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

The essential reference to Win32® technologies and APIs

David Iseminger
Series Editor
www.iseminger.com

Microsoft® Windows® Base Services

BASED ON
msdn™ library

The essential reference to Win32® technologies and APIs

David Iseminger
Series Editor

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.

Intel is a registered trademark of Intel Corporation. BackOffice, FrontPage, Microsoft, Microsoft Press, MSDN, MS-DOS, 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. 097-0002306

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 1
How the Win32 Library Is Designed 3

Chapter 2: What's in This Volume? 5

Processes and Threads 5
Memory 6
Interprocess Communications 7
File Operations 8
Debugging 9
Unicode 9

Chapter 3: Using Microsoft Reference Resources 11

The Microsoft Developer Network (MSDN) 12
Comparing MSDN and MSDN Online 12
MSDN Subscriptions 13
MSDN Library Subscription 15
MSDN Professional Subscription 15
MSDN Universal Subscription 16
Purchasing an MSDN Subscription 16
Using MSDN 17
Exploring MSDN 18
Quick Tips 21
Using MSDN Online 21
Exploring MSDN Online 23
MSDN Online Features 24
MSDN Online Registered Users 28
The Windows Programming Reference Series 29

Chapter 4: Finding the Developer Resources You Need 31

Developer Support 31
Online Resources 33
Learning Products 34
Conferences 36
Other Resources 37

Chapter 5: Getting the Most Out of Win32 Technologies: Part 1 39
Overview 39
Volume 2: User Interface 39
Volume 3: GDI 40
Volume 4: Common Controls 40
Volume 5: The Windows Shell 40
Solution Summary 40

Chapter 6: Processes, Threads, and DLLs 45
Processes and Threads 45
About Processes and Threads 45
Multitasking 45
Scheduling 48
Multiple Threads 54
Child Processes 61
Process Working Set 69
Thread Pooling 70
Job Objects 71
Fibers 73
Process and Thread Reference 74
Process and Thread Functions 74
Process and Thread Structures 184
Process and Thread Macros 207
Dynamic Link Libraries 208
About Dynamic Link Libraries 208
Advantages of Dynamic Linking 209
Dynamic Link Library Creation 210
Dynamic Link Library Entry-Point Function 211
Load-Time Dynamic Linking 213
Run-Time Dynamic Linking 214
Dynamic Link Library Data 215
Dynamic Link Library Redirection 216
Dynamic Link Library Updates 216
Dynamic Link Library Reference 217
Dynamic Link Library Functions 217
Synchronization 234
Getting More Information About Synchronization 235
About Synchronization 235
Wait Functions 235

Synchronization Objects ... 237
Interprocess Synchronization ... 243
Synchronization Object Security and Access Rights ... 244
Synchronization and Overlapped Input and Output ... 246
Asynchronous Procedure Calls ... 247
Critical Section Objects ... 248
Interlocked Variable Access ... 249

Chapter 7: Memory Management ... 251

About Memory Management ... 251
Virtual Address Space ... 251
Virtual Address Space and Physical Storage ... 252
Page State ... 252
Scope of Allocated Memory ... 253
Virtual Memory Functions ... 253
Allocating Virtual Memory ... 254
Freeing Virtual Memory ... 255
Working with Pages ... 255
Heap Functions ... 256
Access Validation Functions ... 257
Address Windowing Extensions ... 258
AWE Functions ... 259
Global and Local Functions ... 260
Standard C Library Functions ... 261
Memory Management Reference ... 261
Memory Management Structures ... 328

Chapter 8: Interprocess Communications ... 337

Interprocess Communications ... 337
About Interprocess Communications ... 337
Choosing an IPC Mechanism ... 338
Using the Clipboard for IPC ... 338
Using COM for IPC ... 339
Using DDE for IPC ... 339
Using a File Mapping for IPC ... 340
Using a Mailslot for IPC ... 340
Using Pipes for IPC ... 341
Using RPC for IPC ... 341
Using Windows Sockets for IPC ... 342
Using WM_COPYDATA for IPC ... 342

Interprocess Communications Reference 343
Interprocess Communications Structures 343
Interprocess Communications Messages 343
Atoms 345
About Atom Tables 345
Global Atom Tables 345
Local Atom Tables 345
Atom Types 346
Atom Reference 346
Atom Functions 346
Atom Macros 356
Clipboard 356
About the Clipboard 357
Clipboard Formats 357
Clipboard Reference 363
Clipboard Functions 363
Clipboard Structures 378
Clipboard Messages 380
Handles and Objects 393
About Handles and Objects 393
Object Manager 394
Object Interface 394
Handle Limitations 395
Handle Inheritance 395
Object Categories 396
User Objects 397
GDI Objects 399
Kernel Objects 400
Handle and Object Reference 404
Handle and Object Functions 404
Hooks 415
About Hooks 415
Hook Chains 415
Hook Procedures 416
Hook Types 416
Hook Reference 420
Hook Functions 420
Hook Structures 456
Hook Messages 465

Chapter 9: File I/O 469

About File I/O 469
File System Organization 469
Accessing Files 469
File Name Conventions 470
Long File Names 471
MS-DOS Device Names 471
File Operations 471
Creating and Opening Files with the CreateFile Function 471
Creating Temporary Files 472
Copying and Moving Files 472
Reading from and Writing to a File 473
Locking and Unlocking Files 474
Searching for Files 474
Monitoring Directories 474
Closing and Deleting Files 475
Directory Operations 475
Asynchronous Input and Output 476
I/O Completion Ports 476
Getting Information About Files 477
File Attributes 477
File Type 479
File Date and Time 479
File Code Page 479
Volume Information 479
File and Directory Security 480
File I/O Reference 481
File I/O Functions 481
File I/O Structures 606
File I/O Enumeration Types 617

Chapter 10: File Systems 621

About File Systems 621
Shared File System Features 621
Opportunistic Locks 621
Alternatives to Opportunistic Lock Operations 622
Local Caching 622
Data Coherency 623
Typical Use 624

Server Response to Open Requests on Locked Files 624
Types of Opportunistic Locks 625
Breaking Opportunistic Locks 627
Opportunistic Lock Examples 628
Opportunistic Lock Operations 631
NTFS File System 632
File System Recovery 632
File Compression 633
Compression Attribute 633
Compression State 633
Obtaining the Size of a Compressed File 634
File Encryption 634
Handling Encrypted Files and Directories 634
Encrypted Files and User Keys 635
Disk Quotas 635
Disk Quota Limits 635
Disk Quota States 636
Administering Disk Quotas 636
Sparse Files 636
Sparse File Operations 637
Obtaining the Size of a Sparse File 638
Sparse Files and Disk Quota 638
Distributed Link Tracking 638
Link Tracking Features 638
Link Tracking Components 639
Link Tracking Limitations 640
Reparse Points 640
Reparse Point Tags 641
Reparse Point Operations 642
Reparse Points and File Operations 643
Reparse Point Restrictions 643
Volume Mount Points and Mounting Volumes 643
Unique Volume Names 645
Path Lengths 645
Mounting a Volume 646
Enumerating Volumes 646
Scanning Volume Mount Points on a Volume 646
Checking Directories for Volume Mount Points 647
Persistent Assignment of Drive Letters 648

