is specific to the associated window. Another window can have its own timer, which has the same identifier as a timer owned by another window. The timers are distinct.

! 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

Timers Overview, Timer Functions, **KillTimer**, **MSG**, **TimerProc**, **WM_TIMER**

TimerProc

The **TimerProc** function is an application-defined callback function that processes **WM_TIMER** messages. The **TIMERPROC** type defines a pointer to this callback function. **TimerProc** is a placeholder for the application-defined function name.

```
VOID CALLBACK TimerProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_TIMER message  
    UINT_PTR idEvent,   // timer identifier  
    DWORD dwTime       // current system time  
);
```

Parameters

hwnd

[in] Handle to the window associated with the timer.

uMsg

[in] Specifies the **WM_TIMER** message.

idEvent

[in] Specifies the timer's identifier.

dwTime

[in] Specifies the number of milliseconds that have elapsed since the system was started. This is the value returned by the **GetTickCount** function.

Return Values

This function does not return a value.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.

Windows 95/98: Requires Windows 98.

Windows CE: Requires version 1.0 or later.

Header: Declared in `winuser.h`; include `windows.h`.

+ See Also

Timers Overview, Timer Functions, **GetTickCount**, **KillTimer**, **SetTimer**, **WM_TIMER**

Timer Messages

The following message is used with timers:

WM_TIMER

WM_TIMER

The **WM_TIMER** message is posted to the installing thread's message queue when a timer expires. The message is posted by the **GetMessage** or **PeekMessage** function.

A window receives this message through its **WindowProc** function.

```
HRESULT CALLBACK WindowProc(  
    HWND hwnd,           // handle to window  
    UINT uMsg,          // WM_TIMER  
    WPARAM wParam,     // timer identifier  
    LPARAM lParam       // timer callback (TIMERPROC)  
);
```


Parameters

wParam

Specifies the timer identifier.

lParam

Pointer to an application-defined callback function that was passed to the **SetTimer** function when the timer was installed.

Return Values

An application should return zero if it processes this message.

Remarks

You can process the message by providing a **WM_TIMER** case in the window procedure. Otherwise, the default window procedure will call the **TimerProc** callback function specified in the call to the **SetTimer** function used to install the timer.

The **WM_TIMER** message is a low-priority message. The **GetMessage** and **PeekMessage** functions post this message only when no other higher-priority messages are in the thread's message queue.

! 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

Timers Overview, Timer Messages, **GetMessage**, **PeekMessage**, **SetTimer**, **TimerProc**

Window Classes

A *window class* is a set of attributes that the system uses as a template to create a window. Every window is a member of a window class. All window classes are process-specific.

About Window Classes

Each window class has an associated window procedure shared by all windows of the same class. The window procedure processes messages for all windows of that class and, therefore, controls their behavior and appearance. For more information, see *Window Procedures*.

A process must register a window class before it can create a window of that class. Registering a window class associates a window procedure, class styles, and other class

attributes with a class name. When a process specifies a class name in the **CreateWindow** or **CreateWindowEx** function, the system creates a window with the window procedure, styles, and other attributes associated with that class name.

Types of Window Classes

There are three types of window classes:

- System classes
- Application global classes
- Application local classes

These types differ in scope, and in when and how they are registered and destroyed. Complete information about how to work with each of the three types of window classes is provided on the companion DVD.

Getting More Information About Window Classes

It's important to be aware of the various issues surrounding window classes. The companion DVD that is bundled inside the Base Services volume of the *Microsoft Win32 Developer's Reference Library* has the complete set of reference information for window classes.

I have included the basic overview of window classes in this printed version to provide you with some familiarity with window classes. As an extension of this overview, and in an effort to provide you with the most complete and comprehensive guide to Win32 development, the companion DVD for *Microsoft Win32 Developer's Library* includes the complete set of information pertaining to window classes. If you have not already done so, go through the installation process on the companion DVD, and all the window class information you need, including detailed usage information and programmatic reference (and all sorts of other MSDN-like programming information) will be a click away.

Window Procedures

Every window has an associated *window procedure*—a function that processes all messages sent or posted to all windows of the class. All aspects of a window's appearance and behavior depend on the window procedure's response to these messages.

About Window Procedures

Each window is a member of a particular window class. The window class determines the default window procedure that an individual window uses to process its messages. All windows belonging to the same class use the same default window procedure. For example, the system defines a window procedure for the combo box class (COMBOBOX); then, all combo boxes use that window procedure.

An application typically registers at least one new window class and its associated window procedure. After registering a class, the application can create many windows of that class, all of which use the same window procedure. Because this means several sources could simultaneously call the same piece of code, you must be careful when modifying shared resources from a window procedure. For more information, see *Window Classes*.

Window procedures for dialog boxes (called dialog box procedures) have a similar structure, and function as regular window procedures. All points referring to window procedures in this section also apply to dialog box procedures. For more information, see *Dialog Boxes*.

Window Procedure Reference

Window Procedure Functions

CallWindowProc

The **CallWindowProc** function passes message information to the specified window procedure.

```

LRESULT CallWindowProc(
    WNDPROC lpPrevWndFunc, // pointer to previous procedure
    HWND hwnd,             // handle to window
    UINT Msg,              // message
    WPARAM wParam,         // first message parameter
    LPARAM lParam           // second message parameter
);

```

Parameters

lpPrevWndFunc

[in] Pointer to the previous window procedure.

If this value is obtained by calling the **GetWindowLong** function with the *nIndex* parameter set to `GWL_WNDPROC` or `DWL_DLGPROC`, it is actually either the address of a window or dialog box procedure, or a handle representing that address.

hwnd

[in] Handle to the window procedure to receive the message.

Msg

[in] Specifies the message.

wParam

[in] Specifies additional message-specific information. The contents of this parameter depend on the value of the *Msg* parameter.

lParam

[in] Specifies additional message-specific information. The contents of this parameter depend on the value of the *Msg* parameter.

Return Values

The return value specifies the result of the message processing and depends on the message sent.

Remarks

Use the **CallWindowProc** function for window subclassing. Usually, all windows with the same class share one window procedure. A subclass is a window or set of windows with the same class whose messages are intercepted and processed by another window procedure (or procedures) before being passed to the window procedure of the class.

The **SetWindowLong** function creates the subclass by changing the window procedure associated with a particular window, causing the system to call the new window procedure instead of the previous one. An application must pass any messages not processed by the new window procedure to the previous window procedure by calling **CallWindowProc**. This allows the application to create a chain of window procedures.

If STRICT is defined, the *lpPrevWndFunc* parameter has the data type **WNDPROC**. The **WNDPROC** type is declared as follows:

```
LRESULT (CALLBACK* WNDPROC) (HWND, UINT, WPARAM, LPARAM);
```

If STRICT is not defined, the *lpPrevWndFunc* parameter has the data type **FARPROC**. The **FARPROC** type is declared as follows:

```
int (FAR WINAPI * FARPROC) ();
```

In C, the **FARPROC** declaration indicates a callback function that has an unspecified parameter list. In C++, however, the empty parameter list in the declaration indicates that a function has no parameters. This subtle distinction can break careless code. Following is one way to handle this situation:

```
#ifdef STRICT
    WNDPROC MyWindowProcedure
#else
    FARPROC MyWindowProcedure
#endif
...
lResult = CallWindowProc(MyWindowProcedure, ...);
```

For further information about functions declared with empty argument lists, refer to *The C++ Programming Language* by Bjarne Stroustrup (Addison-Wesley, 1997).

Windows NT/2000: The **CallWindowProc** function handles Unicode-to-ANSI conversion. You cannot take advantage of this conversion if you call the window procedure directly.

! 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

Window Procedures Overview, Window Procedure Functions, **GetWindowLong**, **SetClassLong**, **SetWindowLong**

DefWindowProc

The **DefWindowProc** function calls the default window procedure to provide default processing for any window messages that an application does not process. This function ensures that every message is processed. **DefWindowProc** is called with the same parameters received by the window procedure.

```
LRESULT DefWindowProc(  
    HWND hWnd,           // handle to window  
    UINT Msg,           // message identifier  
    WPARAM wParam,      // first message parameter  
    LPARAM lParam       // second message parameter  
);
```

Parameters

hWnd

[in] Handle to the window procedure that received the message.

Msg

[in] Specifies the message.

wParam

[in] Specifies additional message information. The content of this parameter depends on the value of the *Msg* parameter.

lParam

[in] Specifies additional message information. The content of this parameter depends on the value of the *Msg* parameter.

Return Values

The return value is the result of the message processing and depends on the message.

! 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

Window Procedures Overview, Window Procedure Functions, **CallWindowProc**, **DefDlgProc**, **WindowProc**

WindowProc

The **WindowProc** function is an application-defined function that processes messages sent to a window. The **WNDPROC** type defines a pointer to this callback function.

WindowProc is a placeholder for the application-defined function name.

```
LRESULT CALLBACK WindowProc(  
    HWND hwnd, // handle to window  
    UINT uMsg, // message identifier  
    WPARAM wParam, // first message parameter  
    LPARAM lParam // second message parameter  
);
```

Parameters

hwnd

[in] Handle to the window.

uMsg

[in] Specifies the message.

wParam

[in] Specifies additional message information. The contents of this parameter depend on the value of the *uMsg* parameter.

lParam

[in] Specifies additional message information. The contents of this parameter depend on the value of the *uMsg* parameter.

Return Values

The return value is the result of the message processing and depends on the message sent.

! 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

Window Procedures Overview, Window Procedure Functions, **CallWindowProc**, **DefWindowProc**, **RegisterClass**

Window Properties

About Window Properties

The Win32 API provides several functions that enable applications to use window properties. Window properties are used typically to associate data with a subclassed window or a window in a multiple document interface (MDI) application. In either case, it is not convenient to use the extra bytes specified in the **CreateWindow** function or class structure for the following two reasons:

- An application might not know how many extra bytes are available or how the space is being used. By using window properties, the application can associate data with a window without accessing the extra bytes.
- An application must access the extra bytes by using offsets. However, window properties are accessed by their string identifiers, not by offsets.

Assigning Window Properties

The **SetProp** function assigns a window property and its string identifier to a window. The **GetProp** function retrieves the window property identified by a given string. The **RemoveProp** function destroys the association between a window and a window property, but does not destroy the data itself.

Enumerating Window Properties

The **EnumProps** and **EnumPropsEx** functions enumerate all of a window's properties by using an application-defined callback function. For more information about the callback function, see **PropEnumProc**.

EnumPropsEx includes an extra parameter for application-defined data used by the callback function. For more information about the callback function, see **PropEnumProcEx**.

Window Property Reference

Window Property Functions

EnumProps

The **EnumProps** function enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. **EnumProps** continues until the last entry is enumerated or the callback function returns FALSE.

To pass application-defined data to the callback function, use the **EnumPropsEx** function.

```
int EnumProps(  
    HWND hWnd, // handle to window  
    PROPENUMPROC lpEnumFunc // callback function  
);
```

Parameters

hWnd

[in] Handle to the window whose property list is to be enumerated.

lpEnumFunc

[in] Pointer to the callback function. For more information about the callback function, see the **PropEnumProc** function.

Return Values

The return value specifies the last value returned by the callback function. It is -1 if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by either other applications or the system itself.

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

Window Properties Overview, Window Property Functions, **EnumPropsEx**, **GetProp**, **PropEnumProc**, **RemoveProp**, **SetProp**

EnumPropsEx

The **EnumPropsEx** function enumerates all entries in the property list of a window by passing them, one by one, to the specified callback function. **EnumPropsEx** continues until the last entry is enumerated or the callback function returns **FALSE**.

```
int EnumPropsEx(
    HWND hWnd,           // handle to window
    PROPENUMPROCEX lpEnumFunc, // callback function
    LPARAM lParam       // application-defined data
);
```

Parameters

hWnd

[in] Handle to the window whose property list is to be enumerated.

lpEnumFunc

[in] Pointer to the callback function. For more information about the callback function, see the **PropEnumProcEx** function.

lParam

[in] Contains application-defined data to be passed to the callback function.

Return Values

The return value specifies the last value returned by the callback function. It is **-1** if the function did not find a property for enumeration.

Remarks

An application can remove only those properties it has added. It must not remove properties added by either other applications or the system itself.

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

Window Properties Overview, Window Property Functions, **GetProp**, **PropEnumProcEx**, **RemoveProp**, **SetProp**

GetProp

The **GetProp** function retrieves a data handle from the property list of the given window. The given character string identifies the handle to be retrieved. The string and handle must have been added to the property list by a previous call to the **SetProp** function.

```
HANDLE GetProp(  
    HWND hWnd,           // handle to window  
    LPCTSTR lpString    // atom or string  
);
```

Parameters

hWnd

[in] Handle to the window whose property list is to be searched.

lpString

[in] Pointer to a null-terminated character string, or contains an atom that identifies a string. If this parameter is an atom, it must have been created by using the **GlobalAddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of the *lpString* parameter; the high-order word must be zero.

Return Values

If the property list contains the given string, the return value is the associated data handle. Otherwise, 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.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Window Properties Overview, Window Property Functions, **EnumProps**, **GlobalAddAtom**, **RemoveProp**, **SetProp**

PropEnumProc

The **PropEnumProc** function is an application-defined callback function used with the **EnumProps** function. The function receives property entries from a window's property list. The **PROPENUMPROC** type defines a pointer to this callback function. **PropEnumProc** is a placeholder for the application-defined function name.

```
BOOL CALLBACK PropEnumProc(  
    HWND hwnd,           // handle to window  
    LPCTSTR lpszString, // string component  
    HANDLE hData        // data handle component  
);
```

Parameters

hwnd

[in] Handle to the window whose property list is being enumerated.

lpszString

[in] Pointer to a null-terminated string. This string is the string component of a property list entry. This is the string that was specified, along with a data handle, when the property was added to the window's property list via a call to the **SetProp** function.

hData

[in] Handle to data. This handle is the data component of a property list entry.

Return Values

Return TRUE to continue the property list enumeration.

Return FALSE to stop the property list enumeration.

Remarks

The following restrictions apply to this callback function:

- The callback function must not yield control or do anything that might yield control to other tasks.
- The callback function can call the **RemoveProp** function. However, **RemoveProp** can remove only the property passed to the callback function through the callback function's parameters.
- The callback function should not attempt to add properties.

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.

Unicode: Declared as Unicode and ANSI prototypes.

+ See Also

Window Properties Overview, Window Property Functions, **EnumProps**, **EnumPropsEx**, **PropEnumProcEx**, **RemoveProp**, **SetProp**

PropEnumProcEx

The **PropEnumProcEx** function is an application-defined callback function used with the **EnumPropsEx** function. The function receives property entries from a window's property list. The **PROPENUMPROCEX** type defines a pointer to this callback function. **PropEnumProcEx** is a placeholder for the application-defined function name.

```
BOOL CALLBACK PropEnumProcEx(  
    HWND hwnd,           // handle to window  
    LPTSTR lpszString,   // string component  
    HANDLE hData,       // data handle component  
    ULONG_PTR dwData    // application-defined data  
);
```

Parameters

hwnd

[in] Handle to the window whose property list is being enumerated.

lpszString

[in] Pointer to a null-terminated string. This string is the string component of a property list entry. This is the string that was specified, along with a data handle, when the property was added to the window's property list via a call to the **SetProp** function.

hData

[in] Handle to data. This handle is the data component of a property list entry.

dwData

[in] Application-defined data. This is the value that was specified as the *lParam* parameter of the call to **EnumPropsEx** that initiated the enumeration.

Return Values

Return TRUE to continue the property list enumeration.

Return FALSE to stop the property list enumeration.

Remarks

The following restrictions apply to this callback function:

- The callback function must not yield control or do anything that might yield control to other tasks.
- The callback function can call the **RemoveProp** function. However, **RemoveProp** can remove only the property passed to the callback function through the callback function's parameters.

- The callback function should not attempt to add properties.

! 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`.

Unicode: Declared as Unicode and ANSI prototypes.

+ See Also

Window Properties Overview, Window Property Functions, **EnumProps**, **EnumPropsEx**, **PropEnumProc**, **RemoveProp**, **SetProp**

RemoveProp

The **RemoveProp** function removes an entry from the property list of the specified window. The specified character string identifies the entry to be removed.

```
HANDLE RemoveProp(
    HWND hWnd,           // handle to window
    LPCTSTR lpString    // atom or string
);
```

Parameters

hWnd

[in] Handle to the window whose property list is to be changed.

lpString

[in] Pointer to a null-terminated character string, or contains an atom that identifies a string. If this parameter is an atom, it must have been created using the **AddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

Return Values

The return value identifies the specified string. If the string cannot be found in the specified property list, the return value is NULL.

Remarks

An application must free the data handles associated with entries removed from a property list. The application can remove only those properties it has added. It must not remove properties added either by other applications or the system itself.

The **RemoveProp** function returns the data handle associated with the string, so that the application can free the data associated with the 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`.

Library: Use `user32.lib`.

Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

+ See Also

Window Properties Overview, Window Property Functions, **AddAtom**, **EnumProps**, **EnumPropsEx**, **GetProp**, **SetProp**

SetProp

The **SetProp** function adds a new entry or changes an existing entry in the property list of the specified window. The function adds a new entry to the list, if the specified character string does not exist already in the list. The new entry contains the string and the handle. Otherwise, the function replaces the string's current handle with the specified handle.

```
BOOL SetProp(  
    HWND hWnd,           // handle to window  
    LPCTSTR lpString,    // atom or string  
    HANDLE hData         // handle to data  
);
```

Parameters

hWnd

[in] Handle to the window whose property list receives the new entry.

lpString

[in] Pointer to a null-terminated string, or contains an atom that identifies a string. If this parameter is an atom, it must be a global atom created by a previous call to the **GlobalAddAtom** function. The atom, a 16-bit value, must be placed in the low-order word of *lpString*; the high-order word must be zero.

hData

[in] Handle to the data to be copied to the property list. The data handle can identify any value useful to the application.

Return Values

If the data handle and string are added to the property list, the return value is nonzero.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Before destroying a window (that is, before processing the `WM_DESTROY` message), an application must remove all entries it has added to the property list. The application must use the **RemoveProp** function to remove the entries.

! 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

Window Properties Overview, Window Property Functions, **EnumProps**, **EnumPropsEx**, **GetProp**, **GlobalAddAtom**, **RemoveProp**

Windows

In a graphical Win32-based application, a window is a rectangular area of the screen where the application displays output and receives input from the user. Therefore, one of the first tasks of a graphical Win32-based application is to create a window.

A window shares the screen with other windows, including those from other applications. Only one window at a time can receive input from the user. The user can use the mouse, keyboard, or other input device to interact with this window and the application that owns it.

About Windows

Desktop Window

When you start the system, it automatically creates the desktop window. The *desktop window* is a system-defined window that paints the background of the screen and serves as the base for all windows displayed by all applications.

The desktop window uses a bitmap to paint the background of the screen. The pattern created by the bitmap is called the *desktop wallpaper*. By default, the desktop window uses the bitmap from a `.bmp` file specified in the registry as the desktop wallpaper.

The **GetDesktopWindow** function returns a handle to the desktop window.

A system configuration application, such as a Control Panel applet, changes the desktop wallpaper by using the **SystemParametersInfo** function with the *wAction* parameter set to `SPI_SETDESKWALLPAPER` and the *lpvParam* parameter specifying a bitmap file

name. **SystemParametersInfo** then loads the bitmap from the specified file, uses the bitmap to paint the background of the screen, and enters the new file name in the registry.

Application Windows

Every graphical Win32-based application creates at least one window, called the *main window*, that serves as the primary interface between the user and the application. Most applications also create other windows, either directly or indirectly, to perform tasks related to the main window. Each window plays a part in displaying output and receiving input from the user.

When you start an application, the system also associates a taskbar button with the application. The *taskbar button* contains the program icon and title. When the application is active, its taskbar button is displayed in the pushed state.

An application window includes elements such as a title bar, menu bar, window menu (formerly known as the system menu), minimize button, maximize button, restore button, close button, sizing border, client area, horizontal scroll bar, and vertical scroll bar. An application's main window typically includes all of these components. Figure 9-2 shows these components in a typical main window.

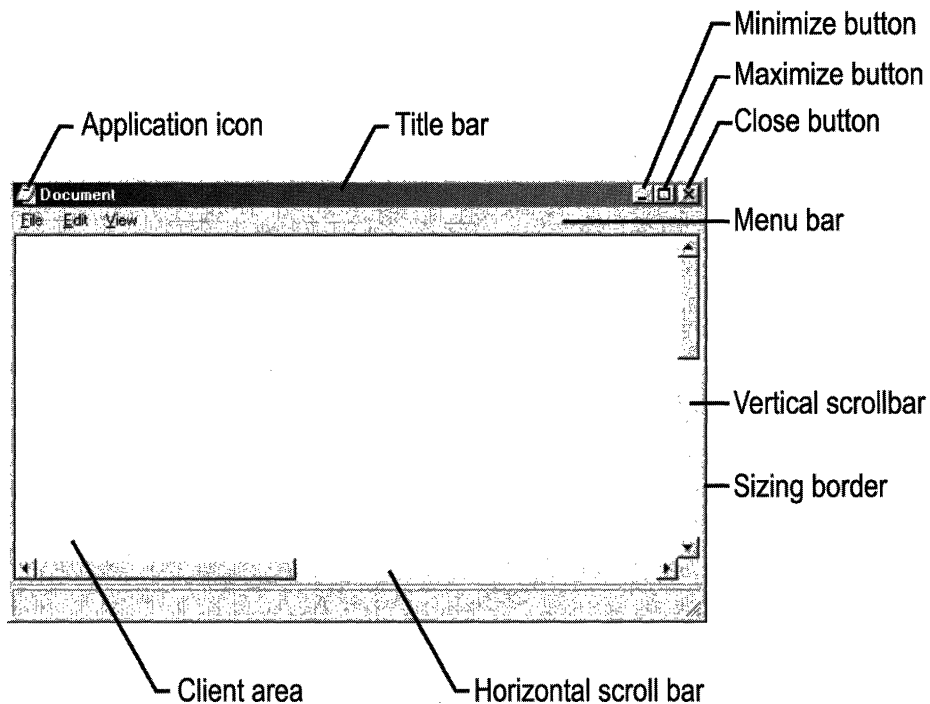


Figure 9-2: Components in an application's main window.

Client Area

The *client area* is the part of a window where the application displays output, such as text or graphics. For example, a desktop-publishing application displays the current page of a document in the client area. The application must provide a function, called a window procedure, to process input to the window and display output in the client area. For more information, see *Window Procedures*.

Nonclient Area

The title bar, menu bar, window menu, minimize and maximize buttons, sizing border, and scroll bars are referred to collectively as the window's *nonclient area*. The system manages most aspects of the nonclient area; the application manages the appearance and behavior of its client area.

The *title bar* displays an application-defined icon and line of text; typically, the text specifies the name of the application or indicates the purpose of the window. An application specifies the icon and text when creating the window. The title bar also makes it possible for the user to move the window by using a mouse or other pointing device.

Most applications include a *menu bar* that lists the commands supported by the application. Items in the menu bar represent the main categories of commands. Clicking an item on the menu bar typically opens a shortcut menu whose items correspond to the tasks within a given category. By clicking a command, the user directs the application to carry out a task.

The *window menu* is created and managed by the system. It contains a standard set of menu items that, when chosen by the user, set a window's size or position, close the application, or perform tasks. For more information, see *Menus*.

The buttons in the upper-right corner affect the size and position of the window. When you click the *maximize button*, the system enlarges the window to the size of the screen and positions the window, so that it covers the entire desktop, minus the taskbar. At the same time, the system replaces the maximize button with the restore button. When you click the *restore button*, the system restores the window to its previous size and position. When you click the *minimize button*, the system reduces the window to the size of its taskbar button, positions the window over the taskbar button, and displays the taskbar button in its normal state. To restore the application to its previous size and position, click its taskbar button. When you click the *close button*, the application exits.

The *sizing border* is an area around the perimeter of the window that enables the user to size the window by using a mouse or other pointing device.

The *horizontal scroll bar* and *vertical scroll bar* convert mouse or keyboard input into values that an application uses to shift the contents of the client area either horizontally or vertically, respectively. For example, a word-processing application that displays a lengthy document typically provides a vertical scroll bar to enable the user to scroll up and down through the document.

Getting More Information About Windows

The companion CD that is bundled inside the Base Services volume of the *Microsoft Win32 Developer's Reference Library* has the complete set of reference information for window programming, with detailed overview information and complete programming element reference.

The overview that is provided in the previous section is geared toward getting you familiar with window programming, but there is a lot more overview and programming information to be had (approximately 200 pages).

Publishing constraints associated with volumes in the Windows Programming Reference Series—which are governed by the mission to provide concise, compact, and portable reference books—did not allow the complete set of window programming reference to be included in this volume. However, to provide you with all the Win32 programming information you need, the *Microsoft Win32 Developer's Reference Library* includes all window programming information on the companion CD. If you have not already done so, install the companion CD, and all window programming information (plus a boatload more) will be at your fingertips.

 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.

Button Reference 53

- Button Functions 53
 - CheckDlgButton
 - CheckRadioButton
 - IsDlgButtonChecked
- Button Messages 56
 - BM_CLICK
 - BM_GETCHECK
 - BM_GETIMAGE
 - BM_GETSTATE
 - BM_SETCHECK
 - BM_SETIMAGE
 - BM_SETSTATE
 - BM_SETSTYLE
 - BN_CLICKED
 - BN_DBLCLK
 - BN_DOUBLECLICKED
 - BN_KILLFOCUS
 - BN_SETFOCUS
 - WM_CTLCOLORBTN

Caret Reference 192

- Caret Functions 192
 - CreateCaret
 - DestroyCaret
 - GetCaretBlinkTime
 - GetCaretPos
 - HideCaret

- SetCaretBlinkTime
- SetCaretPos
- ShowCaret

Combo-Box Reference 73

- Combo-Box Functions 73
 - DlgDirListComboBox
 - DlgDirSelectComboBoxEx
 - GetComboBoxInfo
- Combo-Box Structures 77
 - COMBOBOXINFO
 - COMPAREITEMSTRUCT
 - DRAWITEMSTRUCT
 - MEASUREITEMSTRUCT
- Combo-Box Messages 84
 - CB_ADDSTRING
 - CB_DELETESTRING
 - CB_DIR
 - CB_FINDSTRING
 - CB_FINDSTRINGEXACT
 - CB_GETCOUNT
 - CB_GETCURSEL
 - CB_GETDROPPEDCONTROLRECT
 - CB_GETDROPPEDSTATE
 - CB_GETDROPPEDWIDTH
 - CB_GETEDITSEL
 - CB_GETEXTENDEDUI
 - CB_GETHORIZONTALEXTENT

Combo-Box Reference *(continued)*Combo-Box Messages *(continued)*

CB_GETITEMDATA
 CB_GETITEMHEIGHT
 CB_GETLBTEXT
 CB_GETLBTEXTLEN
 CB_GETLOCALE
 CB_GETTOPINDEX
 CB_INITSTORAGE
 CB_INSERTSTRING
 CB_LIMITTEXT
 CB_RESETCONTENT
 CB_SELECTSTRING
 CB_SETCURSEL
 CB_SETDROPPEDWIDTH
 CB_SETEXTENDSEL
 CB_SETEXTENDEDUI
 CB_SETHORIZONTALAEXTENT
 CB_SETITEMDATA
 CB_SETITEMHEIGHT
 CB_SETLOCALE
 CB_SETTOPINDEX
 CB_SHOWDROPDOWN
 CBN_CLOSEUP
 CBN_DBLCLK
 CBN_DROPDOWN
 CBN_EDITCHANGE
 CBN_EDITUPDATE
 CBN_ERRSPACE
 CBN_KILLFOCUS
 CBN_SELCHANGE
 CBN_SELENCANCEL
 CBN_SELENDOK
 CBN_SETFOCUS
 WM_COMPAREITEM
 WM_DRAWITEM
 WM_MEASUREITEM

Control Reference 50

Control Messages 50
 WM_GETFONT
 WM_SETFONT

Cursor Reference 200

Cursor Functions 200
 ClipCursor
 CopyCursor
 CreateCursor
 DestroyCursor
 GetClipCursor
 GetCursor
 GetCursorInfo
 GetCursorPos
 LoadCursor
 LoadCursorFromFile
 SetCursor

SetCursorPos
 SetSystemCursor
 ShowCursor
 Cursor Structures 216
 CURSORINFO
 Cursor Messages 216
 WM_SETCURSOR

Dialog Box Reference 537

Dialog Box Functions 537
 CreateDialog
 CreateDialogIndirect
 CreateDialogIndirectParam
 CreateDialogParam
 DefDlgProc
 DialogBox
 DialogBoxIndirect
 DialogBoxIndirectParam
 DialogBoxParam
 DialogProc
 EndDialog
 GetDialogBaseUnits
 GetDlgCtrlID
 GetDlgItem
 GetDlgItemInt
 GetDlgItemText
 GetNextDlgGroupItem
 GetNextDlgTabItem
 IsDialogMessage
 MapDialogRect
 MessageBox
 MessageBoxEx
 MessageBoxIndirect
 SendDlgItemMessage
 SetDlgItemInt
 SetDlgItemText
 Dialog Box Structures 582
 DLGITEMTEMPLATE
 DLGITEMTEMPLATEEX
 DLGTEMPLATE
 DLGTEMPLATEEX
 MSGBOXPARAMS
 Dialog Box Messages 595
 DM_GETDEFID
 DM_REPOSITION
 DM_SETDEFID
 WM_CTLCOLORDLG
 WM_ENTERIDLE
 WM_GETDLGCODE
 WM_INITDIALOG
 WM_NEXTDLGCTL

Icon Reference 218

Icon Functions 218
 CopyIcon
 CreateIcon

- CreateIconFromResource
- CreateIconFromResourceEx
- CreateIconIndirect
- DestroyIcon
- DrawIcon
- DrawIconEx
- DuplicateIcon
- ExtractAssociatedIcon
- ExtractIcon
- ExtractIconEx
- GetIconInfo
- LoadIcon
- LookupIconIdFromDirectory
- LookupIconIdFromDirectoryEx
- Icon Structures 239
 - ICONINFO
 - ICONMETRICS
 - WM_ERASEBKGD
 - WM_ICONERASEBKGD
 - WM_PAINTICON
- Keyboard Accelerator Reference 446**
 - Keyboard Accelerator Functions 446
 - CopyAcceleratorTable
 - CreateAcceleratorTable
 - DestroyAcceleratorTable
 - LoadAccelerators
 - TranslateAccelerator
 - Keyboard Accelerator Structures 452
 - ACCEL
 - Keyboard Accelerator Messages 453
 - WM_CHANGEUISTATE
 - WM_INITMENU
 - WM_INITMENUPOPUP
 - WM_MENUCLEAR
 - WM_MENUSELECT
 - WM_QUERYUISTATE
 - WM_SYSCHAR
 - WM_SYSCOMMAND
 - WM_UPDATEUISTATE
- Keyboard-Input Reference 467**
 - Keyboard-Input Functions 467
 - ActivateKeyboardLayout
 - BlockInput
 - EnableWindow
 - GetActiveWindow
 - GetAsyncKeyState
 - GetFocus
 - GetKeyboardLayout
 - GetKeyboardLayoutList
 - GetKeyboardLayoutName
 - GetKeyboardState
 - GetKeyNameText
 - GetKeyState
 - GetLastInputInfo
- IsWindowEnabled
- keybd_event
- LoadKeyboardLayout
- MapVirtualKey
- MapVirtualKeyEx
- OemKeyScan
- RegisterHotKey
- SendInput
- SetActiveWindow
- SetFocus
- SetKeyboardState
- ToAscii
- ToAsciiEx
- ToUnicode
- ToUnicodeEx
- UnloadKeyboardLayout
- UnregisterHotKey
- VkKeyScan
- VkKeyScanEx
- Keyboard-Input Structures 509
 - HARDWAREINPUT
 - INPUT
 - KEYBDINPUT
 - LASTINPUTINFO
 - MOUSEINPUT
- Keyboard-Input Messages 517
 - WM_ACTIVATE
 - WM_CHAR
 - WM_DEADCHAR
 - WM_GETHOTKEY
 - WM_HOTKEY
 - WM_KEYDOWN
 - WM_KEYUP
 - WM_KILLFOCUS
 - WM_SETFOCUS
 - WM_SETHOTKEY
 - WM_SYSDEADCHAR
 - WM_SYSKEYDOWN
 - WM_SYSKEYUP
- Menu Reference 246**
 - Menu Functions 246
 - AppendMenu
 - CheckMenuItem
 - CheckMenuRadioItem
 - CreateMenu
 - CreatePopupMenu
 - DeleteMenu
 - DestroyMenu
 - DrawMenuBar
 - EnableMenuItem
 - EndMenu
 - GetMenu
 - GetMenuBarInfo
 - GetMenuCheckMarkDimensions

Menu Reference *(continued)*Menu Functions *(continued)*

GetMenuDefaultItem
 GetMenuInfo
 GetMenuItemCount
 GetMenuItemID
 GetMenuItemInfo
 GetMenuItemRect
 GetMenuState
 GetMenuString
 GetSubMenu
 GetSystemMenu
 HighlightMenuItem
 InsertMenu
 InsertMenuItem
 IsMenu
 LoadMenu
 LoadMenuIndirect
 MenuItemFromPoint
 ModifyMenu
 RemoveMenu
 SetMenu
 SetMenuDefaultItem
 SetMenuInfo
 SetMenuItemBitmaps
 SetMenuItemInfo
 TrackPopupMenu
 TrackPopupMenuEx
 Menu Structures 297
 MDINEXTMENU
 MENUBARINFO
 MENUEX_TEMPLATE_HEADER
 MENUEX_TEMPLATE_ITEM
 MENUGETOBJECTINFO
 MENUINFO
 MENUITEMINFO
 MENUITEMTEMPLATE
 MENUITEMTEMPLATEHEADER
 TPMPARAMS
 Menu Messages 311
 WM_COMMAND
 WM_CONTEXTMENU
 WM_ENTERMENULOOP
 WM_EXITMENULOOP
 WM_MENUCOMMAND
 WM_MENUDRAG
 WM_MENUGETOBJECT
 WM_MENUBUTTONUP
 WM_NEXTMENU
 WM_UNINITMENUPOPUP

Message and Message Queue Reference 614

Message and Message Queue Functions 614
 BroadcastSystemMessage
 DispatchMessage

GetInputState
 GetMessage
 GetMessageExtraInfo
 GetMessagePos
 GetMessageTime
 GetQueueStatus
 InSendMessage
 InSendMessageEx
 PeekMessage
 PostMessage
 PostQuitMessage
 PostThreadMessage
 RegisterWindowMessage
 ReplyMessage
 SendAsyncProc
 SendMessage
 SendMessageCallback
 SendMessageTimeout
 SendNotifyMessage
 SetMessageExtraInfo
 TranslateMessage
 WaitMessage
 Message and Message Queue Structures 645
 MSG
 Message and Message Queue Messages 646
 WM_APP
 WM_USER

Mouse-Input Reference 372

Mouse-Input Functions 372
 DragDetect
 GetCapture
 GetDoubleClickTime
 GetMouseMovePointsEx
 mouse_event
 ReleaseCapture
 SetCapture
 SetDoubleClickTime
 SwapMouseButton
 TrackMouseEvent
 _TrackMouseEvent
 Mouse-Input Structures 385
 MOUSEMOVEPOINT
 TRACKMOUSEEVENT
 Mouse-Input Messages 387
 WM_APPCOMMAND
 WM_CAPTURECHANGED
 WM_LBUTTONDOWNBLCLK
 WM_LBUTTONDOWN
 WM_LBUTTONUP
 WM_MBUTTONDOWNBLCLK
 WM_MBUTTONDOWN
 WM_MBUTTONUP
 WM_MOUSEACTIVATE
 WM_MOUSEHOVER

| | | |
|--|------------|--|
| WM_MOUSELEAVE | | |
| WM_MOUSEMOVE | | |
| WM_MOUSEWHEEL | | |
| WM_NCHITTEST | | |
| WM_NCLBUTTONDBLCLK | | |
| WM_NCLBUTTONDOWN | | |
| WM_NCLBUTTONUP | | |
| WM_NCMBUTTONDBLCLK | | |
| WM_NCMBUTTONDOWN | | |
| WM_NCMBUTTONUP | | |
| WM_NCMOUSEHOVER | | |
| WM_NCMOUSELEAVE | | |
| WM_NCMOUSEMOVE | | |
| WM_NCRBUTTONDBLCLK | | |
| WM_NCRBUTTONDOWN | | |
| WM_NCRBUTTONUP | | |
| WM_NCXBUTTONDBLCLK | | |
| WM_NCXBUTTONDOWN | | |
| WM_NCXBUTTONUP | | |
| WM_RBUTTONDBLCLK | | |
| WM_RBUTTONDOWN | | |
| WM_RBUTTONUP | | |
| WM_XBUTTONDBLCLK | | |
| WM_XBUTTONDOWN | | |
| WM_XBUTTONUP | | |
| Mouse-Input Macros | 437 | |
| GET_APPCOMMAND_LPARAM | | |
| GET_DEVICE_LPARAM | | |
| GET_KEYSTATE_LPARAM | | |
| GET_KEYSTATE_WPARAM | | |
| GET_NCHITTEST_WPARAM | | |
| GET_XBUTTON_WPARAM | | |
| GET_WHEEL_DELTA_WPARAM | | |
| Multiple Document Interface Reference | 653 | |
| Multiple Document Interface Functions | 653 | |
| CreateMDIWindow | | |
| DefFrameProc | | |
| DefMDIChildProc | | |
| TranslateMDISysAccel | | |
| Multiple Document Interface Structures | 659 | |
| MDICREATESTRUCT | | |
| Multiple Document Interface Messages | 661 | |
| WM_MDIACTIVATE | | |
| WM_MDICASCADE | | |
| WM_MDICREATE | | |
| WM_MDIDESTROY | | |
| WM_MDIGETACTIVE | | |
| WM_MDIICONARRANGE | | |
| WM_MDIMAXIMIZE | | |
| WM_MDIINEXT | | |
| WM_MDIREFRESHMENU | | |
| WM_MDIRESTORE | | |
| WM_MDISETMENU | | |
| WM_MDI TILE | | |
| Scroll-Bar Reference | 134 | |
| Scroll-Bar Functions | 134 | |
| EnableScrollBar | | |
| GetScrollBarInfo | | |
| GetScrollInfo | | |
| GetScrollPos | | |
| GetScrollRange | | |
| ScrollDC | | |
| ScrollWindow | | |
| ScrollWindowEx | | |
| SetScrollInfo | | |
| SetScrollPos | | |
| SetScrollRange | | |
| ShowScrollBar | | |
| Scroll-Bar Structures | 154 | |
| SCROLLBARINFO | | |
| SCROLLINFO | | |
| Scroll-Bar Messages | 157 | |
| SBM_ENABLE_ARROWS | | |
| SBM_GETPOS | | |
| SBM_GETRANGE | | |
| SBM_GETSCROLLINFO | | |
| SBM_SETPOS | | |
| SBM_SETRANGE | | |
| SBM_SETRANGEREDRAW | | |
| SBM_SETSCROLLINFO | | |
| WM_CTLCOLORSCROLLBAR | | |
| WM_HSCROLL | | |
| WM_VSCROLL | | |
| Static-Control Reference | 173 | |
| Static-Control Messages | 173 | |
| STM_GETICON | | |
| STM_GETIMAGE | | |
| STM_SETICON | | |
| STM_SETIMAGE | | |
| STN_CLICKED | | |
| STN_DBLCLK | | |
| STN_DISABLE | | |
| STN_ENABLE | | |
| WM_CTLCOLORSTATIC | | |
| String Reference | 323 | |
| String Functions | 323 | |
| CharLower | | |
| CharLowerBuff | | |
| CharNext | | |
| CharNextExA | | |
| CharPrev | | |
| CharPrevExA | | |
| CharToOem | | |
| CharToOemBuff | | |
| CharUpper | | |
| CharUpperBuff | | |
| CompareString | | |
| FoldString | | |