Volume Mount Point Reference 648
NTFS Change Journal 649
Using the Change Journal Identifier 651
Creating, Modifying, and Deleting a Change Journal 652
Obtaining a Volume Handle for Change Journal Operations 653
Walking a Buffer of Change Journal Records 653
Change Journal Operations 654
Change Journal Structures 654
FAT File System 654
Protected-Mode FAT File System 655
File System Reference 655
File System Functions 655
File System Control Codes 682
File System Interfaces 683
File System Structures 731
File System Macros 736
Disk Quota Interface Error Codes 738

Chapter 11: Structured Exception and Error Handling 741

Structured Exception Handling 741
About Structured Exception Handling 741
Exception Handling 742
Frame-Based Exception Handling 744
Termination Handling 746
Handler Syntax 746
Structured Exception Handling Reference 750
Structured Exception Handling Structures 759
Error Handling 762
About Error Handling 762
Process Error Mode 763
Last-Error Code 763
Notifying the User 763
Message Tables 763
Fatal Application Exit 764
Error Message Guidelines 764
Error Handling Reference 767
Error Handling Functions 767
Error Handling Structures 783

Chapter 12: Unicode **785**
About Unicode and Character Sets 785
Character Sets 785
Single-Byte Character Sets 785
Double-Byte Character Sets 786
Unicode 786
Surrogates 787
Unicode in the Win32 API 789
Win32 Data Types 789
Win32 Function Prototypes 790
Message Translation 791
String Functions 792
Standard C Functions 793
Character Sets Used in Filenames 794
Translation Between String Types 794
Command-Line Arguments 795
Unicode and Character Set Reference 795
Unicode and Character Set Functions 795
Unicode and Character Set Structures 810
Unicode and Character Set Macros 812
Unicode and Character Set Constants 813
ANSI Code-Page Identifiers 813
OEM Code-Page Identifiers 813
Code-Page Identifiers 814
Code-Page Bitfields 816
Unicode Subset Bitfields 817

Appendix A **821**

Appendix B **829**

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 easily find the particular programming element you're looking for. This directory suite, which covers all the functions, structures, enumerations, and other programming elements found in Win32, gets you quickly to the volume and page you need, and also provides an overview of Microsoft technologies that would otherwise take you hours of time, reams of paper, and potfuls of coffee to compile yourself.

How the Win32 Library Is Structured

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

Volume 1: Base Services

Volume 2: User Interface

Volume 3: GDI (Graphical Device Interface)

Volume 4: Common Controls

Volume 5: The Windows Shell

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

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

Part 1: Introduction and Overview

Part 2: Reference

Part 3: Windows Programming Directory

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

Part 2 contains the Win32 reference material particular to its volume, but it is *much* more than a simple collection of function and structure definitions. Because a comprehensive reference resource should include information about *how to use* a particular technology, as well as its definitions of programming elements, the information in Part 2 combines complete programming element definitions as well as instructional and 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 quickly find 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 feel that is 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 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 immediately applied to MSDN or MSDN Online, and vice-versa. In a word, it means *consistency*.

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

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

CHAPTER 2

What's in This Volume?

Each volume in the *Microsoft Win32 Developer's Reference Library* contains reference material that pertains to a certain area of the Win32 programming environment. This volume, *Volume 1: Base Services*, contains the bulk of reference and overview material (not to mention the insider tips and tricks) that developers need to establish the programmatic foundation for their applications. But, what does that mean, really?

It means that this volume provides developers with access to the operating system, and to computer resources, that are the building blocks on which the rest of the application can run. Operations such as memory access and management, processes and thread manipulation, synchronization, file operations, Unicode issues, and interprocess communications all fall under the base services umbrella. To put these concepts into a format that's a little easier to pick through, here are the sections covered in this volume:

Processes and Threads

Memory

Interprocess Communications

File Operations

Debugging

Unicode

Registry

Putting this information into nice, neat categories such as these doesn't make it well explained. In an effort to get you up to speed with the overall meaning behind such logical grouping, let's look at each of these categories in a little more depth, so you can get familiar with them quickly, in case you don't have decades of Windows programming experience.

Processes and Threads

Regardless of the type of application you're developing, you'll be dealing with processes and threads. Every Windows program consists of at least one process (and every process has at least one thread, the first of which is generally called the primary thread). A process is essentially an executing program, while a thread is a unit of execution within a process; each thread is allocated processing time individually of other threads in a given process. Another concept introduced with Windows 2000 is the job object, which allows multiple processes to be managed as a unit.

Processes and threads determine how your code is executed and how code within your application receives processor time. For projects that contain multiple processes, the Job object enables such multiple processes to be managed as a unit.

Within the Processes and Threads category, there are a number of associated base areas that are covered. The following list outlines the base programming areas associated with the Processes and Threads section:

Processes and Threads

Dynamic-Link Libraries

Synchronization

Processes and Threads have already been explained, so more discussion on them isn't necessary.

Dynamic-Link Libraries are commonly called DLLs, and provide a means by which functions and/or data can be developed and packaged in a modular format. Windows itself makes heavy use of DLLs, and provides most of its application programming interfaces to developers in the form of DLLs.

Synchronization enables multiple threads of execution within a given process to coordinate efforts, data, or most importantly, resource access. Synchronization makes use of objects to enable the synchronization among and between threads, including event, mutex, semaphore, process, and thread objects.

Memory

Every program that operates on the Windows platform must deal with memory, and generally speaking, the better an application handles its memory usage and memory management, the better off the application is. On the Windows 2000 platform, every application (that is, every process) is provided with its own virtual address space of 4 GB; the direct result of a 32-bit operating system, which has a 32-bit pointer (0x00000000 through 0xFFFFFFFF = 4 GB of possible values).

The logistics of mapping actual memory (that is, physical memory) to the virtual memory available to each process (the 4 GB worth of addressable memory) is left up to the operating system. Also, the memory space of any given process is protected by the operating system, enabling a process to be assured of its memory's privacy (protection from corruption) from other well-behaving processes. However, the logistics involved with managing an application's own memory is the responsibility of each application. The Memory section provides the programmatic functions and guidance to enable developers to program such memory issues, and includes the following sections:

Memory Management

File Mapping

Memory Management enables developers to supervise and administer the virtual address space available to their application. Memory management includes such tasks as allocating and freeing memory, and working with pages and heaps.

File Mapping is the process of associating a given file's contents with a particular area of a process's virtual memory, using a file-mapping object to maintain the association.

Interprocess Communications

In order to enable one application to communicate with another application, or with any other process running on the system, Windows provides programmatic elements that facilitate communication among different processes. This process of communication between any given process and another process is called interprocess communications, commonly referred to as IPC.

There are many different forms of IPC; some differentiate between client and server, others maintain a specialized division of labor between specialized processes. In the client/server form of IPC, clients generally request services from another process, while the server provides such services. However, any given application can be, and often is, both a client and a server, depending on the request or the service required.

Many different mechanisms are available for communicating between processes. In the interprocess communications section of this volume of the Win32 Library, you'll find the following sections:

Atoms
Clipboard
Handles and Objects
Hooks
Mailslots
Pipes

Atoms are 16-bit integers that enable an application to access a string that has been placed in an atom table. Atom tables, which are defined by the system, store strings and their corresponding identifiers. Atom tables are commonly used on Dynamic Data Exchange (DDE) applications.