String Reference *(continued)*

String Functions *(continued)*

GetStringTypeA
GetStringTypeEx
GetStringTypeW
IsCharAlpha
IsCharAlphaNumeric
IsCharLower
IsCharUpper
LoadString
Istrcat
Istrcmp
Istrcmpi
Istrcpy
Istrcpyn
Istrlen
OemToChar
OemToCharBuff
wsprintf
wvsprintf

Timer Reference 674

Timer Functions 674
KillTimer

QueryPerformanceCounter
QueryPerformanceFrequency
SetTimer
TimerProc
Timer Messages 679
WM_TIMER

Window Procedure Reference 682

Window Procedure Functions 682
CallWindowProc
DefWindowProc
WindowProc

Window Property Reference 687

Window Property Functions 687
EnumProps
EnumPropsEx
GetProp
PropEnumProc
PropEnumProcEx
RemoveProp
SetProp

APPENDIX B

Index B: Volume 1, Elements Listed Alphabetically

A

| | |
|---------------------------------|-----|
| AbnormalTermination | 750 |
| AddAtom | 346 |
| AddUsersToEncryptedFile | 655 |
| AllocateUserPhysicalPages | 261 |
| AreFileApisANSI | 481 |
| AssignProcessToJobObject | 74 |
| AttachThreadInput | 75 |

B

| | |
|----------------------------------|-----|
| Beep | 767 |
| BindIoCompletionCallback | 77 |
| BY_HANDLE_FILE_INFORMATION | 606 |

C

| | |
|-----------------------------|-----|
| CallMsgFilter | 420 |
| CallNextHookEx | 421 |
| CallWndProc | 422 |
| CallWndRetProc | 424 |
| Canceled | 482 |
| CBT_CREATEWND | 456 |
| CBTACTIVATESTRUCT | 456 |
| CBTProc | 425 |
| ChangeClipboardChain | 363 |
| CHARSETINFO | 810 |
| CloseClipboard | 364 |
| CloseHandle | 404 |
| CommandLineToArgvW | 78 |
| ConvertThreadToFiber | 79 |
| COPYDATASTRUCT | 343 |
| CopyFile | 483 |
| CopyFileEx | 485 |
| CopyMemory | 263 |
| CopyProgressRoutine | 486 |
| CountClipboardFormats | 364 |
| CreateDirectory | 488 |
| CreateDirectoryEx | 489 |
| CreateFiber | 80 |
| CreateFile | 491 |
| CreateHardLink | 656 |

| | |
|-------------------------------|-----|
| CreateIoCompletionPort | 502 |
| CreateJobObject | 81 |
| CreateProcess | 82 |
| CreateProcessAsUser | 92 |
| CreateProcessWithLogonW | 100 |
| CreateRemoteThread | 107 |
| CreateThread | 110 |
| CWPRETSTRUCT | 457 |
| CWPSTRUCT | 458 |

D

| | |
|----------------------------------|-----|
| DEBUGHOOKINFO | 459 |
| DebugProc | 429 |
| DecryptFile | 658 |
| DefineDosDevice | 504 |
| DeleteAtom | 347 |
| DeleteFiber | 112 |
| DeleteFile | 506 |
| DeleteVolumeMountPoint | 659 |
| DisableThreadLibraryCalls | 217 |
| DISKQUOTA_USER_INFORMATION | 731 |
| DllMain | 219 |
| DuplicateHandle | 406 |

E

| | |
|-----------------------------------|-----|
| EFS_CERTIFICATE_BLOB | 732 |
| EFS_HASH_BLOB | 733 |
| EmptyClipboard | 365 |
| EncryptFile | 660 |
| ENCRYPTION_CERTIFICATE | 733 |
| ENCRYPTION_CERTIFICATE_HASH | 734 |
| ENCRYPTION_CERTIFICATE_HASH_ | |
| LIST | 735 |
| ENCRYPTION_CERTIFICATE_LIST | 735 |
| EncryptionDisable | 661 |
| EnumClipboardFormats | 366 |
| EVENTMSG | 460 |
| EXCEPTION_POINTERS | 759 |
| EXCEPTION_RECORD | 759 |
| ExitProcess | 113 |
| ExitThread | 115 |

F

| | | | |
|---|-----|--|-----|
| FatalAppExit | 768 | GetDiskFreeSpace | 523 |
| FiberProc | 116 | GetDiskFreeSpaceEx | 525 |
| FILE_NOTIFY_INFORMATION | 609 | GetDriveType | 526 |
| FileEncryptionStatus | 662 | GetEnvironmentStrings | 122 |
| FileIOCompletionRoutine | 507 | GetEnvironmentVariable | 123 |
| FillMemory | 264 | GetExceptionCode | 751 |
| FindAtom | 348 | GetExceptionInformation | 753 |
| FindClose | 509 | GetExitCodeProcess | 124 |
| FindCloseChangeNotification | 510 | GetExitCodeThread | 125 |
| FINDEX_INFO_LEVELS | 617 | GetFiberData | 207 |
| FINDEX_SEARCH_OPS | 618 | GetFileAttributes | 527 |
| FindFirstChangeNotification | 511 | GetFileAttributesEx | 530 |
| FindFirstFile | 513 | GetFileInformationByHandle | 531 |
| FindFirstFileEx | 514 | GetFileSize | 532 |
| FindFirstVolume | 663 | GetFileSizeEx | 533 |
| FindFirstVolumeMountPoint | 665 | GetFileType | 534 |
| FindNextChangeNotification | 517 | GetFullPathName | 535 |
| FindNextFile | 518 | GetGuiResources | 126 |
| FindNextVolume | 666 | GetHandleInformation | 413 |
| FindNextVolumeMountPoint | 667 | GetLastError | 776 |
| FindVolumeClose | 668 | GetLogicalDrives | 536 |
| FindVolumeMountPointClose | 669 | GetLogicalDriveStrings | 537 |
| FlashWindow | 769 | GetLongPathName | 538 |
| FlashWindowEx | 770 | GetModuleFileName | 224 |
| FLASHWINFO | 783 | GetModuleHandle | 225 |
| FlushFileBuffers | 519 | GetMsgProc | 433 |
| FONTSIGNATURE | 810 | GetOpenClipboardWindow | 371 |
| ForegroundIdleProc | 432 | GetPriorityClass | 127 |
| FormatMessage | 771 | GetPriorityClipboardFormat | 372 |
| FreeEncryptionCertificateHashList | 670 | GetProcAddress | 226 |
| FreeEnvironmentStrings | 117 | GetProcessAffinityMask | 128 |
| FreeLibrary | 222 | GetProcessHeap | 266 |
| FreeLibraryAndExitThread | 223 | GetProcessHeaps | 267 |
| FreeUserPhysicalPages | 265 | GetProcessIoCounters | 130 |
| | | GetProcessPriorityBoost | 130 |
| | | GetProcessShutdownParameters | 131 |
| | | GetProcessTimes | 132 |
| | | GetProcessVersion | 134 |
| | | GetProcessWorkingSetSize | 135 |
| | | GetQueuedCompletionStatus | 539 |
| | | GetShortPathName | 541 |
| | | GetStartupInfo | 136 |
| | | GetTempFileName | 543 |
| | | GetTempPath | 545 |
| | | GetTextCharset | 795 |
| | | GetTextCharsetInfo | 796 |
| | | GetThreadPriority | 137 |
| | | GetThreadPriorityBoost | 138 |
| | | GetThreadTimes | 139 |
| | | GetVolumeInformation | 672 |
| | | GetVolumeNameForVolumeMountPoint | 675 |
| | | GetVolumePathName | 676 |
| | | GetWriteWatch | 268 |
| | | GlobalAddAtom | 350 |
| | | GlobalDeleteAtom | 352 |

G

| | |
|----------------------------------|-----|
| GET_FILEEX_INFO_LEVELS | 619 |
| GetAtomName | 349 |
| GetBinaryType | 521 |
| GetClipboardData | 367 |
| GetClipboardFormatName | 368 |
| GetClipboardOwner | 369 |
| GetClipboardSequenceNumber | 370 |
| GetClipboardViewer | 371 |
| GetCommandLine | 117 |
| GetCompressedFileSize | 670 |
| GetCurrentDirectory | 522 |
| GetCurrentFiber | 207 |
| GetCurrentProcess | 118 |
| GetCurrentProcessId | 119 |
| GetCurrentThread | 120 |
| GetCurrentThreadId | 121 |

| | | | |
|-------------------------------------|-----|------------------------------------|-----|
| GlobalFindAtom..... | 353 | GetQuotaUsedText..... | 718 |
| GlobalGetAtomName..... | 354 | GetSid..... | 719 |
| GlobalMemoryStatus..... | 269 | GetSidLength..... | 720 |
| | | Invalidate..... | 721 |
| H | | SetQuotaLimit..... | 721 |
| HeapAlloc..... | 271 | SetQuotaThreshold..... | 722 |
| HeapCompact..... | 273 | IDiskQuotaUserBatch..... | 723 |
| HeapCreate..... | 275 | Add..... | 724 |
| HeapDestroy..... | 277 | Remove..... | 725 |
| HeapFree..... | 278 | RemoveAll..... | 726 |
| HeapLock..... | 280 | FlushToDisk..... | 726 |
| HeapReAlloc..... | 281 | IEnumDiskQuotaUsers..... | 727 |
| HeapSize..... | 284 | Clone..... | 728 |
| HeapUnlock..... | 286 | Next..... | 729 |
| HeapValidate..... | 287 | Reset..... | 730 |
| HeapWalk..... | 289 | Skip..... | 730 |
| | | InitAtomTable..... | 355 |
| | | Int32x32To64..... | 546 |
| | | Int64ShllMod32..... | 547 |
| | | Int64ShraMod32..... | 548 |
| | | Int64ShrllMod32..... | 549 |
| | | IO_COUNTERS..... | 184 |
| | | IsBadCodePtr..... | 290 |
| | | IsBadReadPtr..... | 291 |
| | | IsBadStringPtr..... | 293 |
| | | IsBadWritePtr..... | 294 |
| | | IsClipboardFormatAvailable..... | 373 |
| | | IsDBCSLeadByte..... | 798 |
| | | IsDBCSLeadByteEx..... | 799 |
| | | IsReparseTagHighLatency..... | 736 |
| | | IsReparseTagMicrosoft..... | 737 |
| | | IsReparseTagNameSurrogate..... | 738 |
| | | IsTextUnicode..... | 800 |
| | | | |
| | | J | |
| IDiskQuotaControl..... | 683 | JOBOBJECT_ASSOCIATE_COMPLETION_ | |
| AddUserName..... | 684 | PORT..... | 85 |
| AddUserSid..... | 686 | JOBOBJECT_BASIC_ACCOUNTING_ | |
| CreateEnumUsers..... | 688 | INFORMATION..... | 188 |
| CreateUserBatch..... | 690 | JOBOBJECT_BASIC_AND_IO_ACCOUNTING_ | |
| DeleteUser..... | 691 | INFORMATION..... | 190 |
| FindUserName..... | 692 | JOBOBJECT_BASIC_LIMIT_ | |
| FindUserSid..... | 693 | INFORMATION..... | 191 |
| GetDefaultQuotaLimit..... | 694 | JOBOBJECT_BASIC_PROCESS_ID_ | |
| GetDefaultQuotaLimitText..... | 695 | LIST..... | 195 |
| GetDefaultQuotaThreshold..... | 696 | JOBOBJECT_BASIC_UI_ | |
| GetDefaultQuotaThresholdText..... | 697 | RESTRICTIONS..... | 196 |
| GetQuotaLogFlags..... | 698 | JOBOBJECT_END_OF_JOB_TIME_ | |
| GetQuotaState..... | 699 | INFORMATION..... | 197 |
| GiveUserNameResolutionPriority..... | 700 | JOBOBJECT_EXTENDED_LIMIT_ | |
| Initialize..... | 701 | INFORMATION..... | 199 |
| InvalidateSidNameCache..... | 702 | JOBOBJECT_SECURITY_LIMIT_ | |
| SetDefaultQuotaLimit..... | 703 | INFORMATION..... | 200 |
| SetDefaultQuotaThreshold..... | 704 | | |
| SetQuotaLogFlags..... | 705 | | |
| SetQuotaState..... | 706 | | |
| ShutdownNameResolution..... | 707 | | |
| IDiskQuotaEvents..... | 708 | | |
| OnUserNameChanged..... | 708 | | |
| IDiskQuotaUser..... | 709 | | |
| GetAccountStatus..... | 710 | | |
| GetID..... | 711 | | |
| GetName..... | 712 | | |
| GetQuotaInformation..... | 713 | | |
| GetQuotaLimit..... | 714 | | |
| GetQuotaLimitText..... | 715 | | |
| GetQuotaThreshold..... | 716 | | |
| GetQuotaThresholdText..... | 716 | | |
| GetQuotaUsed..... | 717 | | |

| | |
|--------------------------|-----|
| JournalPlaybackProc..... | 434 |
| JournalRecordProc..... | 437 |

K

| | |
|----------------------|-----|
| KBDLLHOOKSTRUCT..... | 460 |
| KeyboardProc..... | 439 |

L

| | |
|---------------------------|-----|
| LARGE_INTEGER..... | 610 |
| LoadLibrary..... | 228 |
| LoadLibraryEx..... | 230 |
| LOCALESIGNATURE..... | 811 |
| LockFile..... | 550 |
| LockFileEx..... | 551 |
| LowLevelKeyboardProc..... | 441 |
| LowLevelMouseProc..... | 442 |

M

| | |
|----------------------------------|-----|
| MAKEINTATOM..... | 356 |
| MapUserPhysicalPages..... | 295 |
| MapUserPhysicalPagesScatter..... | 297 |
| MEMORY_BASIC_INFORMATION..... | 328 |
| MEMORYSTATUS..... | 331 |
| MessageBeep..... | 777 |
| MessageProc..... | 444 |
| METAFILEPICT..... | 378 |
| MOUSEHOOKSTRUCT..... | 462 |
| MOUSEHOOKSTRUCTEX..... | 463 |
| MouseProc..... | 446 |
| MoveFile..... | 553 |
| MoveFileEx..... | 554 |
| MoveFileWithProgress..... | 557 |
| MoveMemory..... | 298 |
| MSLLHOOKSTRUCT..... | 464 |
| MulDiv..... | 560 |
| MultiByteToWideChar..... | 802 |

O

| | |
|--------------------|-----|
| OFSTRUCT..... | 611 |
| OpenClipboard..... | 374 |
| OpenJobObject..... | 141 |
| OpenProcess..... | 142 |
| OpenThread..... | 144 |

P

| | |
|---------------------------------|-----|
| PostQueuedCompletionStatus..... | 561 |
| PROCESS_HEAP_ENTRY..... | 333 |
| PROCESS_INFORMATION..... | 202 |

Q

| | |
|---|-----|
| QueryDosDevice..... | 562 |
| QueryInformationJobObject..... | 146 |
| QueryRecoveryAgentsOnEncryptedFile..... | 677 |
| QueryUsersOnEncryptedFile..... | 678 |
| QueueUserWorkItem..... | 148 |

R

| | |
|-----------------------------------|-----|
| RaiseException..... | 754 |
| ReadDirectoryChangesW..... | 563 |
| ReadFile..... | 567 |
| ReadFileEx..... | 571 |
| ReadFileScatter..... | 574 |
| RegisterClipboardFormat..... | 375 |
| RemoveDirectory..... | 576 |
| RemoveUsersFromEncryptedFile..... | 679 |
| ReplaceFile..... | 577 |
| ResetWriteWatch..... | 299 |
| ResumeThread..... | 150 |

S

| | |
|-----------------------------------|-----|
| SearchPath..... | 580 |
| SetClipboardData..... | 376 |
| SetClipboardViewer..... | 377 |
| SetCurrentDirectory..... | 581 |
| SetEndOfFile..... | 582 |
| SetEnvironmentVariable..... | 151 |
| SetErrorMode..... | 778 |
| SetFileApisToANSI..... | 583 |
| SetFileApisToOEM..... | 585 |
| SetFileAttributes..... | 586 |
| SetFilePointer..... | 588 |
| SetFilePointerEx..... | 591 |
| SetHandleInformation..... | 414 |
| SetInformationJobObject..... | 152 |
| SetLastError..... | 780 |
| SetLastErrorEx..... | 781 |
| SetPriorityClass..... | 153 |
| SetProcessAffinityMask..... | 155 |
| SetProcessPriorityBoost..... | 156 |
| SetProcessShutdownParameters..... | 157 |
| SetProcessWorkingSetSize..... | 159 |
| SetThreadAffinityMask..... | 161 |
| SetThreadIdealProcessor..... | 162 |
| SetThreadPriority..... | 163 |
| SetThreadPriorityBoost..... | 165 |
| SetUnhandledExceptionFilter..... | 756 |
| SetUserFileEncryptionKey..... | 680 |
| SetVolumeLabel..... | 593 |
| SetVolumeMountPoint..... | 681 |
| SetWindowsHookEx..... | 447 |

| | |
|----------------------|-----|
| ShellProc | 451 |
| Sleep | 166 |
| SleepEx | 167 |
| STARTUPINFO | 202 |
| SuspendThread | 169 |
| SwitchToFiber | 170 |
| SwitchToThread | 171 |
| SysMsgProc | 453 |

T

| | |
|----------------------------|-----|
| TerminateJobObject | 172 |
| TerminateProcess | 173 |
| TerminateThread | 174 |
| TEXT | 812 |
| ThreadProc | 176 |
| TlsAlloc | 176 |
| TlsFree | 178 |
| TlsGetValue | 179 |
| TlsSetValue | 180 |
| TranslateCharsetInfo | 805 |

U

| | |
|--------------------------------|-----|
| UInt32x32To64 | 594 |
| ULARGE_INTEGER | 611 |
| UnhandledExceptionFilter | 757 |
| UnhookWindowsHookEx | 455 |
| UnlockFile | 595 |
| UnlockFileEx | 596 |
| UserHandleGrantAccess | 181 |

V

| | |
|----------------------|-----|
| VirtualAlloc | 301 |
| VirtualAllocEx | 306 |
| VirtualFree | 311 |

| | |
|------------------------|-----|
| VirtualFreeEx | 313 |
| VirtualLock | 316 |
| VirtualProtect | 318 |
| VirtualProtectEx | 320 |
| VirtualQuery | 323 |
| VirtualQueryEx | 325 |
| VirtualUnlock | 326 |

W

| | |
|---------------------------------|-----|
| WaitForInputIdle | 182 |
| WideCharToMultiByte | 806 |
| WIN32_FILE_ATTRIBUTE_DATA | 612 |
| WIN32_FIND_DATA | 614 |
| WM_ASKCBFORMATNAME | 380 |
| WM_CANCELJOURNAL | 465 |
| WM_CHANGECHAIN | 381 |
| WM_CLEAR | 382 |
| WM_COPY | 383 |
| WM_COPYDATA | 343 |
| WM_CUT | 383 |
| WM_DESTROYCLIPBOARD | 384 |
| WM_DRAWCLIPBOARD | 385 |
| WM_HSCROLLCLIPBOARD | 386 |
| WM_PAINTCLIPBOARD | 387 |
| WM_PASTE | 388 |
| WM_QUEUESYNC | 467 |
| WM_RENDERALLFORMATS | 389 |
| WM_RENDERFORMAT | 390 |
| WM_SIZECLIPBOARD | 391 |
| WM_VSCROLLCLIPBOARD | 392 |
| WriteFile | 598 |
| WriteFileEx | 601 |
| WriteFileGather | 604 |

Z

| | |
|------------------|-----|
| ZeroMemory | 327 |
|------------------|-----|

APPENDIX B

Index B: Volume 2, Elements Listed Alphabetically

A

| | |
|------------------------------|-----|
| ACCEL..... | 452 |
| ActivateKeyboardLayout | 467 |
| AppendMenu | 246 |

B

| | |
|-----------------------------|-----|
| BlockInput..... | 469 |
| BM_CLICK..... | 56 |
| BM_GETCHECK | 57 |
| BM_GETIMAGE | 58 |
| BM_GETSTATE | 59 |
| BM_SETCHECK | 60 |
| BM_SETIMAGE | 61 |
| BM_SETSTATE | 62 |
| BM_SETSTYLE..... | 63 |
| BN_CLICKED..... | 64 |
| BN_DBLCLK | 65 |
| BN_DOUBLECLICKED..... | 66 |
| BN_KILLFOCUS | 66 |
| BN_SETFOCUS..... | 67 |
| BroadcastSystemMessage..... | 614 |

C

| | |
|-------------------------------|-----|
| CallWindowProc | 682 |
| CB_ADDSTRING | 84 |
| CB_DELETESTRING..... | 85 |
| CB_DIR | 86 |
| CB_FINDSTRING | 88 |
| CB_FINDSTRINGEXACT | 89 |
| CB_GETCOUNT | 90 |
| CB_GETCURSEL | 91 |
| CB_GETDROPPEDCONTROLRECT..... | 92 |
| CB_GETDROPPEDSTATE | 93 |
| CB_GETDROPPEDWIDTH | 93 |
| CB_GETEDITSEL | 94 |
| CB_GETEXTENDEDUI..... | 95 |
| CB_GETHORIZONTALTEXT | 96 |
| CB_GETITEMDATA..... | 97 |
| CB_GETITEMHEIGHT..... | 98 |
| CB_GETLBTEXT | 99 |
| CB_GETLBTEXTLEN | 100 |

| | |
|----------------------------|-----|
| CB_GETLOCALE | 101 |
| CB_GETTOINDEX | 102 |
| CB_INITSTORAGE | 103 |
| CB_INSERTSTRING | 104 |
| CB_LIMITTEXT..... | 105 |
| CB_RESETCONTENT | 106 |
| CB_SELECTSTRING | 106 |
| CB_SETCURSEL | 108 |
| CB_SETDROPPEDWIDTH..... | 108 |
| CB_SETEXTSEL..... | 109 |
| CB_SETEXTENDEDUI..... | 110 |
| CB_SETHORIZONTALTEXT | 111 |
| CB_SETITEMDATA..... | 112 |
| CB_SETITEMHEIGHT..... | 113 |
| CB_SETLOCALE..... | 114 |
| CB_SETTOINDEX | 115 |
| CB_SHOWDROPDOWN..... | 116 |
| CBN_CLOSEUP | 117 |
| CBN_DBLCLK | 118 |
| CBN_DROPDOWN | 119 |
| CBN_EDITCHANGE..... | 120 |
| CBN_EDITUPDATE | 120 |
| CBN_ERRSPACE | 121 |
| CBN_KILLFOCUS | 122 |
| CBN_SELCHANGE | 123 |
| CBN_SELENDCANCEL | 124 |
| CBN_SELENDOK..... | 125 |
| CBN_SETFOCUS..... | 125 |
| CharLower | 323 |
| CharLowerBuff | 324 |
| CharNext | 325 |
| CharNextExA | 326 |
| CharPrev | 327 |
| CharPrevExA | 327 |
| CharToOem | 328 |
| CharToOemBuff..... | 329 |
| CharUpper | 330 |
| CharUpperBuff | 331 |
| CheckDlgButton | 53 |
| CheckMenuItem..... | 249 |
| CheckMenuRadioItem | 250 |
| CheckRadioButton..... | 54 |
| ClipCursor..... | 200 |
| COMBOBOXINFO | 77 |

| | |
|---------------------------------|-----|
| COMPAREITEMSTRUCT | 78 |
| CompareString | 332 |
| CopyAcceleratorTable | 446 |
| CopyCursor | 201 |
| CopyIcon | 218 |
| CreateAcceleratorTable | 447 |
| CreateCaret | 192 |
| CreateCursor | 202 |
| CreateDialog | 537 |
| CreateDialogIndirect | 539 |
| CreateDialogIndirectParam | 541 |
| CreateDialogParam | 543 |
| Createlcon | 219 |
| CreatelconFromResource | 221 |
| CreatelconFromResourceEx | 222 |
| CreatelconIndirect | 224 |
| CreateMDIWindow | 653 |
| CreateMenu | 251 |
| CreatePopupMenu | 252 |
| CURSORINFO | 216 |

D

| | |
|-------------------------------|-----|
| DefDlgProc | 545 |
| DefFrameProc | 655 |
| DefMDIChildProc | 657 |
| DefWindowProc | 684 |
| DeleteMenu | 253 |
| DestroyAcceleratorTable | 448 |
| DestroyCaret | 193 |
| DestroyCursor | 203 |
| DestroyIcon | 225 |
| DestroyMenu | 254 |
| DialogBox | 546 |
| DialogBoxIndirect | 547 |
| DialogBoxIndirectParam | 550 |
| DialogBoxParam | 552 |
| DialogProc | 553 |
| DispatchMessage | 616 |
| DlgDirListComboBox | 73 |
| DlgDirSelectComboBoxEx | 75 |
| DLGITEMTEMPLATE | 582 |
| DLGITEMTEMPLATEEX | 584 |
| DLGTEMPLATE | 586 |
| DLGTEMPLATEEX | 589 |
| DM_GETDEFID | 595 |
| DM_REPOSITION | 596 |
| DM_SETDEFID | 596 |
| DragDetect | 372 |
| DrawIcon | 225 |
| DrawIconEx | 227 |
| DRAWITEMSTRUCT | 80 |
| DrawMenuBar | 255 |
| DuplicateIcon | 229 |

E

| | |
|-----------------------------|-----|
| EnableMenuItem | 256 |
| EnableScrollBar | 134 |
| EnableWindow | 470 |
| EndDialog | 555 |
| EndMenu | 258 |
| EnumProps | 687 |
| EnumPropsEx | 688 |
| ExtractAssociatedIcon | 229 |
| ExtractIcon | 231 |
| ExtractIconEx | 232 |

F

| | |
|------------------|-----|
| FoldString | 336 |
|------------------|-----|

G

| | |
|------------------------------|-----|
| GET_APPCOMMAND_LPARAM | 437 |
| GET_DEVICE_LPARAM | 438 |
| GET_KEYSTATE_LPARAM | 439 |
| GET_KEYSTATE_WPARAM | 440 |
| GET_NCHITTEST_WPARAM | 440 |
| GET_WHEEL_DELTA_WPARAM | 441 |
| GET_XBUTTON_WPARAM | 441 |
| GetActiveWindow | 472 |
| GetAsyncKeyState | 472 |
| GetCapture | 373 |
| GetCaretBlinkTime | 194 |
| GetCaretPos | 195 |
| GetClipCursor | 204 |
| GetComboBoxInfo | 76 |
| GetCursor | 205 |
| GetCursorInfo | 206 |
| GetCursorPos | 207 |
| GetDialogBaseUnits | 556 |
| GetDlgCtrlID | 557 |
| GetDlgItem | 558 |
| GetDlgItemInt | 559 |
| GetDlgItemText | 561 |
| GetDoubleClickTime | 373 |
| GetFocus | 474 |
| GetIconInfo | 233 |
| GetInputState | 617 |
| GetKeyboardLayout | 475 |
| GetKeyboardLayoutList | 476 |
| GetKeyboardLayoutName | 477 |
| GetKeyboardState | 478 |
| GetKeyNameText | 479 |
| GetKeyState | 480 |
| GetLastInputInfo | 482 |
| GetMenu | 258 |
| GetMenuBarInfo | 259 |

| | | | |
|----------------------------------|-----|-----------------------------------|-----|
| GetMenuCheckMarkDimensions | 260 | KEYBDINPUT | 511 |
| GetMenuDefaultItem | 261 | KillTimer | 674 |
| GetMenuInfo | 262 | | |
| GetMenuItemCount | 263 | L | |
| GetMenuItemID | 264 | LASTINPUTINFO | 513 |
| GetMenuItemInfo | 264 | LoadAccelerators | 449 |
| GetMenuItemRect | 266 | LoadCursor | 208 |
| GetMenuState | 267 | LoadCursorFromFile | 209 |
| GetMenuString | 269 | LoadIcon | 235 |
| GetMessage | 618 | LoadKeyboardLayout | 485 |
| GetMessageExtraInfo | 620 | LoadMenu | 278 |
| GetMessagePos | 621 | LoadMenuIndirect | 279 |
| GetMessageTime | 622 | LoadString | 353 |
| GetMouseMovePointsEx | 374 | LookupIconIdFromDirectory | 236 |
| GetNextDlgGroupItem | 562 | LookupIconIdFromDirectoryEx | 238 |
| GetNextDlgTabItem | 563 | lstrcat | 354 |
| GetProp | 689 | lstrcmp | 355 |
| GetQueueStatus | 622 | lstrcmpi | 356 |
| GetScrollBarInfo | 136 | lstrcpy | 358 |
| GetScrollInfo | 137 | lstrcpyn | 359 |
| GetScrollPos | 139 | lstrlen | 360 |
| GetScrollRange | 140 | | |
| GetStringTypeA | 338 | M | |
| GetStringTypeEx | 342 | MapDialogRect | 566 |
| GetStringTypeW | 346 | MapVirtualKey | 487 |
| GetSubMenu | 270 | MapVirtualKeyEx | 489 |
| GetSystemMenu | 271 | MDICREATESTRUCT | 659 |
| | | MDINEXTMENU | 297 |
| H | | MEASUREITEMSTRUCT | 82 |
| HARDWAREINPUT | 509 | MENUBARINFO | 297 |
| HideCaret | 195 | MENUEX_TEMPLATE_HEADER | 298 |
| HiliteMenuItem | 272 | MENUEX_TEMPLATE_ITEM | 299 |
| | | MENUGETOBJECTINFO | 301 |
| I | | MENUINFO | 302 |
| ICONINFO | 239 | MenuItemFromPoint | 280 |
| ICONMETRICS | 240 | MENUITEMINFO | 304 |
| INPUT | 510 | MENUITEMTEMPLATE | 309 |
| InSendMessage | 624 | MENUITEMTEMPLATEHEADER | 310 |
| InSendMessageEx | 625 | MessageBox | 567 |
| InsertMenu | 273 | MessageBoxEx | 572 |
| InsertMenuItem | 276 | MessageBoxIndirect | 577 |
| IsCharAlpha | 350 | ModifyMenu | 281 |
| IsCharAlphaNumeric | 351 | mouse_event | 376 |
| IsCharLower | 352 | MOUSEINPUT | 514 |
| IsCharUpper | 352 | MOUSEMOVEPOINT | 385 |
| IsDialogMessage | 564 | MSG | 645 |
| IsDlgButtonChecked | 55 | MSGBOXPARAMS | 593 |
| IsMenu | 278 | | |
| IsWindowEnabled | 483 | O | |
| | | OemKeyScan | 491 |
| K | | OemToChar | 361 |
| keybd_event | 483 | OemToCharBuff | 361 |

P

| | |
|------------------------|-----|
| PeekMessage..... | 626 |
| PostMessage..... | 628 |
| PostQuitMessage..... | 630 |
| PostThreadMessage..... | 631 |
| PropEnumProc..... | 690 |
| PropEnumProcEx..... | 691 |

Q

| | |
|--------------------------------|-----|
| QueryPerformanceCounter..... | 675 |
| QueryPerformanceFrequency..... | 676 |

R

| | |
|----------------------------|-----|
| RegisterHotKey..... | 492 |
| RegisterWindowMessage..... | 632 |
| ReleaseCapture..... | 379 |
| RemoveMenu..... | 284 |
| RemoveProp..... | 692 |
| ReplyMessage..... | 633 |

S

| | |
|--------------------------|-----|
| SBM_ENABLE_ARROWS..... | 157 |
| SBM_GETPOS..... | 158 |
| SBM_GETRANGE..... | 159 |
| SBM_GETSCROLLINFO..... | 159 |
| SBM_SETPOS..... | 161 |
| SBM_SETRANGE..... | 162 |
| SBM_SETRANGEREDRAW..... | 163 |
| SBM_SETSCROLLINFO..... | 164 |
| SCROLLBARINFO..... | 154 |
| ScrollDC..... | 142 |
| SCROLLINFO..... | 155 |
| ScrollWindow..... | 143 |
| ScrollWindowEx..... | 145 |
| SendAsyncProc..... | 634 |
| SendDlgItemMessage..... | 579 |
| SendInput..... | 494 |
| SendMessage..... | 636 |
| SendMessageCallback..... | 637 |
| SendMessageTimeout..... | 639 |
| SendNotifyMessage..... | 640 |
| SetActiveWindow..... | 495 |
| SetCapture..... | 380 |
| SetCaretBlinkTime..... | 196 |
| SetCaretPos..... | 197 |
| SetCursor..... | 211 |
| SetCursorPos..... | 212 |
| SetDlgItemInt..... | 580 |
| SetDlgItemText..... | 581 |

| | |
|--------------------------|-----|
| SetDoubleClickTime..... | 381 |
| SetFocus..... | 496 |
| SetKeyboardState..... | 497 |
| SetMenu..... | 285 |
| SetMenuDefaultItem..... | 286 |
| SetMenuInfo..... | 287 |
| SetMenuItemBitmaps..... | 288 |
| SetMenuItemInfo..... | 290 |
| SetMessageExtraInfo..... | 642 |
| SetProp..... | 693 |
| SetScrollInfo..... | 147 |
| SetScrollPos..... | 149 |
| SetScrollRange..... | 151 |
| SetSystemCursor..... | 213 |
| SetTimer..... | 677 |
| ShowCaret..... | 198 |
| ShowCursor..... | 215 |
| ShowScrollBar..... | 152 |
| STM_GETICON..... | 173 |
| STM_GETIMAGE..... | 174 |
| STM_SETICON..... | 175 |
| STM_SETIMAGE..... | 176 |
| STN_CLICKED..... | 177 |
| STN_DBLCLK..... | 177 |
| STN_DISABLE..... | 178 |
| STN_ENABLE..... | 179 |
| SwapMouseButton..... | 382 |

T

| | |
|---------------------------|-----|
| TimerProc..... | 678 |
| ToAscii..... | 498 |
| ToAsciiEx..... | 499 |
| ToUnicode..... | 501 |
| ToUnicodeEx..... | 503 |
| TPMPARAMS..... | 310 |
| TrackMouseEvent..... | 383 |
| TRACKMOUSEEVENT..... | 385 |
| TrackPopupMenu..... | 291 |
| TrackPopupMenuEx..... | 294 |
| TranslateAccelerator..... | 450 |
| TranslateMDisysAccel..... | 658 |
| TranslateMessage..... | 642 |

U

| | |
|---------------------------|-----|
| UnloadKeyboardLayout..... | 505 |
| UnregisterHotKey..... | 506 |

V

| | |
|------------------|-----|
| VkKeyScan..... | 507 |
| VkKeyScanEx..... | 508 |