The **Clipboard** is the same common clipboard that users of Windows operating systems are accustomed to using—programmatically speaking, the clipboard section in this volume is a set of functions and structures that enable applications to transfer data. The clipboard is an easy way to transfer data between (or within) applications, because all applications have access to the clipboard, but is only appropriate for one-time data exchange (such as a copy and paste procedure). For an ongoing exchange of information between processes, Dynamic Data Exchange Management Library (DDEML) is a better choice.

Handles and Objects represent the functional programmatic pair that enables resources (objects) to be examined or modified (handles). Applications are not allowed to directly access system data or resources (objects), so handles them to do so.

Hooks are used by applications to install subroutines that monitor a system's message traffic, thereby enabling the application to process certain types of messages before they reach the target window procedure. Hooks are not good for performance, as they introduce additional processing burden on the system, and should be used sparingly in production-based applications.

Mailslots are a form of interprocess communication that provides one-way, somewhat unreliable, means of sending data to one or more processes. As the name implies, mailslots are similar to sending a letter; there's a good chance that the message will get to its intended location (in this case, either a server sitting on the network or a group of computers), but there is no delivery guarantee. The lack of guaranteed transmission is attributable to mailslots' use of datagrams, which by definition are not guaranteed to reach their destination. For two-way or non-datagram transmission of messages, use pipes.

Pipes are a means of enabling an interprocess communication that, like mailslots, use a section of shared memory to exchange data. Unlike mailslots, however, pipes use packets (as opposed to datagrams) to transmit data across the network, and also enable two-way communication. The process or application that creates a pipe is called the pipe server, while a process or processes that connect to that pipe are called pipe clients.

Collectively, these IPC-enabling technologies provide the tools that application developers need in order to enable communication between applications. Any given application may implement one of these IPC mechanisms rather than the other, or one of these IPC mechanisms in addition to another. For example, an application almost certainly will use handles and objects, but might not use mailslots.

File Operations

Most applications work with files of some sort, and at one time or another throughout the course of their operation, generally to store or retrieve information from some sort of storage resource (such as a hard drive, network server, or other such devices). File Operations consist of the following categories:

File Input and Output
File Systems

File Input and Output provides the necessary operations that applications or services might perform on files, such as creating, deleting, reading, writing, locking, searching, monitoring, and other such file-related operations. Since files are the basic unit of storage for Windows applications, there are lots of functions, structures, and enumerations associated with File I/O.

File Systems that are supported by the various Windows operating systems differ with each operating system. For example, Windows 98 does not natively support NT File

System (NTFS)—although Windows 98 clients can read from NTFS volumes shared by Windows NT or Windows 2000 computers—while Windows NT versions 4.0 and earlier do not support FAT32. The various programmatic issues surrounding the use, access, and protection of files and resources on the various Windows file systems are explained in the File Systems section.

Debugging

The goal behind debugging an application is to monitor, find, and fix errors in programming code. The Win32 environment provides debugging capabilities to enable application developers to find such application bugs throughout the course of testing and development, as well as a group of supplementary programming capabilities to augment the debugging process. The following lists the set of supplementary debugging capabilities:

Structured Exception Handling

Errors

Structured Exception Handling enables developers of applications to handle software exceptions (exceptions initiated by an application or the operating system) and hardware exceptions (exceptions initiated by the CPU, such as a divide-by-zero exception). With structured exception handling, developers gain control over how such hardware and software exceptions are handled, enabling and facilitating the debugging process.

Errors are fairly self-explanatory; Win32 provides functions and structures that enable developers to have their application receive or display errors, perhaps initiating a particular section of code to handle such errors (such as those explained in structured exception handling).

Unicode

Application developers from around the world, including North America, are realizing that the global economy means that opportunities exist for software programs throughout the world. As such, enabling your application to be *localized*—that is, modified in such a way that it becomes a viable product for various local languages throughout the world—is becoming more of a priority for many projects. Windows provides many features and capabilities to make your Win32 application, from its inception, as international-friendly as possible.

At the heart of the internationalization of Windows applications is **Unicode**. Unicode is an extension of the traditional 8-bit ASCII character set, and was created specifically in an effort to facititate a common international character set. In basic terms, Unicode uses 16 bits for character encoding rather than the commonly used 8-bit character set in ASCII, enabling a complete (single) character set that includes international computing characters. By making your applications Unicode-ready, you take long strides in making your application localization-friendly, and thereby ready for the international market. Windows NT and Windows 2000 were built from the ground up with Unicode support.

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.