W

| | | | |
|----------------------|-----|--------------------|-----|
| WaitMessage | 644 | WM_MENUCHAR | 456 |
| WindowProc | 685 | WM_MENUCOMMAND | 316 |
| WM_ACTIVATE | 517 | WM_MENUDRAG | 316 |
| WM_APP | 646 | WM_MENUGETOBJECT | 317 |
| WM_APPCOMMAND | 387 | WM_MENURBUTTONUP | 318 |
| WM_CAPTURECHANGED | 390 | WM_MENUSELECT | 458 |
| WM_CHANGEUISTATE | 453 | WM_MOUSEACTIVATE | 399 |
| WM_CHAR | 518 | WM_MOUSEHOVER | 401 |
| WM_COMMAND | 311 | WM_MOUSELEAVE | 402 |
| WM_COMPAREITEM | 126 | WM_MOUSEMOVE | 403 |
| WM_CONTEXTMENU | 312 | WM_MOUSEWHEEL | 404 |
| WM_CTLCOLORBTN | 68 | WM_NCHITTEST | 407 |
| WM_CTLCOLORDLG | 597 | WM_NCLBUTTONDBLCLK | 409 |
| WM_CTLCOLORSCROLLBAR | 165 | WM_NCLBUTTONDOWN | 410 |
| WM_CTLCOLORSTATIC | 180 | WM_NCLBUTTONUP | 411 |
| WM_DEADCHAR | 520 | WM_NCMBUTTONDBLCLK | 412 |
| WM_DRAWITEM | 127 | WM_NCMBUTTONDOWN | 414 |
| WM_ENTERIDLE | 599 | WM_NCMBUTTONUP | 415 |
| WM_ENTERMENULOOP | 314 | WM_NCMOUSEHOVER | 416 |
| WM_ERASEBKGD | 241 | WM_NCMOUSELEAVE | 417 |
| WM_EXITMENULOOP | 315 | WM_NCMOUSEMOVE | 418 |
| WM_GETDLGCODE | 600 | WM_NCRBUTTONDBLCLK | 419 |
| WM_GETFONT | 50 | WM_NCRBUTTONDOWN | 420 |
| WM_GETHOTKEY | 522 | WM_NCRBUTTONUP | 421 |
| WM_HOTKEY | 523 | WM_NCXBUTTONDBLCLK | 423 |
| WM_HSCROLL | 166 | WM_NCXBUTTONDOWN | 424 |
| WM_ICONERASEBKGD | 242 | WM_NCXBUTTONUP | 426 |
| WM_INITDIALOG | 601 | WM_NEXTDLGCTL | 602 |
| WM_INITMENU | 455 | WM_NEXTMENU | 319 |
| WM_INITMENUPOPUP | 456 | WM_PAINTICON | 243 |
| WM_KEYDOWN | 524 | WM_QUERYUISTATE | 459 |
| WM_KEYUP | 526 | WM_RBUTTONDBLCLK | 427 |
| WM_KILLFOCUS | 527 | WM_RBUTTONDOWN | 429 |
| WM_LBUTTONDBLCLK | 391 | WM_RBUTTONUP | 430 |
| WM_LBUTTONDOWN | 392 | WM_SETCURSOR | 217 |
| WM_LBUTTONUP | 394 | WM_SETFOCUS | 528 |
| WM_MBUTTONDBLCLK | 395 | WM_SETFONT | 51 |
| WM_MBUTTONDOWN | 397 | WM_SETHOTKEY | 529 |
| WM_MBUTTONUP | 398 | WM_SYSCHAR | 460 |
| WM_MDIACTIVATE | 661 | WM_SYSCOMMAND | 462 |
| WM_MDICASCADE | 662 | WM_SYSDEADCHAR | 530 |
| WM_MDICREATE | 663 | WM_SYSKEYDOWN | 532 |
| WM_MDIDESTROY | 665 | WM_SYSKEYUP | 534 |
| WM_MDIGETACTIVE | 666 | WM_TIMER | 679 |
| WM_MDIICONARRANGE | 667 | WM_UNINITMENUPOPUP | 320 |
| WM_MDIMAXIMIZE | 667 | WM_UPDATEUISTATE | 464 |
| WM_MDINEXT | 668 | WM_USER | 647 |
| WM_MDIREFRESHMENU | 669 | WM_VSCROLL | 168 |
| WM_MDIRESTORE | 670 | WM_XBUTTONDBLCLK | 431 |
| WM_MDISETMENU | 671 | WM_XBUTTONDOWN | 433 |
| WM_MDITILE | 672 | WM_XBUTTONUP | 435 |
| WM_MEASUREITEM | 128 | wsprintf | 362 |
| | | wvsprintf | 366 |

APPENDIX B

Index B: Volume 3, Elements Listed Alphabetically

A

| | |
|---------------------|-----|
| AbortPath..... | 586 |
| AlphaBlend..... | 66 |
| AngleArc..... | 371 |
| AnimatePalette..... | 202 |
| Arc..... | 373 |
| ArcTo..... | 375 |

B

| | |
|-----------------------|-----|
| BeginPaint..... | 512 |
| BeginPath..... | 587 |
| BitBit..... | 69 |
| BITMAP..... | 116 |
| BITMAPCOREHEADER..... | 118 |
| BITMAPCOREINFO..... | 119 |
| BITMAPFILEHEADER..... | 121 |
| BITMAPINFO..... | 122 |
| BITMAPINFOHEADER..... | 123 |
| BITMAPV4HEADER..... | 128 |
| BITMAPV5HEADER..... | 133 |
| BLENDFUNCTION..... | 140 |

C

| | |
|------------------------------|-----|
| CancelDC..... | 295 |
| ChangeDisplaySettings..... | 296 |
| ChangeDisplaySettingsEx..... | 299 |
| Chord..... | 354 |
| ClientToScreen..... | 252 |
| CloseEnhMetaFile..... | 397 |
| CloseFigure..... | 589 |
| COLORADJUSTMENT..... | 142 |
| COLORREF..... | 223 |
| CombineTransform..... | 253 |
| CopyEnhMetaFile..... | 398 |
| CopyRect..... | 619 |
| CreateBitmap..... | 71 |
| CreateBitmapIndirect..... | 73 |
| CreateBrushIndirect..... | 157 |
| CreateCompatibleBitmap..... | 74 |
| CreateCompatibleDC..... | 303 |
| CreateDC..... | 304 |

| | |
|------------------------------|-----|
| CreateDIBitmap..... | 76 |
| CreateDIBPatternBrushPt..... | 159 |
| CreateDIBSection..... | 78 |
| CreateEnhMetaFile..... | 399 |
| CreateHalftonePalette..... | 203 |
| CreateHatchBrush..... | 160 |
| CreateIC..... | 306 |
| CreatePalette..... | 204 |
| CreatePatternBrush..... | 162 |
| CreatePen..... | 605 |
| CreatePenIndirect..... | 607 |
| CreateSolidBrush..... | 163 |

D

| | |
|------------------------|-----|
| DeleteDC..... | 307 |
| DeleteEnhMetaFile..... | 401 |
| DeleteObject..... | 308 |
| DIBSECTION..... | 145 |
| DISPLAY_DEVICE..... | 344 |
| DPtoLP..... | 254 |
| DrawAnimatedRects..... | 513 |
| DrawCaption..... | 514 |
| DrawEdge..... | 516 |
| DrawEscape..... | 309 |
| DrawFocusRect..... | 518 |
| DrawFrameControl..... | 519 |
| DrawState..... | 522 |
| DrawStateProc..... | 525 |

E

| | |
|---------------------------------|-----|
| Ellipse..... | 356 |
| EMR..... | 421 |
| EMRALPHABLEND..... | 423 |
| EMRANGLEARC..... | 425 |
| EMRARC..... | 426 |
| EMRARCTO..... | 426 |
| EMRCHORD..... | 426 |
| EMRPIE..... | 426 |
| EMRBITBLT..... | 427 |
| EMRCREATEBRUSHINDIRECT..... | 431 |
| EMRCREATECOLORSPACE..... | 432 |
| EMRCREATEDIBPATTERNBRUSHPT..... | 434 |

| | | | |
|--------------------------------|-----|--|-----|
| EMRCREATEMONOBRUSH..... | 435 | EMRSCALEWINDOWEXTEX..... | 468 |
| EMRCREATEPALETTE..... | 436 | EMRSELECTOBJECT..... | 469 |
| EMRCREATEPEN..... | 437 | EMRDELETEOBJECT..... | 469 |
| EMRELLIPSE | | EMRSELECTPALETTE..... | 470 |
| EMRECTANGLE..... | 437 | EMRSETARCDIRECTION..... | 471 |
| EMREOF..... | 438 | EMRSETBKCOLOR..... | 471 |
| EMREXCLUDECLIPRECT..... | 439 | EMRSETTEXTCOLOR..... | 471 |
| EMRINTERSECTCLIPRECT..... | 439 | EMRSETCOLORADJUSTMENT..... | 472 |
| EMREXTCREATEFONTINDIRECTW..... | 439 | EMRSETCOLORSPACE..... | 469 |
| EMREXTCREATEPEN..... | 440 | EMRSELECTCOLORSPACE..... | 469 |
| EMREXTFLOODFILL..... | 441 | EMRDELETETECOLORSPACE..... | 469 |
| EMREXTSELECTCLIPRGN..... | 442 | EMRSETDIBITSTODEVICE..... | 472 |
| EMREXTTEXTOUTA..... | 443 | EMRSETICMPROFILE..... | 474 |
| EMREXTTEXTOUTW..... | 443 | EMRSETMAPPERFLAGS..... | 475 |
| EMRFILLPATH | | EMRSETMITERLIMIT..... | 476 |
| EMRSTROKEANDFILLPATH..... | 444 | EMRSETPALETTEENTRIES..... | 476 |
| EMRSTROKEPATH..... | 444 | EMRSETPIXELV..... | 477 |
| EMRFILLRGN..... | 444 | EMRSETVIEWPORTEXTEX..... | 478 |
| EMRFORMAT..... | 445 | EMRSETWINDOWEXTEX..... | 478 |
| EMRFRAMERGN..... | 446 | EMRSETVIEWPORTORGEX..... | 479 |
| EMRGDICOMMENT..... | 447 | EMRSETWINDOWORGEX..... | 479 |
| EMRGLSBOUNDEDRECORD..... | 448 | EMRSETBRUSHORGEX..... | 479 |
| EMRGLSRECORD..... | 449 | EMRSETWORLDTRANSFORM..... | 479 |
| EMRGRADIENTFILL..... | 450 | EMRSTRETCHBLT..... | 480 |
| EMRINVERTRGN..... | 451 | EMRSTRETCHDIBITS..... | 482 |
| EMRPAINTRGN..... | 451 | EMRTEXT..... | 484 |
| EMRLINETO..... | 452 | EMRTRANSPARENTBLT..... | 485 |
| EMRMOVETOEX..... | 452 | EndPoint..... | 526 |
| EMRMASKBLT..... | 452 | EndPath..... | 590 |
| EMRMODIFYWORLDTRANSFORM..... | 455 | Enhanced Metafile Records with No Parameters..... | 487 |
| EMROFFSETCLIPRGN..... | 455 | Enhanced Metafile Records with One Parameter..... | 487 |
| EMRPIXELFORMAT..... | 456 | EnhMetaFileProc..... | 402 |
| EMRPLGBLT..... | 457 | ENHMETAHEADER..... | 488 |
| EMRPOLYDRAW..... | 459 | ENHMETARECORD..... | 491 |
| EMRPOLYDRAW16..... | 460 | EnumDisplayDevices..... | 310 |
| EMRPOLYLINE..... | 461 | EnumDisplaySettings..... | 311 |
| EMRPOLYBEZIER..... | 461 | EnumDisplaySettingsEx..... | 313 |
| EMRPOLYGON..... | 461 | EnumEnhMetaFile..... | 403 |
| EMRPOLYBEZIERTO..... | 461 | EnumObjects..... | 316 |
| EMRPOLYLINETO..... | 461 | EnumObjectsProc..... | 317 |
| EMRPOLYLINE16..... | 462 | EqualRect..... | 619 |
| EMRPOLYBEZIER16..... | 462 | ExcludeClipRect..... | 177 |
| EMRPOLYGON16..... | 462 | ExcludeUpdateRgn..... | 526 |
| EMRPOLYBEZIERTO16..... | 462 | ExtCreatePen..... | 608 |
| EMRPOLYLINETO16..... | 462 | ExtFloodFill..... | 80 |
| EMRPOLYPOLYLINE..... | 463 | EXTLOGPEN..... | 611 |
| EMRPOLYPOLYGON..... | 463 | ExtSelectClipRgn..... | 178 |
| EMRPOLYPOLYLINE16..... | 464 | | |
| EMRPOLYPOLYGON16..... | 464 | | |
| EMRPOLYTEXTOUTA..... | 464 | | |
| EMRPOLYTEXTOUTW..... | 464 | | |
| EMRRESIZEPALETTE..... | 466 | | |
| EMRSTOREDC..... | 466 | | |
| EMRROUNDRECT..... | 467 | | |
| EMRSCALEVIEWPORTEXTEX..... | 468 | | |
| | | F | |
| | | FillPath..... | 591 |
| | | FillRect..... | 357 |
| | | FlattenPath..... | 592 |

| | | | |
|-----------------------------------|-----|---------------------------|-----|
| FrameRect..... | 358 | GetViewportOrgEx..... | 259 |
| | | GetWindowDC..... | 537 |
| | | GetWindowExtEx..... | 260 |
| | | GetWindowOrgEx..... | 261 |
| | | GetWindowRgn..... | 539 |
| | | GetWinMetaFileBits..... | 413 |
| | | GetWorldTransform..... | 262 |
| | | GRADIENT_RECT..... | 146 |
| | | GRADIENT_TRIANGLE..... | 147 |
| | | GradientFill..... | 88 |
| | | GrayString..... | 540 |
| G | | H | |
| GdiComment..... | 404 | HANDLETABLE..... | 491 |
| GdiFlush..... | 527 | HTUI_ColorAdjustment..... | 211 |
| GdiGetBatchLimit..... | 529 | | |
| GdiSetBatchLimit..... | 530 | I | |
| GetArcDirection..... | 376 | InflateRect..... | 620 |
| GetBitmapDimensionEx..... | 82 | IntersectClipRect..... | 184 |
| GetBkColor..... | 531 | IntersectRect..... | 621 |
| GetBkMode..... | 531 | InvalidateRect..... | 542 |
| GetBoundsRect..... | 532 | InvalidateRgn..... | 543 |
| GetBrushOrgEx..... | 164 | InvertRect..... | 359 |
| GetBValue..... | 226 | IsRectEmpty..... | 622 |
| GetClipBox..... | 180 | | |
| GetClipRgn..... | 181 | L | |
| GetColorAdjustment..... | 205 | LineDDA..... | 377 |
| GetCurrentObject..... | 318 | LineDDAProc..... | 378 |
| GetCurrentPositionEx..... | 255 | LineTo..... | 379 |
| GetDC..... | 319 | LoadBitmap..... | 90 |
| GetDCBrushColor..... | 320 | LockWindowUpdate..... | 544 |
| GetDCEx..... | 321 | LOGBRUSH..... | 169 |
| GetDCOrgEx..... | 323 | LOGBRUSH32..... | 172 |
| GetDCPenColor..... | 324 | LOGPALETTE..... | 224 |
| GetDeviceCaps..... | 325 | LOGPEN..... | 615 |
| GetDIBColorTable..... | 83 | LPtoDP..... | 263 |
| GetDIBits..... | 84 | | |
| GetEnhMetaFile..... | 407 | M | |
| GetEnhMetaFileBits..... | 408 | MAKEPOINTS..... | 631 |
| GetEnhMetaFileHeader..... | 411 | MAKEROP4..... | 152 |
| GetEnhMetaFilePaletteEntries..... | 412 | MapWindowPoints..... | 264 |
| GetGraphicsMode..... | 256 | MaskBit..... | 92 |
| GetGValue..... | 226 | ModifyWorldTransform..... | 265 |
| GetMapMode..... | 257 | MoveToEx..... | 381 |
| GetMetaRgn..... | 182 | | |
| GetMiterLimit..... | 593 | O | |
| GetNearestColor..... | 206 | OffsetClipRgn..... | 185 |
| GetNearestPaletteIndex..... | 207 | OffsetRect..... | 623 |
| GetObject..... | 331 | | |
| GetObjectType..... | 333 | | |
| GetPaletteEntries..... | 208 | | |
| GetPath..... | 594 | | |
| GetPixel..... | 87 | | |
| GetRandomRgn..... | 183 | | |
| GetROP2..... | 533 | | |
| GetRValue..... | 227 | | |
| GetStockObject..... | 334 | | |
| GetStretchBitMode..... | 88 | | |
| GetSysColorBrush..... | 165 | | |
| GetSystemPaletteEntries..... | 209 | | |
| GetSystemPaletteUse..... | 210 | | |
| GetUpdateRect..... | 535 | | |
| GetUpdateRgn..... | 536 | | |
| GetViewportExtEx..... | 258 | | |

| | |
|--------------------------|-----|
| OffsetViewportOrgEx..... | 267 |
| OffsetWindowOrgEx..... | 268 |
| OutputProc | 546 |

P

| | |
|----------------------------|-----|
| PaintDesktop | 547 |
| PAINTSTRUCT | 561 |
| PALETTEENTRY | 224 |
| PALETTEINDEX | 228 |
| PALETTEINDEX | 229 |
| PatBit | 166 |
| PathToRegion | 596 |
| Pie | 360 |
| PlayEnhMetaFile | 415 |
| PlayEnhMetaFileRecord..... | 417 |
| PlgBit | 95 |
| POINT | 629 |
| POINTL..... | 492 |
| POINTS | 629 |
| POINTSTOPOINT | 631 |
| POINTTOPOINTS | 632 |
| PolyBezier | 382 |
| PolyBezierTo | 383 |
| PolyDraw | 384 |
| Polygon..... | 362 |
| Polyline | 386 |
| PolylineTo..... | 387 |
| PolyPolygon..... | 363 |
| PolyPolyline..... | 388 |
| PtInRect..... | 624 |
| PtVisible..... | 186 |

R

| | |
|----------------------|-----|
| RealizePalette | 213 |
| RECT | 630 |
| Rectangle | 364 |
| RECTL..... | 493 |
| RectVisible | 187 |
| RedrawWindow | 547 |
| ReleaseDC | 336 |
| ResetDC | 337 |
| ResizePalette | 214 |
| RestoreDC..... | 338 |
| RGB..... | 230 |
| RGBQUAD | 148 |
| RGBTRIPLE | 149 |
| RoundRect..... | 365 |

S

| | |
|-------------------------|-----|
| SaveDC | 339 |
| ScaleViewportExtEx..... | 269 |

| | |
|---------------------------|-----|
| ScaleWindowExtEx..... | 270 |
| ScreenToClient | 271 |
| SelectClipPath | 188 |
| SelectClipRgn | 189 |
| SelectObject..... | 340 |
| SelectPalette..... | 215 |
| SetArcDirection..... | 389 |
| SetBitmapDimensionEx..... | 97 |
| SetBkColor | 550 |
| SetBkMode | 551 |
| SetBoundsRect..... | 552 |
| SetBrushOrgEx..... | 168 |
| SetColorAdjustment..... | 216 |
| SetDCBrushColor | 342 |
| SetDCPenColor | 343 |
| SetDIBColorTable..... | 98 |
| SetDIBits | 100 |
| SetDIBitsToDevice..... | 102 |
| SetEnhMetaFileBits | 418 |
| SetGraphicsMode | 272 |
| SetMapMode..... | 274 |
| SetMetaRgn | 191 |
| SetMiterLimit | 597 |
| SetPaletteEntries | 217 |
| SetPixel..... | 105 |
| SetPixelV | 106 |
| SetRect | 625 |
| SetRectEmpty | 626 |
| SetROP2..... | 554 |
| SetStretchBltMode..... | 107 |
| SetSystemPaletteUse..... | 219 |
| SetViewportExtEx | 276 |
| SetViewportOrgEx | 278 |
| SetWindowExtEx | 279 |
| SetWindowOrgEx..... | 280 |
| SetWindowRgn | 556 |
| SetWinMetaFileBits | 419 |
| SetWorldTransform..... | 282 |
| SIZE | 150 |
| StretchBlt | 109 |
| StretchDIBits | 111 |
| StrokeAndFillPath | 598 |
| StrokePath | 599 |
| SubtractRect | 626 |

T

| | |
|----------------------|-----|
| TransparentBlt | 114 |
| TRIVERTEX..... | 151 |

U

| | |
|-----------------------|-----|
| UnionRect | 628 |
| UnrealizeObject | 221 |