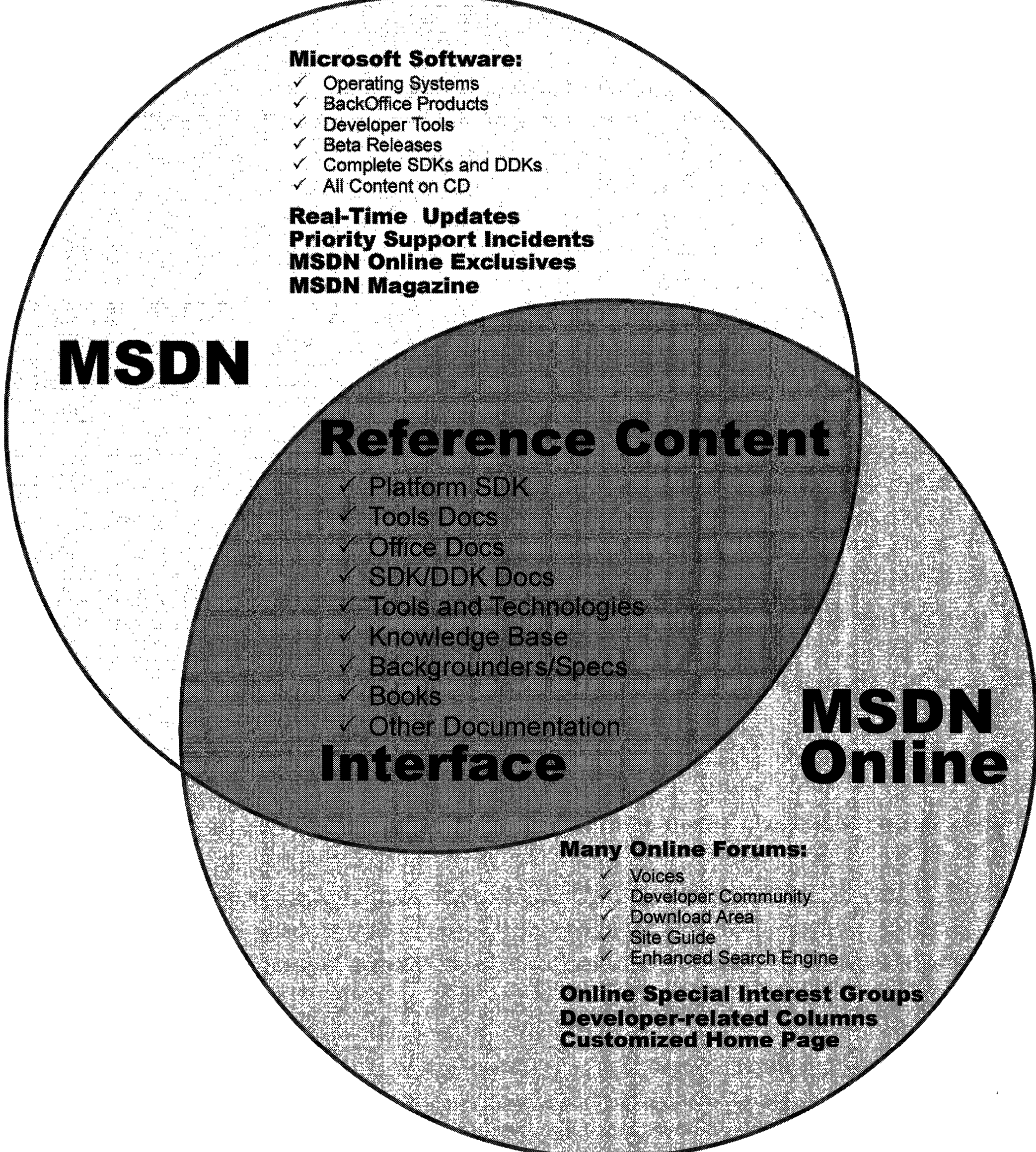


Figure 3-1: The similarities and differences in coverage between MSDN and MSDN Online.

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 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.

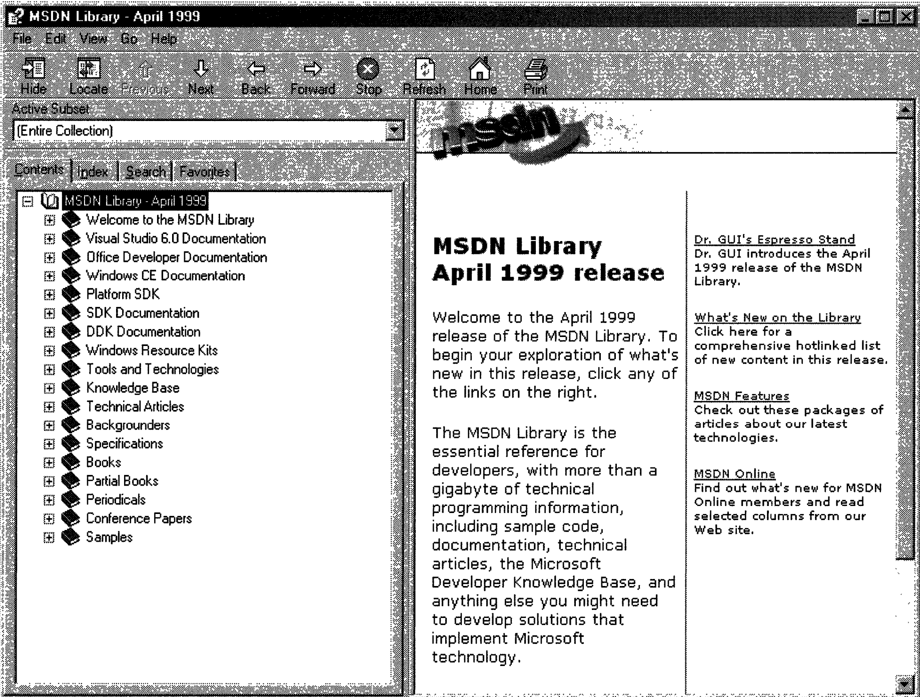


Figure 3-2: The MSDN interface.

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 1.1GB 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.

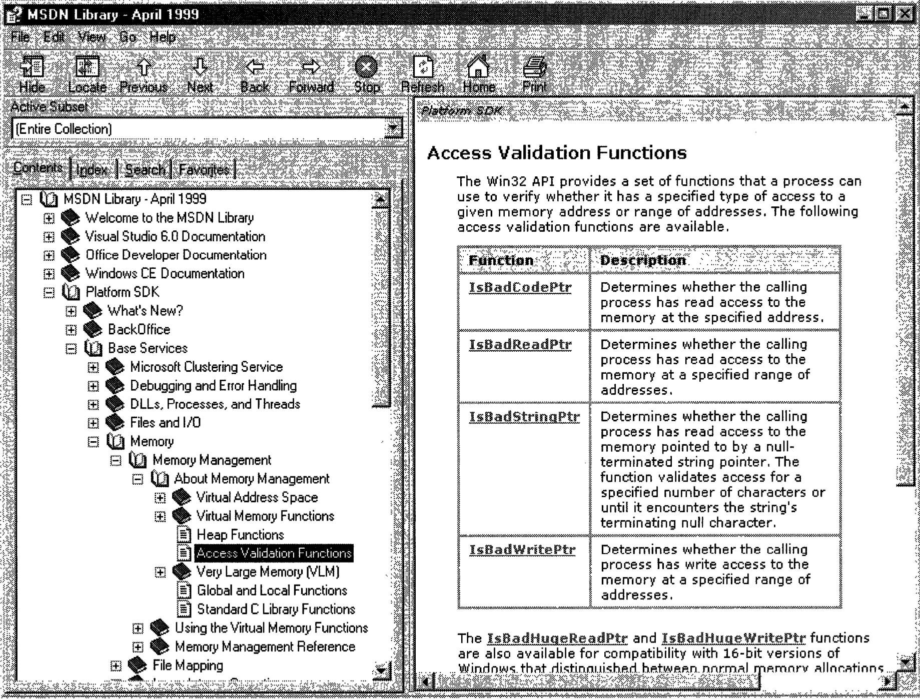


Figure 3-3: Basic exploration of MSDN.

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, 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 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 Enterprise Docs
Visual C++, Platform SDK and WinCE 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.

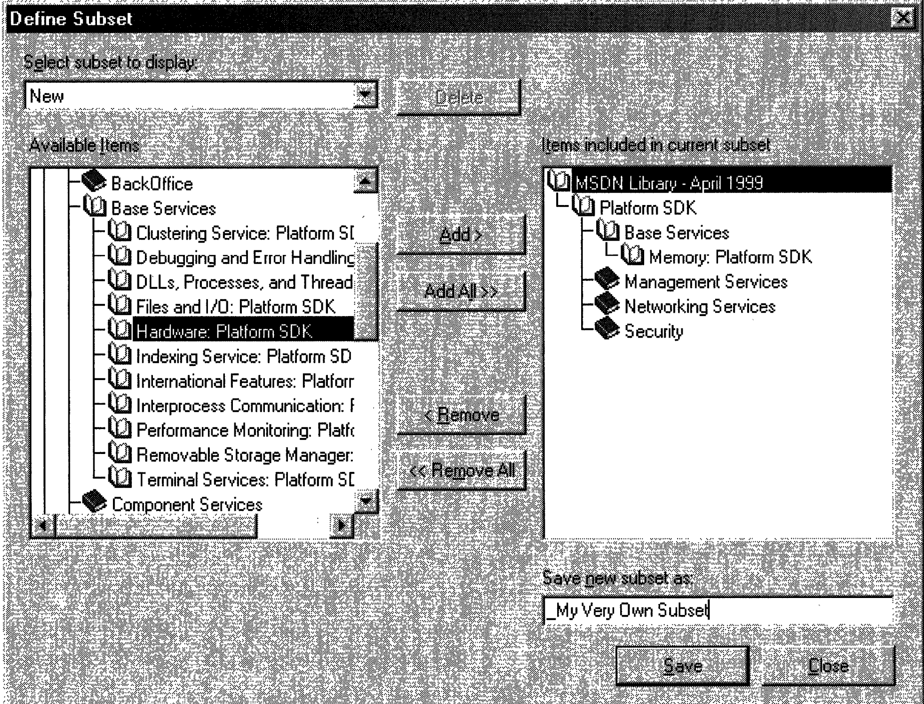


Figure 3-4: The Define Subset window.

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 and 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.

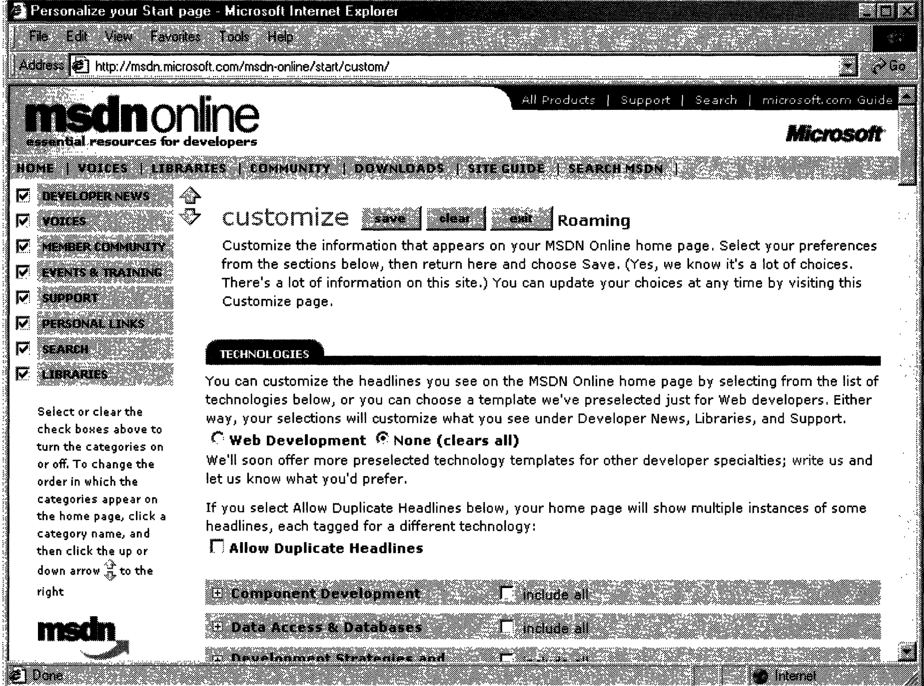


Figure 3-5: The MSDN Online customization page.

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

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

Once you've defined your profile—that is, customized the MSDN Online content you want to see—MSDN Online shows you the most recent information pertinent to your profile when you go to MSDN Online's home page, with the categories you've chosen included in the order you specify. Note that clearing a given 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 will still be available as a site feature.

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

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—include:

Home	Voices
Libraries	Community
Downloads	Site Guide
Search MSDN	

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.

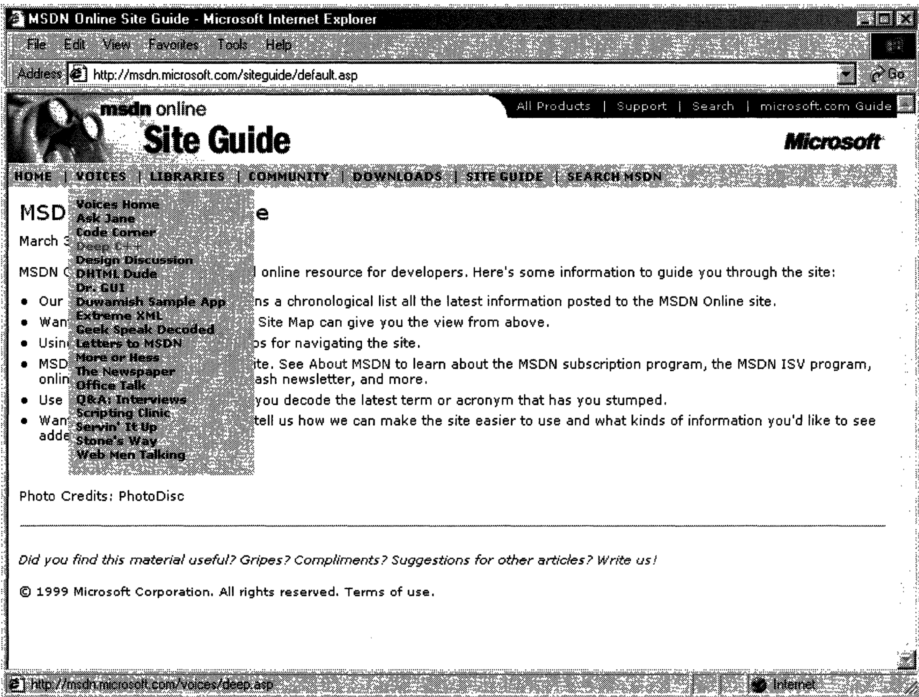


Figure 3-6: The MSDN Online navigation toolbar with its drop-down menus.

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.

Each "voice" in the Voices site 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.

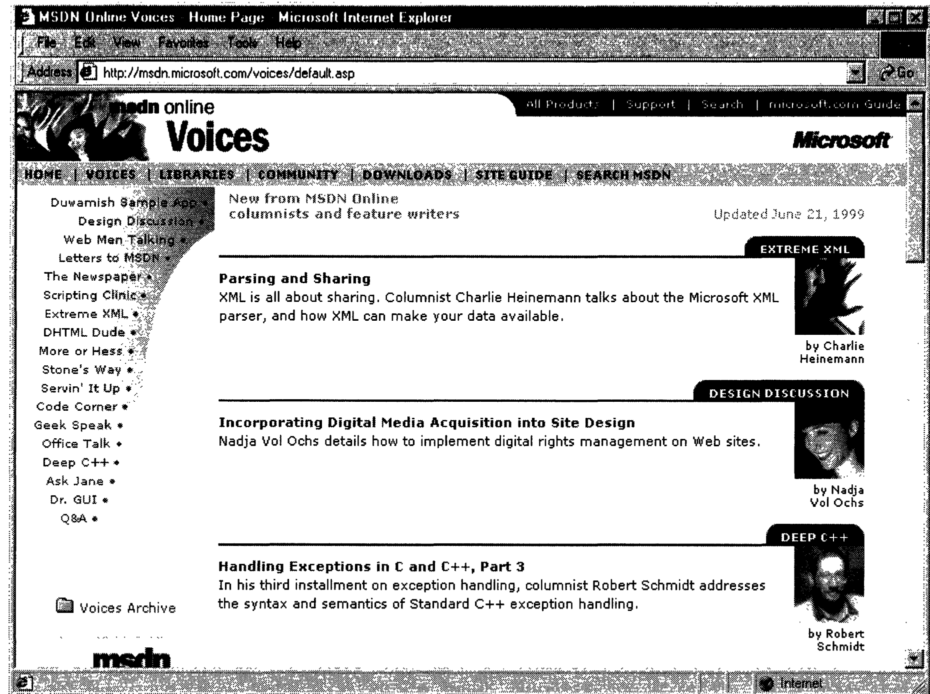


Figure 3-7: The Voices home page.