UpdateColors 222
UpdateWindow 557

V

ValidateRect 558
ValidateRgn 559
VIDEOPARAMETERS 345

W

WidenPath 600
WindowFromDC 560
WM_DEVMODECHANGE 350

WM_DISPLAYCHANGE 562
WM_NCPAINT 563
WM_PAINT 564
WM_PALETTECHANGED 231
WM_PALETTEISCHANGING 232
WM_PRINT 566
WM_PRINTCLIENT 567
WM_QUERYNEWPALETTE 233
WM_SETREDRAW 568
WM_SYNCPAINT 569
WM_SYSCOLORCHANGE 234

X

XFORM 284

APPENDIX B

Index B: Volume 4, Elements Listed Alphabetically

A

| | |
|----------------------------|-----|
| ACM_OPEN | 127 |
| ACM_PLAY | 128 |
| ACM_STOP | 129 |
| ACN_START | 136 |
| ACN_STOP | 136 |
| AddPropSheetPageProc | 435 |
| Animate_Close | 130 |
| Animate_Create | 130 |
| Animate_Open | 131 |
| Animate_OpenEx | 132 |
| Animate_Play | 133 |
| Animate_Seek | 134 |
| Animate_Stop | 135 |

C

| | |
|-------------------------------|-----|
| CBEM_DELETEITEM | 145 |
| CBEM_GETCOMBOCONTROL | 146 |
| CBEM_GETEDITCONTROL | 146 |
| CBEM_GETEXTENDEDSTYLE | 147 |
| CBEM_GETIMAGELIST | 147 |
| CBEM_GETITEM | 148 |
| CBEM_GETUNICODEFORMAT | 149 |
| CBEM_HASEDITCHANGED | 149 |
| CBEM_INSERTITEM | 150 |
| CBEM_SETEXTENDEDSTYLE | 151 |
| CBEM_SETIMAGELIST | 151 |
| CBEM_SETITEM | 152 |
| CBEM_SETUNICODEFORMAT | 153 |
| CBEN_BEGINEDIT | 154 |
| CBEN_DELETEITEM | 154 |
| CBEN_DRAGBEGIN | 155 |
| CBEN_ENDEDIT | 155 |
| CBEN_GETDISPINFO | 156 |
| CBEN_INSERTITEM | 157 |
| CCM_GETUNICODEFORMAT | 86 |
| CCM_GETVERSION | 87 |
| CCM_SETUNICODEFORMAT | 88 |
| CCM_SETVERSION | 89 |
| COLORSCHEME | 104 |
| COMBOBOXEXITEM | 158 |
| CreatePropertySheetPage | 435 |

| | |
|---------------------------|-----|
| CreateStatusWindow | 562 |
| CreateUpDownControl | 735 |

D

| | |
|---------------------------------|-----|
| DateTime_GetMonthCal | 205 |
| DateTime_GetMonthCalColor | 205 |
| DateTime_GetMonthCalFont | 207 |
| DateTime_GetRange | 207 |
| DateTime_GetSystemtime | 208 |
| DateTime_SetFormat | 209 |
| DateTime_SetMonthCalColor | 210 |
| DateTime_SetMonthCalFont | 211 |
| DateTime_SetRange | 211 |
| DateTime_SetSystemtime | 212 |
| DestroyPropertySheetPage | 436 |
| DL_BEGINDRAG | 228 |
| DL_CANCELDRAG | 229 |
| DL_DRAGGING | 230 |
| DL_DROPPED | 230 |
| DRAGLISTINFO | 231 |
| DrawInsert | 226 |
| DrawStatusText | 563 |
| DTM_GETMCCOLOR | 197 |
| DTM_GETMCFONT | 198 |
| DTM_GETMONTHCAL | 198 |
| DTM_GETRANGE | 199 |
| DTM_GETSYSTEMTIME | 200 |
| DTM_SETFORMAT | 200 |
| DTM_SETMCCOLOR | 201 |
| DTM_SETMCFONT | 202 |
| DTM_SETRANGE | 203 |
| DTM_SETSYSTEMTIME | 204 |
| DTN_CLOSEUP | 213 |
| DTN_DATETIMECHANGE | 214 |
| DTN_DROPDOWN | 215 |
| DTN_FORMAT | 216 |
| DTN_FORMATQUERY | 216 |
| DTN_USERSTRING | 217 |
| DTN_WMKEYDOWN | 218 |

E

| | |
|----------------------------------|-----|
| ExtensionPropSheetPageProc | 437 |
|----------------------------------|-----|

F

| | |
|------------------------------|-----|
| FIRST_IPADDRESS | 325 |
| FlatSB_EnableScrollBar | 236 |
| FlatSB_GetScrollInfo | 237 |
| FlatSB_GetScrollPos | 238 |
| FlatSB_GetScrollProp | 239 |
| FlatSB_GetScrollRange | 241 |
| FlatSB_SetScrollInfo | 242 |
| FlatSB_SetScrollPos | 243 |
| FlatSB_SetScrollProp | 244 |
| FlatSB_SetScrollRange | 247 |
| FlatSB_ShowScrollBar | 248 |
| FORWARD_WM_NOTIFY | 92 |
| FOURTH_IPADDRESS | 326 |

G

| | |
|------------------------------|----|
| GetEffectiveClientRect | 81 |
| GetMUILanguage | 82 |

H

| | |
|----------------------------------|-----|
| HANDLE_WM_NOTIFY | 93 |
| HDHITTESTINFO | 301 |
| HDITEM | 303 |
| HDLAYOUT | 306 |
| HDM_CLEARFILTER | 258 |
| HDM_CREATEDRAGIMAGE | 259 |
| HDM_DELETEITEM | 259 |
| HDM_EDITFILTER | 260 |
| HDM_GETBITMAPMARGIN | 261 |
| HDM_GETIMAGELIST | 261 |
| HDM_GETITEM | 262 |
| HDM_GETITEMCOUNT | 262 |
| HDM_GETITEMRECT | 263 |
| HDM_GETORDERARRAY | 264 |
| HDM_GETUNICODEFORMAT | 265 |
| HDM_HITTEST | 265 |
| HDM_INSERTITEM | 266 |
| HDM_LAYOUT | 266 |
| HDM_ORDERTOINDEX | 267 |
| HDM_SETBITMAPMARGIN | 268 |
| HDM_SETFILTERCHANGETIMEOUT | 268 |
| HDM_SETHOTDIVIDER | 269 |
| HDM_SETIMAGELIST | 270 |
| HDM_SETITEM | 271 |
| HDM_SETORDERARRAY | 271 |
| HDM_SETUNICODEFORMAT | 272 |
| HDN_BEGINDRAG | 291 |
| HDN_BEGINTRACK | 292 |
| HDN_DIVIDERDBLCLICK | 292 |
| HDN_ENDDRAG | 293 |
| HDN_ENDTRACK | 293 |

| | |
|-------------------------------------|-----|
| HDN_FILTERBTNCLICK | 294 |
| HDN_FILTERCHANGE | 295 |
| HDN_GETDISPINFO | 295 |
| HDN_ITEMCHANGED | 296 |
| HDN_ITEMCHANGING | 297 |
| HDN_ITEMCLICK | 297 |
| HDN_ITEMDBLCLICK | 298 |
| HDN_TRACK | 298 |
| HDTEXTFILTER Structure | 306 |
| Header_ClearFilter | 274 |
| Header_CreateDragImage | 275 |
| Header_DeleteItem | 275 |
| Header_EditFilter | 276 |
| Header_GetBitmapMargin | 277 |
| Header_GetImageList | 278 |
| Header_GetItem | 278 |
| Header_GetItemCount | 279 |
| Header_GetItemRect | 280 |
| Header_GetOrderArray | 281 |
| Header_GetUnicodeFormat | 282 |
| Header_InsertItem | 282 |
| Header_Layout | 283 |
| Header_OrderToIndex | 284 |
| Header_SetBitmapMargin | 285 |
| Header_SetFilterChangeTimeout | 286 |
| Header_SetHotDivider | 286 |
| Header_SetImageList | 287 |
| Header_SetItem | 288 |
| Header_SetOrderArray | 289 |
| Header_SetUnicodeFormat | 290 |
| HKM_GETHOTKEY | 315 |
| HKM_SETHOTKEY | 316 |
| HKM_SETRULES | 317 |

I

| | |
|-----------------------------|-----|
| INDEXTOSTATEIMAGEMASK | 94 |
| InitCommonControls | 83 |
| InitCommonControlsEx | 83 |
| INITCOMMONCONTROLSEX | 104 |
| InitializeFlatSB | 235 |
| initMUILanguage | 84 |
| IPM_CLEARADDRESS | 320 |
| IPM_GETADDRESS | 321 |
| IPM_ISBLANK | 322 |
| IPM_SETADDRESS | 322 |
| IPM_SETFOCUS | 323 |
| IPM_SETRANGE | 323 |
| IPN_FIELDCHANGED | 324 |

L

| | |
|--------------------|-----|
| LBItemFromPt | 227 |
|--------------------|-----|

M

| | |
|----------------------------------|-----|
| MakeDragList | 228 |
| MAKEIPADDRESS | 326 |
| MAKEIPRANGE | 327 |
| MCHITTESTINFO | 382 |
| MCM_GETCOLOR | 339 |
| MCM_GETCURSEL | 340 |
| MCM_GETFIRSTDAYOFWEEK | 341 |
| MCM_GETMAXSELCOUNT | 342 |
| MCM_GETMAXTODAYWIDTH | 342 |
| MCM_GETMINREQRECT | 343 |
| MCM_GETMONTHDELTA | 344 |
| MCM_GETMONTHRANGE | 345 |
| MCM_GETRANGE | 346 |
| MCM_GETSELRANGE | 347 |
| MCM_GETTODAY | 347 |
| MCM_GETUNICODEFORMAT | 348 |
| MCM_HITTEST | 349 |
| MCM_SETCOLOR | 351 |
| MCM_SETCURSEL | 352 |
| MCM_SETDAYSTATE | 353 |
| MCM_SETFIRSTDAYOFWEEK | 354 |
| MCM_SETMAXSELCOUNT | 354 |
| MCM_SETMONTHDELTA | 355 |
| MCM_SETRANGE | 356 |
| MCM_SETSELRANGE | 357 |
| MCM_SETTODAY | 358 |
| MCM_SETUNICODEFORMAT | 358 |
| MCN_GETDAYSTATE | 379 |
| MCN_SELCHANGE | 380 |
| MCN_SELECT | 380 |
| MenuHelp | 564 |
| MonthCal_GetColor | 359 |
| MonthCal_GetCurSel | 360 |
| MonthCal_GetFirstDayOfWeek | 361 |
| MonthCal_GetMaxSelCount | 362 |
| MonthCal_GetMaxTodayWidth | 363 |
| MonthCal_GetMinReqRect | 363 |
| MonthCal_GetMonthDelta | 364 |
| MonthCal_GetMonthRange | 365 |
| MonthCal_GetRange | 366 |
| MonthCal_GetSelRange | 367 |
| MonthCal_GetToday | 368 |
| MonthCal_GetUnicodeFormat | 368 |
| MonthCal_HitTest | 369 |
| MonthCal_SetColor | 370 |
| MonthCal_SetCurSel | 371 |
| MonthCal_SetDayState | 372 |
| MonthCal_SetFirstDayOfWeek | 373 |
| MonthCal_SetMaxSelCount | 374 |
| MonthCal_SetMonthDelta | 375 |
| MonthCal_SetRange | 376 |
| MonthCal_SetSelRange | 377 |
| MonthCal_SetToday | 377 |

| | |
|---------------------------------|-----|
| MonthCal_SetUnicodeFormat | 378 |
| MONTHDAYSTATE | 385 |

N

| | |
|-------------------------------------|-----|
| NM_CHAR | 95 |
| NM_CLICK | 95 |
| NM_CLICK (status bar) | 578 |
| NM_CLICK (tab) | 639 |
| NM_CUSTOMDRAW | 117 |
| NM_CUSTOMDRAW (header) | 299 |
| NM_CUSTOMDRAW (rebar) | 535 |
| NM_CUSTOMDRAW (Tooltip) | 687 |
| NM_CUSTOMDRAW (trackbar) | 728 |
| NM_DBLCLK | 96 |
| NM_DBLCLK (status bar) | 579 |
| NM_HOVER | 96 |
| NM_KEYDOWN | 97 |
| NM_KILLFOCUS | 98 |
| NM_KILLFOCUS (date time) | 219 |
| NM_NCHITTEST | 98 |
| NM_NCHITTEST (rebar) | 536 |
| NM_OUTOFMEMORY | 99 |
| NM_RCLICK | 99 |
| NM_RCLICK (header) | 300 |
| NM_RCLICK (status bar) | 579 |
| NM_RCLICK (tab) | 639 |
| NM_RDBLCLK | 100 |
| NM_RDBLCLK (status bar) | 580 |
| NM_RELEASEDCAPTURE | 101 |
| NM_RELEASEDCAPTURE (header) | 301 |
| NM_RELEASEDCAPTURE (monthcal) | 381 |
| NM_RELEASEDCAPTURE (pager) | 408 |
| NM_RELEASEDCAPTURE (rebar) | 537 |
| NM_RELEASEDCAPTURE (tab) | 640 |
| NM_RELEASEDCAPTURE (trackbar) | 729 |
| NM_RELEASEDCAPTURE (up-down) | 746 |
| NM_RETURN | 101 |
| NM_SETCURSOR | 102 |
| NM_SETCURSOR (ComboBoxEx) | 157 |
| NM_SETFOCUS | 102 |
| NM_SETFOCUS (date time) | 219 |
| NM_TOOLTIPS_CREATED | 103 |
| NMCBEDRAGBEGIN | 161 |
| NMCBEENDEDIT | 160 |
| NMCHAR | 105 |
| NMCOMBOBOXEX | 161 |
| NMCUSTOMDRAW | 119 |
| NMDATETIMECHANGE | 220 |
| NMDATETIMEFORMAT | 221 |
| NMDATETIMEFORMATQUERY | 222 |
| NMDATETIMESTRING | 223 |
| NMDATETIMEWMKEYDOWN | 224 |
| NMDAYSTATE | 384 |
| NMHDDISPIFNO | 307 |

| | |
|----------------------------------|-----|
| NMHFILTERBTNCLICK Structure..... | 308 |
| NMHDR | 106 |
| NMHEADER..... | 309 |
| NMIPADDRESS | 329 |
| NMKEY..... | 107 |
| NM_MOUSE..... | 107 |
| NMOBJECTNOTIFY | 108 |
| NMPGCALCSIZE | 410 |
| NMPGSCROLL | 411 |
| NMRBAUTOSIZE | 544 |
| NMREBAR..... | 545 |
| NMREBARCHEVRON | 546 |
| NMREBARCHILDSize..... | 547 |
| NMSELCHANGE..... | 384 |
| NMTCKEYDOWN | 644 |
| NMTOOLTIPS_CREATED..... | 109 |
| NMTTCUSTOMDRAW..... | 691 |
| NMTTDispInfo | 691 |
| NMUPDOWN | 747 |

P

| | |
|------------------------------------|-----|
| Pager_ForwardMouse..... | 399 |
| Pager_GetBkColor | 399 |
| Pager_GetBorder | 400 |
| Pager_GetButtonSize..... | 401 |
| Pager_GetButtonState | 401 |
| Pager_GetDropTarget..... | 402 |
| Pager_GetPos..... | 403 |
| Pager_RecalcSize..... | 403 |
| Pager_SetBkColor..... | 404 |
| Pager_SetBorder..... | 405 |
| Pager_SetButtonSize..... | 406 |
| Pager_SetChild | 406 |
| Pager_SetPos | 407 |
| PBM_DELTAPOS | 417 |
| PBM_GETPOS..... | 417 |
| PBM_GETRANGE | 418 |
| PBM_SETBARCOLOR | 419 |
| PBM_SETBKCOLOR..... | 419 |
| PBM_SETPOS..... | 420 |
| PBM_SETRANGE..... | 421 |
| PBM_SETRANGE32..... | 421 |
| PBM_SETSTEP | 422 |
| PBM_STEPIT | 423 |
| PBRANGE..... | 423 |
| PGM_FORWARDMOUSE | 390 |
| PGM_GETBKCOLOR | 391 |
| PGM_GETBORDER | 391 |
| PGM_GETBUTTONSIZE..... | 392 |
| PGM_GETBUTTONSTATE | 392 |
| PGM_GETDROPTARGET..... | 393 |
| PGM_GETPOS..... | 394 |
| PGM_RECALCSIZE..... | 395 |
| PGM_SETBKCOLOR..... | 395 |
| PGM_SETBORDER | 396 |
| PGM_SETBUTTONSIZE..... | 396 |
| PGM_SETCHILD..... | 397 |
| PGM_SETPOS | 398 |
| PGN_CALCSIZE | 409 |
| PGN_SCROLL..... | 409 |
| PropertySheet..... | 438 |
| PropSheet_AddPage..... | 461 |
| PropSheet_Apply..... | 461 |
| PropSheet_CancelToClose | 462 |
| PropSheet_Changed | 463 |
| PropSheet_GetCurrentPageHwnd | 464 |
| PropSheet_GetTabControl | 465 |
| PropSheet_HwndToIndex..... | 465 |
| PropSheet_IdToIndex..... | 466 |
| PropSheet_IndexToHwnd..... | 467 |
| PropSheet_IndexToId..... | 467 |
| PropSheet_IndexToPage | 468 |
| PropSheet_InsertPage | 469 |
| PropSheet_IsDialogMessage | 470 |
| PropSheet_PageToIndex | 471 |
| PropSheet_PressButton | 472 |
| PropSheet_QuerySiblings | 473 |
| PropSheet_RebootSystem | 474 |
| PropSheet_RemovePage..... | 474 |
| PropSheet_RestartWindows..... | 475 |
| PropSheet_SetCurSel | 476 |
| PropSheet_SetCurSelByID..... | 477 |
| PropSheet_SetFinishText..... | 477 |
| PropSheet_SetHeaderSubTitle | 478 |
| PropSheet_SetHeaderTitle..... | 479 |
| PropSheet_SetTitle..... | 480 |
| PropSheet_SetWizButtons | 481 |
| PropSheet_UnChanged..... | 482 |
| PROPSHEETHEADER..... | 493 |
| PROPSHEETPAGE..... | 499 |
| PropSheetPageProc..... | 439 |
| PropSheetProc | 440 |
| PSHNOTIFY | 503 |
| PSM_ADDPAGE | 441 |
| PSM_APPLY..... | 442 |
| PSM_CANCELTOCLOSE | 442 |
| PSM_CHANGED | 443 |
| PSM_GETCURRENTPAGEHWND..... | 444 |
| PSM_GETTABCONTROL..... | 445 |
| PSM_HWNDTOINDEX | 445 |
| PSM_IDTOINDEX | 446 |
| PSM_INDEXTOHWND..... | 446 |
| PSM_INDEXTOID | 447 |
| PSM_INDEXTOPAGE..... | 447 |
| PSM_INSERTPAGE..... | 448 |
| PSM_ISDIALOGMESSAGE | 449 |
| PSM_PAGETOINDEX..... | 450 |
| PSM_PRESSBUTTON..... | 451 |
| PSM_QUERYSIBLINGS..... | 451 |

| | |
|-------------------------------|-----|
| PSM_REBOOTSYSM..... | 452 |
| PSM_REMOVEPAGE..... | 453 |
| PSM_RESTARTWINDOWS..... | 453 |
| PSM_SETCURSEL..... | 454 |
| PSM_SETCURSELID..... | 455 |
| PSM_SETFINISHTEXT..... | 456 |
| PSM_SETHEADERSUBTITLE..... | 456 |
| PSM_SETHHEADERTITLE..... | 457 |
| PSM_SETTITLE..... | 458 |
| PSM_SETWIZBUTTONS..... | 459 |
| PSM_UNCHANGED..... | 460 |
| PSN_APPLY..... | 483 |
| PSN_GETOBJECT..... | 484 |
| PSN_HELP..... | 484 |
| PSN_KILLACTIVE..... | 485 |
| PSN_QUERYCANCEL..... | 486 |
| PSN_QUERYINITIALFOCUS..... | 487 |
| PSN_RESET..... | 488 |
| PSN_SETACTIVE..... | 489 |
| PSN_TRANSLATEACCELERATOR..... | 489 |
| PSN_WIZBACK..... | 490 |
| PSN_WIZFINISH..... | 491 |
| PSN_WIZNEXT..... | 492 |