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*.

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.

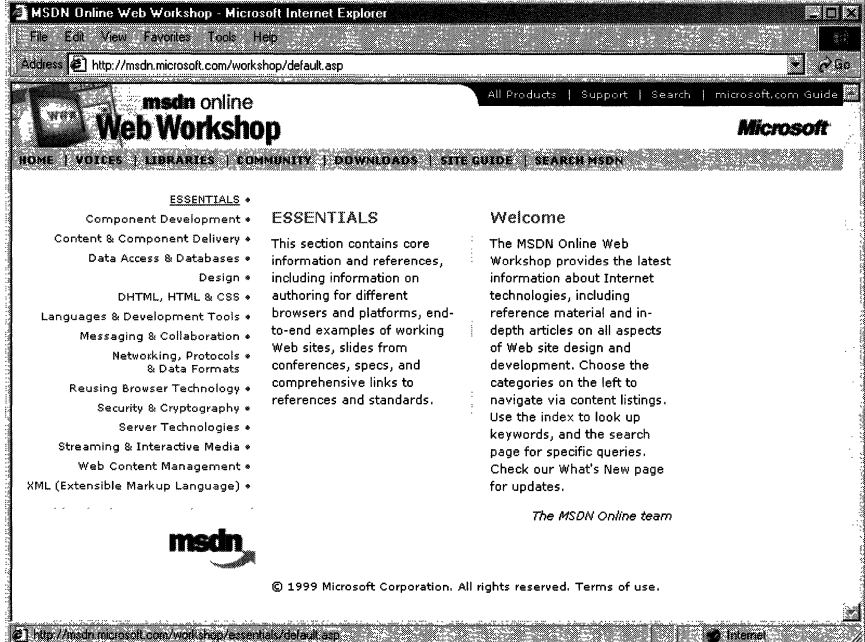


Figure 3-8: The Web Workshop home page, with its bulleted list of exploration start points.

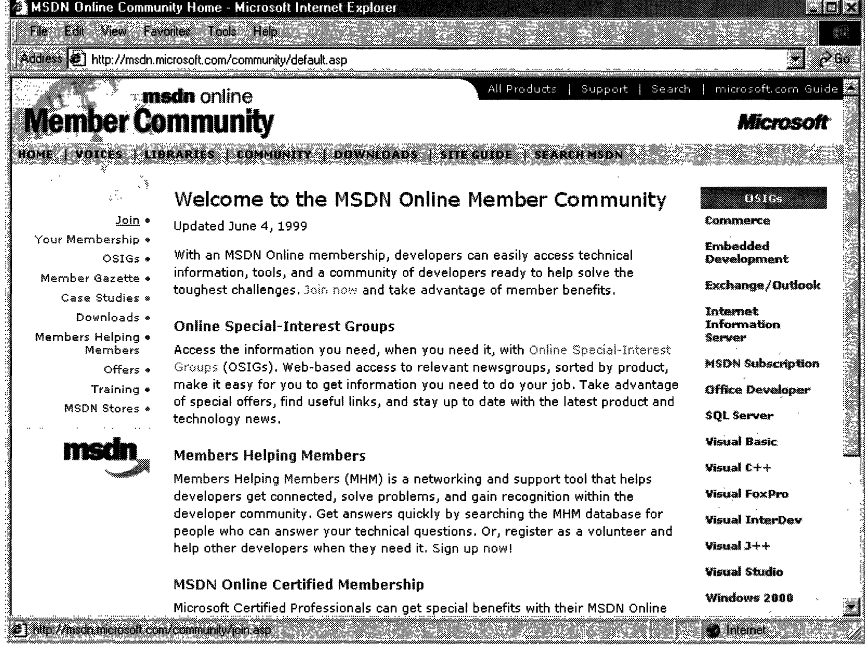


Figure 3-9: The Community home page.

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.

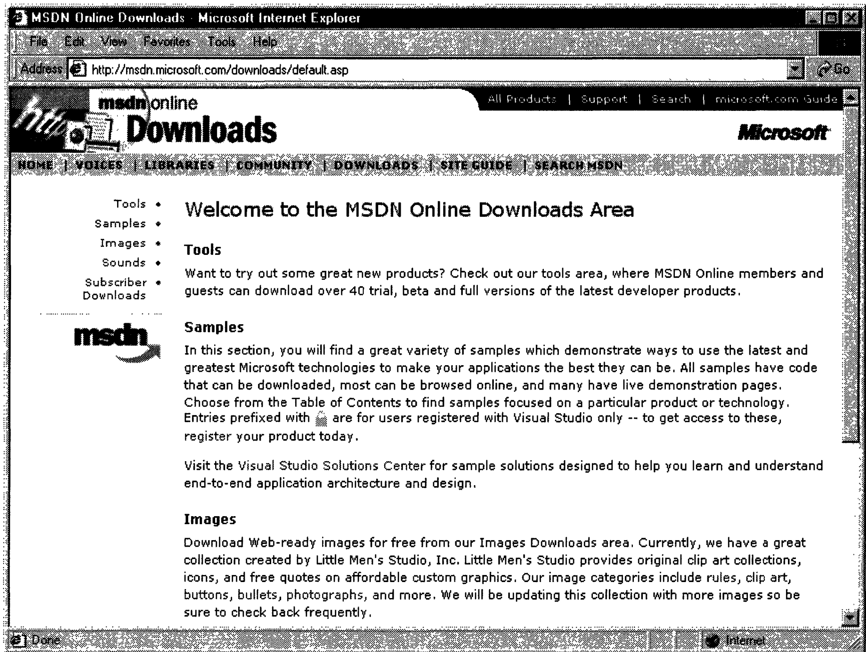


Figure 3-10: The Downloads home page.

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.