R

| | |
|--------------------------|-----|
| RB_BEGINDRAG..... | 510 |
| RB_DELETEBAND..... | 511 |
| RB_DRAGMOVE..... | 511 |
| RB_ENDDRAG..... | 512 |
| RB_GETBANDBORDERS..... | 512 |
| RB_GETBANDCOUNT..... | 513 |
| RB_GETBANDINFO..... | 514 |
| RB_GETBARHEIGHT..... | 515 |
| RB_GETBARINFO..... | 515 |
| RB_GETBKCOLOR..... | 516 |
| RB_GETCOLORSCHEME..... | 516 |
| RB_GETDROPTARGET..... | 517 |
| RB_GETPALETTE..... | 518 |
| RB_GETRECT..... | 518 |
| RB_GETROWCOUNT..... | 519 |
| RB_GETROWHEIGHT..... | 519 |
| RB_GETTEXTCOLOR..... | 520 |
| RB_GETTOOLTIPS..... | 520 |
| RB_GETUNICODEFORMAT..... | 521 |
| RB_HITTEST..... | 522 |
| RB_IDTOINDEX..... | 522 |
| RB_INSERTBAND..... | 523 |
| RB_MAXIMIZEBAND..... | 524 |
| RB_MINIMIZEBAND..... | 524 |
| RB_MOVEBAND..... | 525 |
| RB_PUSHCHEVRON..... | 526 |
| RB_SETBANDINFO..... | 527 |
| RB_SETBARINFO..... | 527 |
| RB_SETBKCOLOR..... | 528 |

| | |
|--------------------------|-----|
| RB_SETCOLORSCHEME..... | 529 |
| RB_SETPALETTE..... | 529 |
| RB_SETPARENT..... | 530 |
| RB_SETTEXTCOLOR..... | 531 |
| RB_SETTOOLTIPS..... | 532 |
| RB_SETUNICODEFORMAT..... | 532 |
| RB_SHOWBAND..... | 533 |
| RB_SIZETORECT..... | 534 |
| RBHITTESTINFO..... | 548 |
| RBN_AUTOSIZE..... | 537 |
| RBN_BEGINDRAG..... | 538 |
| RBN_CHEVRONPUSHED..... | 539 |
| RBN_CHILDSize..... | 539 |
| RBN_DELETEBAND..... | 540 |
| RBN_DELETINGBAND..... | 541 |
| RBN_ENDDRAG..... | 541 |
| RBN_GETOBJECT..... | 542 |
| RBN_HEIGHTCHANGE..... | 543 |
| RBN_LAYOUTCHANGED..... | 543 |
| REBARBANDINFO..... | 548 |
| REBARINFO..... | 552 |

S

| | |
|---------------------------|-----|
| SB_GETBORDERS..... | 565 |
| SB_GETICON..... | 566 |
| SB_GETPARTS..... | 566 |
| SB_GETRECT..... | 567 |
| SB_GETTEXT..... | 567 |
| SB_GETTEXTLENGTH..... | 569 |
| SB_GETTIPTEXT..... | 570 |
| SB_GETUNICODEFORMAT..... | 570 |
| SB_ISSIMPLE..... | 571 |
| SB_SETBKCOLOR..... | 572 |
| SB_SETICON..... | 572 |
| SB_SETMINHEIGHT..... | 573 |
| SB_SETPARTS..... | 574 |
| SB_SETTEXT..... | 574 |
| SB_SETTIPTEXT..... | 575 |
| SB_SETUNICODEFORMAT..... | 576 |
| SB_SIMPLE..... | 577 |
| SBN_SIMPLEMODECHANGE..... | 580 |
| SECOND_IPADDRESS..... | 328 |
| ShowHideMenuCtl..... | 85 |

T

| | |
|-------------------------------|-----|
| TabCtrl_AdjustRect..... | 619 |
| TabCtrl_DeleteAllItems..... | 620 |
| TabCtrl_Deleteltem..... | 620 |
| TabCtrl_DeselectAll..... | 621 |
| TabCtrl_GetCurFocus..... | 622 |
| TabCtrl_GetCurSel..... | 622 |
| TabCtrl_GetExtendedStyle..... | 623 |

| | | | |
|--------------------------------|-----|----------------------------|-----|
| TabCtrl_GetImageList | 623 | TCHITTESTINFO | 644 |
| TabCtrl_GetItem | 624 | TCITEM | 645 |
| TabCtrl_GetItemCount | 625 | TCITEMHEADER | 647 |
| TabCtrl_GetItemRect | 625 | TCM_ADJUSTRECT | 601 |
| TabCtrl_GetRowCount | 626 | TCM_DELETEALLITEMS | 601 |
| TabCtrl_GetToolTips | 627 | TCM_DELETEITEM | 602 |
| TabCtrl_GetUnicodeFormat | 627 | TCM_DESELECTALL | 602 |
| TabCtrl_HighlightItem | 628 | TCM_GETCURFOCUS | 603 |
| TabCtrl_HitTest | 629 | TCM_GETCURSEL | 604 |
| TabCtrl_InsertItem | 629 | TCM_GETEXTENDEDSTYLE | 604 |
| TabCtrl_RemoveImage | 630 | TCM_GETIMAGELIST | 605 |
| TabCtrl_SetCurFocus | 631 | TCM_GETITEM | 605 |
| TabCtrl_SetCurSel | 632 | TCM_GETITEMCOUNT | 606 |
| TabCtrl_SetExtendedStyle | 632 | TCM_GETITEMRECT | 606 |
| TabCtrl_SetImageList | 633 | TCM_GETROWCOUNT | 607 |
| TabCtrl_SetItem | 634 | TCM_GETTOOLTIPS | 607 |
| TabCtrl_SetItemExtra | 634 | TCM_GETUNICODEFORMAT | 608 |
| TabCtrl_SetItemSize | 635 | TCM_HIGHLIGHTITEM | 609 |
| TabCtrl_SetMinTabWidth | 636 | TCM_HITTEST | 609 |
| TabCtrl_SetPadding | 637 | TCM_INSERTITEM | 610 |
| TabCtrl_SetToolTips | 637 | TCM_REMOVEIMAGE | 611 |
| TabCtrl_SetUnicodeFormat | 638 | TCM_SETCURFOCUS | 611 |
| TBM_CLEARSEL | 705 | TCM_SETCURSEL | 612 |
| TBM_CLEARTEXT | 706 | TCM_SETEXTENDEDSTYLE | 613 |
| TBM_GETBUDDY | 706 | TCM_SETIMAGELIST | 614 |
| TBM_GETCHANNELRECT | 707 | TCM_SETITEM | 614 |
| TBM_GETLINESIZE | 708 | TCM_SETITEMEXTRA | 615 |
| TBM_GETNUMTICS | 708 | TCM_SETITEMSIZE | 616 |
| TBM_GETPAGESIZE | 709 | TCM_SETMINTABWIDTH | 616 |
| TBM_GETPOS | 710 | TCM_SETPADDING | 617 |
| TBM_GETPTICS | 710 | TCM_SETTOOLTIPS | 617 |
| TBM_GETRANGEMAX | 711 | TCM_SETUNICODEFORMAT | 618 |
| TBM_GETRANGEMIN | 711 | TCN_FOCUSCHANGE | 640 |
| TBM_GETSELEND | 712 | TCN_GETOBJECT | 641 |
| TBM_GETSELSTART | 713 | TCN_KEYDOWN | 642 |
| TBM_GETTHUMBLENGTH | 713 | TCN_SELCHANGE | 642 |
| TBM_GETTHUMBRECT | 714 | TCN_SELCHANGING | 643 |
| TBM_GETTIC | 715 | THIRD_IPADDRESS | 329 |
| TBM_GETTICPOS | 715 | TOOLINFO | 693 |
| TBM_GETTOOLTIPS | 716 | TTHITTESTINFO | 695 |
| TBM_GETUNICODEFORMAT | 716 | TTM_ACTIVATE | 666 |
| TBM_SETBUDDY | 717 | TTM_ADDTOOL | 666 |
| TBM_SETLINESIZE | 718 | TTM_ADJUSTRECT | 667 |
| TBM_SETPAGESIZE | 719 | TTM_DELTOOL | 668 |
| TBM_SETPOS | 719 | TTM_ENUMTOOLS | 669 |
| TBM_SETRANGE | 720 | TTM_GETBUBBLESIZE | 669 |
| TBM_SETRANGEMAX | 721 | TTM_GETCURRENTTOOL | 670 |
| TBM_SETRANGEMIN | 722 | TTM_GETDELAYTIME | 671 |
| TBM_SETSEL | 722 | TTM_GETMARGIN | 671 |
| TBM_SETSELEND | 723 | TTM_GETMAXTIPWIDTH | 672 |
| TBM_SETSELSTART | 724 | TTM_GETTEXT | 673 |
| TBM_SETTHUMBLENGTH | 725 | TTM_GETTIPBKCOLOR | 674 |
| TBM_SETTIC | 725 | TTM_GETTOOLCOUNT | 674 |
| TBM_SETTIPSIDE | 726 | TTM_GETTOOLINFO | 675 |
| TBM_SETTOOLTIPS | 727 | TTM_HITTEST | 675 |

| | |
|---------------------------|-----|
| TTM_NEWTOOLRECT | 676 |
| TTM_POP | 677 |
| TTM_RELAYEVENT | 677 |
| TTM_SETDELAYTIME | 678 |
| TTM_SETMARGIN | 679 |
| TTM_SETMAXTIPWIDTH | 680 |
| TTM_SETTIPBKCOLOR | 681 |
| TTM_SETTIPTEXTCOLOR | 681 |
| TTM_SETTITLE | 682 |
| TTM_SETTOOLINFO | 683 |
| TTM_TRACKACTIVATE | 683 |
| TTM_TRACKPOSITION | 684 |
| TTM_UPDATE | 685 |
| TTM_UPDATETIPTTEXT | 686 |
| TTM_WINDOWFROMPOINT | 686 |
| TTN_GETDISPINFO | 688 |
| TTN_POP | 689 |
| TTN_SHOW | 690 |

U

| | |
|--------------------|-----|
| UDACCEL | 748 |
| UDM_GETACCEL | 737 |
| UDM_GETBASE | 738 |

| | |
|----------------------------|-----|
| UDM_GETBUDDY | 738 |
| UDM_GETPOS | 738 |
| UDM_GETRANGE | 739 |
| UDM_GETRANGE32 | 740 |
| UDM_GETUNICODEFORMAT | 740 |
| UDM_SETACCEL | 741 |
| UDM_SETBASE | 742 |
| UDM_SETBUDDY | 742 |
| UDM_SETPOS | 743 |
| UDM_SETRANGE | 743 |
| UDM_SETRANGE32 | 744 |
| UDM_SETUNICODEFORMAT | 745 |
| UDN_DELTAPOS | 746 |
| UninitializeFlatSB | 249 |

W

| | |
|-----------------------|----|
| WM_NOTIFY | 90 |
| WM_NOTIFYFORMAT | 91 |

APPENDIX B

Index B: Volume 5, Elements Listed Alphabetically

A

| | |
|-----------------------------|-----|
| ABM_ACTIVATE | 721 |
| ABM_GETAUTOHIDEBAR | 721 |
| ABM_GETSTATE | 722 |
| ABM_GETTASKBARPOS | 723 |
| ABM_NEW | 723 |
| ABM_QUERYPOS | 724 |
| ABM_REMOVE | 724 |
| ABM_SETAUTOHIDEBAR | 725 |
| ABM_SETPOS | 726 |
| ABM_WINDOWPOSCHANGED | 726 |
| ABN_FULLSCREENAPP | 727 |
| ABN_POSCHANGED | 728 |
| ABN_STATECHANGE | 728 |
| ABN_WINDOWARRANGE | 729 |
| AssocCreate | 670 |
| ASSOCDATA | 561 |
| ASSOCF | 561 |
| ASSOCKEY | 563 |
| AssocQueryKey | 671 |
| AssocQueryString | 672 |
| AssocQueryStringByKey | 674 |
| ASSOCSTR | 563 |

B

| | |
|--------------------------|-----|
| BrowseCallbackProc | 481 |
|--------------------------|-----|

C

| | |
|-----------------------|-----|
| ChrCmpl | 575 |
| ColorAdjustLuma | 707 |
| ColorHLSToRGB | 708 |
| ColorRGBToHLS | 708 |
| CPL_DBLCLK | 730 |
| CPL_EXIT | 730 |
| CPL_GETCOUNT | 731 |
| CPL_INIT | 732 |
| CPL_INQUIRE | 732 |
| CPL_NEWINQUIRE | 733 |
| CPL_STARTWPARMS | 735 |

| | |
|-----------------|-----|
| CPL_STOP | 736 |
| CPIApplet | 409 |

D

| | |
|--------------------------|-----|
| DefScreenSaverProc | 410 |
| DllGetVersion | 411 |
| DLLGETVERSIONPROC | 412 |
| DllInstall | 710 |
| DoEnvironmentSubst | 413 |
| DragAcceptFiles | 414 |
| DragFinish | 415 |
| DragQueryFile | 416 |
| DragQueryPoint | 417 |

F

| | |
|-----------------------------|-----|
| FindEnvironmentString | 418 |
| FindExecutable | 419 |
| FM_GETDRIVEINFO | 736 |
| FM_GETFILESEL | 737 |
| FM_GETFILESELLFN | 738 |
| FM_GETFOCUS | 739 |
| FM_GETSELCOUNT | 739 |
| FM_GETSELCOUNTLFN | 740 |
| FM_REFRESH_WINDOWS | 740 |
| FM_RELOAD_EXTENSIONS | 741 |
| FMEVENT_HELPMENUITEM | 742 |
| FMEVENT_HELPSTRING | 742 |
| FMEVENT_INITMENU | 743 |
| FMEVENT_LOAD | 744 |
| FMEVENT_SELCHANGE | 745 |
| FMEVENT_TOOLBARLOAD | 745 |
| FMEVENT_UNLOAD | 746 |
| FMEVENT_USER_REFRESH | 746 |
| FMExtensionProc | 483 |
| FOLDERFLAGS | 564 |
| FOLDERVIEWMODE | 566 |

G

| | |
|------------------------------|-----|
| GetMenuContextHelpId | 420 |
| GetWindowContextHelpId | 420 |

H

HashData..... 711

I

IACList
 Expand 141

IACList2
 GetOptions 143
 SetOptions..... 143

IActiveDesktop
 AddDesktopItem Method..... 145
 AddDesktopItemWithUI Method..... 146
 AddUrl Method 148
 ApplyChanges 149
 GenerateDesktopItemHtml..... 150
 GetDesktopItem 150
 GetDesktopItemByID..... 151
 GetDesktopItemBySource..... 152
 GetPattern 153
 GetDesktopItemCount..... 153
 GetDesktopItemOptions 154
 GetWallpaper 154
 GetWallpaperOptions 155
 ModifyDesktopItem..... 156
 RemoveDesktopItem..... 157
 SetDesktopItemOptions 157
 SetPattern..... 158
 SetWallpaper 159
 SetWallpaperOptions 159

IASyncOperation
 EndOperation 161
 GetAsyncMode 162
 InOperation..... 163
 SetAsyncMode 163
 StartOperation 164

IAutoComplete
 Enable 167
 Init..... 168

IAutoComplete2
 GetOptions 170
 SetOptions..... 171

IColumnProvider
 GetColumnInfo 174
 GetItemData 175
 Initialize..... 176

ICommDlgBrowser
 IncludeObject 177
 OnDefaultCommand..... 178
 OnStateChange..... 178

ICommDlgBrowser2
 GetDefaultMenuText 180
 GetViewFlags 181
 Notify 182

IContextMenu
 GetCommandString 183
 InvokeCommand..... 185
 QueryContextMenu 186

IContextMenu2
 HandleMenuMsg..... 189

IContextMenu3
 HandleMenuMsg2..... 191

ICopyHook
 CopyCallback..... 193

ICurrentWorkingDirectory
 GetDirectory 195
 SetDirectory 196

IDeskBand
 GetBandInfo..... 197

IDockingWindow
 CloseDW 199
 ResizeBorderDW 199
 ShowDW 201

IDockingWindowFrame
 AddToolbar 202
 FindToolbar 203
 RemoveToolbar 204

IDockingWindowSite
 GetBorderDW 214
 RequestBorderSpaceDW 215
 SetBorderSpaceDW 215

IDragSourceHelper
 InitializeFromBitmap 206
 InitializeFromWindow..... 207

IDropTargetHelper
 DragEnter..... 209
 DragLeave 210
 DragOver 210
 Drop 211
 Show 212

IEmptyVolumeCache
 Deactivate 217
 GetSpaceUsed..... 218
 Initialize 219
 Purge 221
 ShowProperties..... 222