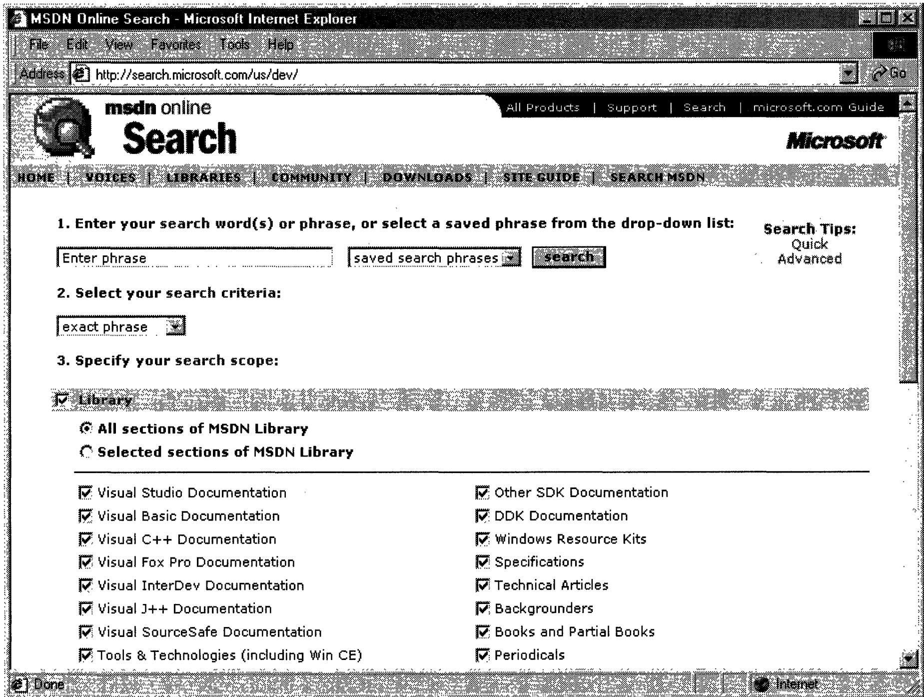


Figure 3-11: The Search MSDN home page.

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 also get all sorts of other timely information, such as chat reminders that let you know when experts on a given subject will be chatting in the MSDN Online Community site. You can also sign up to get newsletters based on your membership in various OSIGs—again, only if

you want to. It's easy for me to suggest that you become a registered user for MSDN Online—I'm a registered user, and it's a great resource.

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

Directory Services Library

Networking 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 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 1

It's impossible to cover everything that a developer might run into when creating a Windows application, but there are common problems that crop up during the development process that *can* be addressed. This chapter—Chapter 5—presents a series of simple but common programming errors for which developers of Windows applications should look out during the development process.

Each Chapter 5 in this pack contains different information. With the tips provided by each volume's Chapter 5 contribution, I hope you'll find the error-avoidance information collectively covered to be fairly useful. The information provided in this collection of five chapters is broken down in the following form:

Volume 1: Overview and Solution Summary

Volume 2: Avoiding Invalid Validations

Volume 3: RPC Errors and Kernel-Mode Specifiers

Volume 4: Buffer Overflows and Miscellaneous Errors

Volume 5: Memory Abuse and Miscalculations

Overview

In order to provide an idea of what you can expect in the next four volumes' Chapter 5 content, the following list outlines the contents of each volume's particular area of common programming errors. It's a bit like a table of contents, with the intention of pointing you to the information you might be interested in on a given error-prone programming day (that's probably a Monday morning, or late Friday afternoon).

Volume 2: User Interface

Avoiding Invalid Validations

Handle-Based Objects

Correlated Parameters

Limits of Exception Handling

Alternate Code Paths

Trusted Data Sources

Volume 3: GDI

RPC Errors and Kernel-Mode Specifiers

RPC Errors

Kernel-Mode Specifiers

Volume 4: Common Controls

Buffer Overflows and Miscellaneous Errors

Buffer Overflows

Miscellaneous Errors

Volume 5: The Windows Shell

Memory Abuse and Miscalculations

Memory Abuse

Miscalculations

Solution Summary

Since you have this book in your hand, you might want to know the short answers to these problems. In order to satisfy that request, this section provides the short-answer listings from each of the summary sections in the other volumes. If you find that these short answers don't provide the specifics you need, grab the volume in which the long answers are provided. Just so you don't have to do too much book juggling, each volume's Chapter 5 also includes the short answers from this list associated with the errors its content covers.

Volume 2: Avoiding Invalid Validations

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.

Volume 3: RPC Errors and Kernel-Mode Specifiers

RPC Errors

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

Kernel-Mode Specifiers

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

Volume 4: Buffer Overflows and Miscellaneous Errors

Buffer Overflows

1. Simple buffer overflow: Always check actual buffer size when accessing a buffer, instead of some known maximum.
2. Size overflow or underflow: When using an offset address, ensure that the location is not beyond either end of the buffer.
3. Abuse of enumerated types: On complex size calculations, ensure that total size is greater than the fixed header.
4. Using internal lengths for comparisons to external input: Beware of strings without NULL termination. If there is a size, use it!

Miscellaneous Errors

1. Dangers of typecasting: Be careful when casting input data to another type.
2. Operator precedence: Double-check precedence order in complex expressions.
3. Conditional termination confusion: Ensure that all clauses of a compound conditional are equivalent (each result should execute the same code), or are special-cased, where appropriate.
4. Misuse of OPTIONAL parameters: Check all pointer parameters for NULL (especially optional parameters)
5. Return value confusion and inconsistencies: Don't hard-code strings in code (for example, "Administrators").
6. Don't rely on volatile objects: Beware of multiple checks of volatile data.
7. Avoid spinlock order problems: Always acquire locks in a consistent order.
8. Determining membership in Administrators group: Beware of (and, preferably, eliminate or reduce) inconsistencies with common interfaces (for example, **GetLastError** and functions returning handles).

Volume 5: Memory Abuse and Miscalculations

Memory Abuse

1. Allocation failures: Always check for allocation failure.
2. Uninitialized memory: Always initialize data.
3. Leaks: Release (free/delete) any allocation after it is no longer needed.
4. Using freed resources: After memory is released, don't access it again! (Suggestion: Set the pointer to NULL on free.)
5. Resource attacks: Have quotas for how much a client can allocate (and ensure client specific data is protected).

Miscalculations

1. Division by zero: Be sure to check for zero for any division.
2. Signed versus unsigned variables: Any signed value can be negative. Furthermore, be wary of the following:
   - Implicit signed values. The values **int** and **enum** are signed; **char** is signed on x86, but *not* on Alpha.
   - Use unsigned values where signed values don't make sense. Counts and lengths are not negative.
   - For range checks, check both upper and lower bounds (or specify unsigned).
3. Floating-point variables: All floating-point operations should be surrounded by **try-except** protection.

CHAPTER 6

Processes, Threads, and DLLs

Processes and Threads

A Win32-based application consists of one or more processes. A *process*, in the simplest terms, is an executing program. One or more threads run in the context of the process. A *thread* is the basic unit to which the operating system allocates processor time. A thread can execute any part of the process code, including parts currently being executed by another thread. A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them.

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

About Processes and Threads

Each process provides the resources needed to execute a program. A process has a virtual address space, executable code, data, object handles, environment variables, a base priority, and minimum and maximum working set sizes. Each process is started with a single thread, often called the *primary thread*, but can create additional threads from any of its threads.

All threads of a process share its virtual address space and system resources. In addition, each thread maintains exception handlers, a scheduling priority, and a set of structures the system will use to save the thread context until it is scheduled. The *thread context* includes the thread's set of machine registers, the kernel stack, a thread environment block, and a user stack in the address space of the thread's process.

Windows NT/2000 and Windows 95/98 support *preemptive multitasking*, which creates the effect of simultaneous execution of multiple threads from multiple processes. On a multiprocessor computer, Windows NT/2000 can simultaneously execute as many threads as there are processors on the computer.

Multitasking

A multitasking operating system divides the available processor time among the processes or threads that need it. The system is designed for preemptive multitasking; it allocates a processor *time slice* to each thread it executes. The currently executing thread is suspended when its time slice elapses, allowing another thread to run. When

the system switches from one thread to another, it saves the context of the preempted thread and restores the saved context of the next thread in the queue.