IEmptyVolumeCache2
 InitializeEx..... 224

IEmptyVolumeCacheCallback
 PurgeProgress..... 227
 ScanProgress 228

IEnumExtraSearch
 Clone..... 229
 Next..... 230
 Reset..... 231
 Skip 231

IEnumIDList
 Clone..... 233
 Next..... 233

| | | | |
|------------------------------|-----|------------------------------|-----|
| Reset | 235 | StopProgressDialog | 276 |
| Skip | 235 | Timer | 276 |
| IExtractIcon | | IQueryAssociations | |
| Extract | 237 | GetData | 279 |
| GetIconLocation | 238 | GetEnum | 280 |
| IExtractImage | | GetKey | 280 |
| Extract | 241 | GetString | 281 |
| GetLocation | 241 | Init | 282 |
| IExtractImage2 | | IQueryInfo | |
| GetDateStamp | 244 | GetInfoFlags | 284 |
| IFileViewer | | GetInfoTip | 285 |
| PrintTo | 245 | IReconcilableObject | |
| Show | 246 | GetProgressFeedbackMax | |
| ShowInitialize | 247 | Estimate | 286 |
| IFileViewerSite | | Reconcile | 287 |
| GetPinnedWindow | 248 | IReconcileInitiator | |
| SetPinnedWindow | 249 | SetAbortCallback | 292 |
| IInputObject | | SetProgressFeedback | 293 |
| HasFocusIO | 250 | IRemoteComputer | |
| TranslateAcceleratorIO | 251 | Initialize | 294 |
| UIActivateIO | 251 | IResolveShellLink | |
| IInputObjectSite | | ResolveShellLink | 296 |
| OnFocusChangeIS | 253 | IRunnableTask | |
| InetIsOffline | 421 | IsRunning | 298 |
| INewShortcutHook | | Kill | 299 |
| GetExtension | 254 | Resume | 299 |
| GetFolder | 255 | Run | 300 |
| GetName | 256 | Suspend | 300 |
| GetReferent | 256 | IShellBrowser | |
| SetFolder | 257 | BrowseObject | 302 |
| SetReferent | 258 | EnableModelessSB | 304 |
| INotifyReplica | | GetControlWindow | 304 |
| YouAreAReplica | 259 | GetViewStateStream | 306 |
| IntlStrEqN | 576 | InsertMenusSB | 307 |
| IntlStrEqNI | 577 | OnViewWindowActive | 308 |
| IntlStrEqWorker | 578 | QueryActiveShellView | 309 |
| IObjMgr | | RemoveMenusSB | 310 |
| Append | 260 | SendControlMsg | 311 |
| Remove | 261 | SetMenuSB | 312 |
| IPersistFileSystemFolder | | SetStatusTextSB | 313 |
| GetFolderTargetInfo | 265 | SetToolBarItems | 314 |
| InitializeEx | 266 | TranslateAcceleratorSB | 315 |
| IPersistFolder | | IShellChangeNotify | |
| Initialize | 262 | OnChange | 316 |
| IPersistFolder2 | | IShellDetails | |
| GetCurFolder | 263 | ColumnClick | 319 |
| IProgressDialog | | GetDetailsOf | 320 |
| HasUserCancelled | 269 | IShellExecuteHook | |
| SetAnimation | 269 | Execute | 323 |
| SetCancelMsg | 270 | IShellExtInit | |
| SetLine | 271 | Initialize | 324 |
| SetProgress | 272 | IShellFolder | |
| SetProgress64 | 273 | BindToObject | 327 |
| SetTitle | 274 | BindToStorage | 328 |
| StartProgressDialog | 274 | CompareIDs | 329 |

| | | | |
|-----------------------------|-----|-----------------------------|-----|
| CreateViewObject | 331 | DestroyViewWindow | 387 |
| EnumObjects | 332 | EnableModeless | 387 |
| GetAttributesOf | 333 | EnableModelessSV | 388 |
| GetDisplayNameOf | 335 | GetCurrentInfo | 388 |
| GetUIObjectOf | 337 | GetItemObject | 389 |
| ParseDisplayName | 338 | Refresh | 390 |
| SetNameOf | 342 | SaveViewState | 391 |
| IShellFolder2 | | SelectItem | 392 |
| EnumSearches | 344 | TranslateAccelerator | 393 |
| GetDefaultColumn | 345 | UIActivate | 394 |
| GetDefaultColumnState | 346 | IShellView2 | |
| GetDefaultSearchGUID | 347 | CreateViewWindow2 | 396 |
| GetDetailsEx | 347 | GetView | 397 |
| GetDetailsOf | 348 | HandleRename | 397 |
| MapNameToSCID | 349 | SelectAndPositionItem | 398 |
| IShellIcon | | ITaskbarList | |
| GetIconOf | 351 | ActivateTab | 400 |
| IShellIconOverlay | | AddTab | 400 |
| GetOverlayIconIndex | 353 | DeleteTab | 401 |
| GetOverlayIndex | 354 | HrInit | 402 |
| IShellIconOverlayIdentifier | | SetActiveAlt | 402 |
| GetOverlayInfo | 356 | IUniformResourceLocator | |
| GetPriority | 357 | GetURL | 403 |
| IsMemberOf | 358 | InvokeCommand | 405 |
| IShellLink | | SetURL | 406 |
| GetArguments | 360 | IURL_SETURL_FLAGS | 566 |
| GetDescription | 361 | IURL_SETURL_INVOKECOMMAND_ | |
| GetHotkey | 361 | FLAGS | 567 |
| GetIconLocation | 362 | IURLSearchHook | |
| GetIDList | 363 | Translate | 407 |
| GetPath | 364 | | |
| GetShowCmd | 365 | | |
| GetWorkingDirectory | 366 | | |
| Resolve | 366 | | |
| SetArguments | 368 | | |
| SetDescription | 369 | | |
| SetHotkey | 370 | | |
| SetIconLocation | 371 | | |
| SetIDList | 371 | | |
| SetPath | 372 | | |
| SetRelativePath | 373 | | |
| SetShowCmd | 374 | | |
| SetWorkingDirectory | 375 | | |
| IShellLinkDataList | | | |
| AddDataBlock | 376 | | |
| CopyDataBlock | 377 | | |
| GetFlags | 378 | | |
| RemoveDataBlock | 379 | | |
| SetFlags | 379 | | |
| IShellPropSheetExt | | | |
| AddPages | 381 | | |
| ReplacePage | 382 | | |
| IShellView | | | |
| AddPropertySheetPages | 384 | | |
| CreateViewWindow | 385 | | |

M

| | |
|-----------------------------|-----|
| MAKEDLLVERULL | 571 |
| MIMEAssociationDialog | 421 |
| MLLoadLibrary | 579 |

P

| | |
|-----------------------------|-----|
| PathAddBackslash | 610 |
| PathAddExtension | 610 |
| PathAppend | 611 |
| PathBuildRoot | 612 |
| PathCanonicalize | 613 |
| PathCombine | 614 |
| PathCommonPrefix | 615 |
| PathCompactPath | 615 |
| PathCompactPathEx | 616 |
| PathCreateFromUrl | 617 |
| PathFileExists | 618 |
| PathFindExtension | 619 |
| PathFindFileName | 620 |
| PathFindNextComponent | 620 |
| PathFindOnPath | 621 |

| | | | |
|----------------------------------|-----|----------------------------------|-----|
| PathFindSuffixArray | 622 | SHAppBarMessage | 429 |
| PathGetArgs | 623 | SHAutoComplete | 712 |
| PathGetCharType | 623 | SHBindToParent | 430 |
| PathGetDriveNumber | 624 | SHBrowseForFolder | 431 |
| PathsContentType | 625 | SHChangeNotify | 432 |
| PathsDirectory | 625 | SHCONTF | 568 |
| PathsDirectoryEmpty | 626 | SHCopyKey | 675 |
| PathsFileSpec | 627 | SHCreateDirectoryEx | 437 |
| PathsHTMLFile | 627 | SHCreateProcessAsUser | 438 |
| PathsLFNFileSpec | 628 | SHCreateShellPalette | 709 |
| PathsNetworkPath | 629 | SHCreateStreamOnFile | 714 |
| PathsPrefix | 630 | SHCreateThread | 714 |
| PathsRelative | 630 | SHDeleteEmptyKey | 676 |
| PathsRoot | 631 | SHDeleteKey | 677 |
| PathsSameRoot | 632 | SHDeleteValue | 678 |
| PathsSystemFolder | 632 | Shell_NotifyIcon | 439 |
| PathsUNC | 633 | ShellAbout | 441 |
| PathsUNCServer | 634 | ShellExecute | 442 |
| PathsUNCServerShare | 634 | ShellExecuteEx | 445 |
| PathsURL | 635 | SHEmptyRecycleBin | 447 |
| PathMakePretty | 636 | SHEnumKeyEx | 679 |
| PathMakeSystemFolder | 636 | SHEnumValue | 680 |
| PathMatchSpec | 637 | SHFileOperation | 448 |
| PathParseIconLocation | 638 | SHFreeNameMappings | 449 |
| PathQuoteSpaces | 639 | SHGetDataFromIDLList | 450 |
| PathRelativePathTo | 639 | SHGetDesktopFolder | 451 |
| PathRemoveArgs | 641 | SHGetDiskFreeSpace | 452 |
| PathRemoveBackslash | 641 | SHGetFileInfo | 453 |
| PathRemoveBlanks | 642 | SHGetFolderLocation | 457 |
| PathRemoveExtension | 642 | SHGetFolderPath | 458 |
| PathRemoveFileSpec | 643 | SHGetIconOverlayIndex | 461 |
| PathRenameExtension | 644 | SHGetInstanceExplorer | 462 |
| PathSearchAndQualify | 644 | SHGetMalloc | 463 |
| PathSetDlgItemPath | 645 | SHGetNewLinkInfo | 464 |
| PathSkipRoot | 646 | SHGetPathFromIDLList | 466 |
| PathStripPath | 647 | SHGetSettings | 466 |
| PathStripToRoot | 647 | SHGetSpecialFolderLocation | 468 |
| PathUndecorate | 648 | SHGetSpecialFolderPath | 469 |
| PathUnExpandEnvStrings | 649 | SHGetThreadRef | 716 |
| PathUnmakeSystemFolder | 650 | SHGetValue | 681 |
| PathUnquoteSpaces | 651 | SHGNO | 569 |
| | | SHInvokePrinterCommand | 470 |
| | | SHLoadInProc | 472 |
| | | SHOpenRegStream | 717 |
| | | SHOpenRegStream2 | 718 |
| | | SHQueryInfoKey | 683 |
| | | SHQueryRecycleBin | 473 |
| | | SHQueryValueEx | 684 |
| | | SHRegCloseUSKey | 685 |
| | | SHRegCreateUSKey | 686 |
| | | SHREGDEL_FLAGS | 705 |
| | | SHRegDeleteEmptyUSKey | 687 |
| | | SHRegDeleteUSValue | 688 |
| | | SHRegDuplicateHKey | 689 |
| | | SHREGENUM_FLAGS | 706 |
| R | | | |
| RegisterDialogClasses | 423 | | |
| REGSAM | 669 | | |
| S | | | |
| ScreenSaverConfigureDialog | 424 | | |
| ScreenSaverProc | 425 | | |
| SetMenuContextHelpId | 426 | | |
| SetWindowContextHelpId | 427 | | |
| SHAddToRecentDocs | 428 | | |

| | | | |
|--------------------------------|-----|-------------------------------------|-----|
| SHRegEnumUSKey | 690 | StrRStrl | 602 |
| SHRegEnumUSValue | 691 | StrSpn | 603 |
| SHRegGetBoolUSValue | 692 | StrStr | 604 |
| SHRegGetPath | 693 | StrStrl | 604 |
| SHRegGetUSValue | 694 | StrToInt | 605 |
| SHRegOpenUSKey | 696 | StrToIntEx | 606 |
| SHRegQueryInfoUSKey | 697 | StrTrim | 607 |
| SHRegQueryUSValue | 698 | | |
| SHRegSetPath | 700 | T | |
| SHRegSetUSValue | 701 | TranslateURL | 475 |
| SHRegWriteUSValue | 702 | TRANSLATEURL_IN_FLAGS | 570 |
| SHSetThreadRef | 719 | | |
| SHSetValue | 704 | U | |
| SHStrDup | 580 | UndeleteFile | 484 |
| SOANGLETENTHS | 573 | UrlApplyScheme | 651 |
| SoftwareUpdateMessageBox | 473 | URLAssociationDialog | 476 |
| SOPALETTEINDEX | 573 | URLASSOCIATIONDIALOG_IN_FLAGS | 571 |
| SOPALETTERGB | 573 | UrlCanonicalize | 653 |
| SORGB | 574 | UrlCombine | 654 |
| SOSETRATIO | 574 | UrlCompare | 655 |
| StrCat | 581 | UrlCreateFromPath | 656 |
| StrCatBuff | 581 | UrlEscape | 657 |
| StrChr | 582 | UrlEscapeSpaces | 658 |
| StrChrl | 583 | UrlGetLocation | 659 |
| StrCmp | 584 | UrlGetPart | 660 |
| StrCmpl | 585 | UrlHash | 661 |
| StrCmpN | 585 | Urlls | 662 |
| StrCmpNI | 586 | UrllsFileUrl | 663 |
| StrCpy | 587 | UrllsNoHistory | 664 |
| StrCpyN | 588 | UrllsOpaque | 665 |
| StrCSpn | 589 | UrlUnEscape | 666 |
| StrCSpnl | 590 | UrlUnEscapeInPlace | 667 |
| StrDup | 591 | | |
| StrFormatByteSize | 592 | W | |
| StrFormatByteSize64A | 593 | WinHelp | 477 |
| StrFormatKBSize | 594 | WM_CPL_LAUNCH | 747 |
| StrFromTimeInterval | 595 | WM_CPL_LAUNCHED | 747 |
| StrIsIntlEqual | 596 | WM_DROPFILES | 748 |
| StrNCat | 597 | WM_HELP | 749 |
| StrPBrk | 598 | WM_TCARD | 749 |
| StrRChr | 598 | wnsprintf | 608 |
| StrRChrl | 599 | wvnsprintf | 609 |
| StrRetToBuf | 600 | | |
| StrRetToStr | 601 | | |

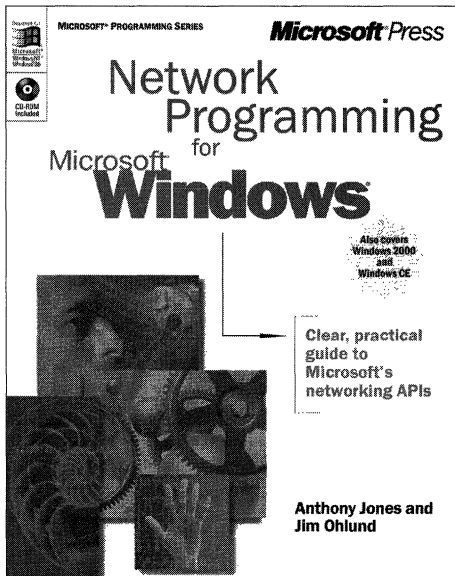


Here they are in one place—

practical, detailed explanations

of the Microsoft

networking APIs!



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).

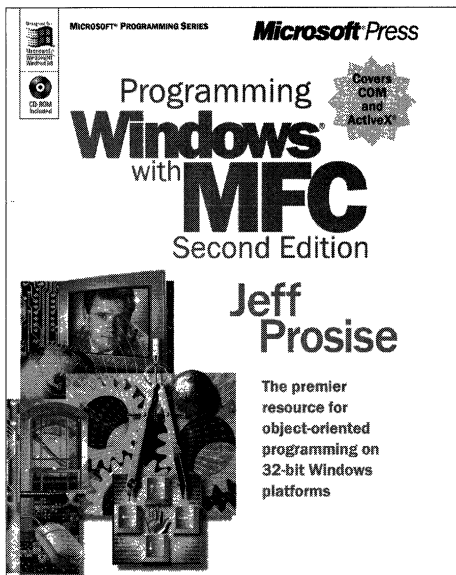
U.S.A. \$49.99
U.K. £46.99 [V.A.T. included]
Canada \$74.99
ISBN 0-7356-0560-2

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

Petzold for the MFC programmer!



U.S.A. **\$59.99**
U.K. £56.99 [V.A.T. included]
Canada \$89.99
ISBN 1-57231-695-0

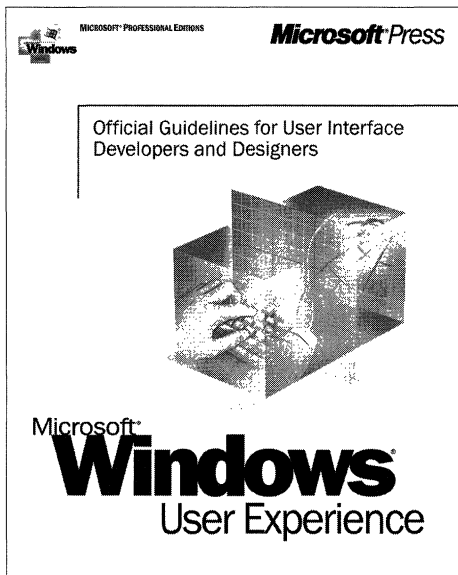
Expanding what's widely considered the definitive exposition of Microsoft's powerful C++ class library for the Windows API, PROGRAMMING WINDOWS® WITH MFC, Second Edition, fully updates the classic original with all-new coverage of COM, OLE, and ActiveX.® Author Jeff Prosise deftly builds your comprehension 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

Official Guidelines for User Interface Developers and Designers



U.S.A. **\$49.99**
U.K. £46.99 [V.A.T. included]
Canada \$74.99
ISBN 0-7356-0566-1

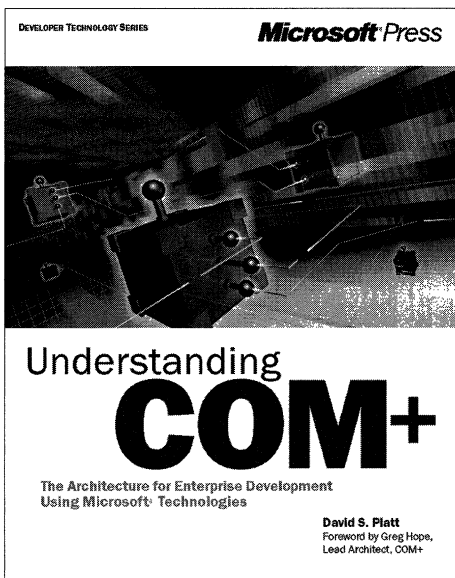
Here are the revised, updated, official Microsoft guidelines for creating well-designed, visually and functionally consistent user interfaces for applications that run on the Microsoft Windows family of operating systems, including Windows 98 and Windows 2000. A revision of *The Windows Interface Guidelines for Software Design*, the standard resource for designing Windows interfaces, MICROSOFT WINDOWS USER EXPERIENCE is an essential handbook for all programmers and designers who work with the latest releases of Windows and Microsoft Internet Explorer, regardless of experience level or development tools used. It covers the basic principles of user-interface design and methodologies, and it specifies how you can apply data-centered concepts such as objects and properties to interface design. The book includes detailed information on mouse, keyboard, and other input-device interaction and on how to use the common interface elements supplied by the system. It also includes information about supporting international and disabled users.

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

Learn how
COM+
can simplify your
development tasks



Wouldn't it be great to have an enterprise application's infrastructure so that you could inherit what you need and spend your time writing your own business logic? COM+ is what you've been waiting for—an advanced development environment that provides prefabricated solutions to common enterprise application problems. UNDERSTANDING COM+ is a succinct, entertaining book that offers an overview of COM+ and key COM+ features, explains the role of COM+ in enterprise development, and describes the services it can provide for your components and clients. You'll learn how COM+ can streamline application development to help you get enterprise applications up and running and out the door.

U.S.A. **\$24.99**
U.K. £22.99
Canada \$37.99
ISBN 0-7356-0666-8

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® **Windows** User Interface



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 UI

This volume provides complete reference materials about Windows User Interface programming elements such as buttons, edit and static controls, combo and list boxes, and scroll bars. It also includes information about resources such as carets, cursors, icons, menus, and strings; user input via mouse and keyboard, keyboard accelerators, and the Common Dialog Box Library; and windowing dialog boxes, messages and message queues, the Multiple-Document Interface, timers, window classes, procedures, properties, and more.