The length of the time slice depends on the operating system and the processor. Because each time slice is small (approximately 20 milliseconds), multiple threads appear to be executing at the same time. This is actually the case on multiprocessor systems, where the executable threads are distributed among the available processors. However, you must use caution when using multiple threads in an application, because system performance can decrease if there are too many threads.

Advantages of Multitasking

To the user, the advantage of multitasking is the ability to have several applications open and working at the same time. For example, a user can edit a file with one application while another application is recalculating a spreadsheet.

To the application developer, the advantage of multitasking is the ability to create applications that use more than one process and to create processes that use more than one thread of execution. For example, a process can have a user interface thread that manages interactions with the user (keyboard and mouse input), and worker threads that perform other tasks while the user interface thread waits for user input. If you give the user interface thread a higher priority, the application will be more responsive to the user, while the worker threads use the processor efficiently during the times when there is no user input.

When to Use Multitasking

There are two ways to implement multitasking: as a single process with multiple threads or as multiple processes, each with one or more threads. An application can put each thread that requires a private address space and private resources into its own process, to protect it from the activities of other process threads.

A multithreaded process can manage mutually exclusive tasks with threads, such as providing a user interface and performing background calculations. Creating a multithreaded process can also be a convenient way to structure a program that performs several similar or identical tasks concurrently. For example, a named pipe server can create a thread for each client process that attaches to the pipe. This thread manages the communication between the server and the client. Your process could use multiple threads to accomplish the following tasks:

- Manage input for multiple windows.
- Manage input from several communications devices.
- Distinguish tasks of varying priority. For example, a high-priority thread manages time-critical tasks, and a low-priority thread performs other tasks.
- Allow the user interface to remain responsive, while allocating time to background tasks.

It is typically more efficient for an application to implement multitasking by creating a single, multithreaded process, rather than creating multiple processes, for the following reasons:

- The system can perform a context switch more quickly for threads than processes, because a process has more overhead than a thread does (the process context is larger than the thread context).
- All threads of a process share the same address space and can access the global variables of the process, which can simplify communication between threads.
- All threads of a process can share open handles to resources, such as files and pipes.

The Win32 API also provides alternative methods that can be used in the place of multithreading. The most significant of these methods are asynchronous input and output (I/O), I/O completion ports, asynchronous procedure calls (APC), and the ability to wait for multiple events.

A single thread can initiate multiple time-consuming I/O requests that can run concurrently using asynchronous I/O. Asynchronous I/O can be performed on files, pipes, and serial communication devices. For more information, see *Synchronization and Overlapped Input and Output*.

A single thread can block its own execution while waiting for any one or all of several events to occur. This is more efficient than using multiple threads, each waiting for a single event, and more efficient than using a single thread that consumes processor time by continually checking for events to occur. For more information, see *Wait Functions*.

Multitasking Considerations

The recommended guideline is to use as few threads as possible, thereby minimizing the use of system resources. This improves performance. Multitasking has resource requirements and potential conflicts to be considered when designing your application. The resource requirements are as follows:

- The system consumes memory for the context information required by both processes and threads. Therefore, the number of processes and threads that can be created is limited by available memory.
- Keeping track of a large number of threads consumes significant processor time. If there are too many threads, most of them will not be able to make significant progress. If most of the current threads are in one process, threads in other processes are scheduled less frequently.

Providing shared access to resources can create conflicts. To avoid them, you must synchronize access to shared resources. This is true for system resources (such as communications ports), resources shared by multiple processes (such as file handles), or the resources of a single process (such as global variables) accessed by multiple threads. Failure to synchronize access properly (in the same or in different processes) can lead to problems such as *deadlock* and *race conditions*. The Win32 API provides a

set of synchronization objects and functions you can use to coordinate resource sharing among multiple threads. For more information about synchronization, see *Synchronizing Execution of Multiple Threads*. Reducing the number of threads makes it easier and more effective to synchronize resources.

A good design for a multithreaded application is the pipeline server. In this design, you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

Scheduling

The system scheduler controls multitasking by determining which of the competing threads receives the next processor time slice. The scheduler determines which thread runs next using its scheduling priority.

Scheduling Priorities

Each thread is assigned a scheduling priority. The priority levels range from zero (lowest priority) to 31 (highest priority). Only the zero-page thread can have a priority of zero. The zero-page thread is a system thread.

The priority of each thread is determined by the following criteria:

- The priority class of its process
- The priority level of the thread within the priority class of its process

The priority class and priority level are combined to form the *base priority* of a thread.

Priority Class

Each process belongs to one of the following priority classes:

IDLE_PRIORITY_CLASS
BELOW_NORMAL_PRIORITY_CLASS
NORMAL_PRIORITY_CLASS
ABOVE_NORMAL_PRIORITY_CLASS
HIGH_PRIORITY_CLASS
REALTIME_PRIORITY_CLASS

Windows 2000: Note that BELOW_NORMAL_PRIORITY_CLASS and ABOVE_NORMAL_PRIORITY_CLASS are new for Windows 2000.

By default, the priority class of a process is NORMAL_PRIORITY_CLASS. Use the **CreateProcess** function to specify the priority class of a child process when you create it. If the calling process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS, the new process will inherit this class. Use the **GetPriorityClass** function to determine the current priority class of a process and the **SetPriorityClass** function to change the priority class of a process.

Processes that monitor the system, such as screen savers or applications that periodically update a display, should use IDLE_PRIORITY_CLASS. This prevents the threads of this process, which do not have high priority, from interfering with higher priority threads.

Use HIGH_PRIORITY_CLASS with care. If a thread runs at the highest priority level for extended periods, other threads in the system will not get processor time. If several threads are set at high priority at the same time, the threads lose their effectiveness. The high-priority class should be reserved for threads that must respond to time-critical events. If your application performs one task that requires the high-priority class while the rest of its tasks are normal priority, use **SetPriorityClass** to raise the priority class of the application temporarily; then reduce it after the time-critical task has been completed. Another strategy is to create a high-priority process that has all of its threads blocked most of the time, awakening threads only when critical tasks are needed. The important point is that a high-priority thread should execute for a brief time, and only when it has time-critical work to perform.

You should almost never use REALTIME_PRIORITY_CLASS, because this interrupts system threads that manage mouse input, keyboard input, and background disk flushing. This class can be appropriate for applications that "talk" directly to hardware or that perform brief tasks that should have limited interruptions.

Priority Level

The following are priority levels within each priority class:

THREAD_PRIORITY_IDLE
THREAD_PRIORITY_LOWEST
THREAD_PRIORITY_BELOW_NORMAL
THREAD_PRIORITY_NORMAL
THREAD_PRIORITY_ABOVE_NORMAL
THREAD_PRIORITY_HIGHEST
THREAD_PRIORITY_TIME_CRITICAL

All threads are created using THREAD_PRIORITY_NORMAL. This means that the thread priority is the same as the process priority class. After you create a thread, use the **SetThreadPriority** function to adjust its priority relative to other threads in the process.

A typical strategy is to use THREAD_PRIORITY_ABOVE_NORMAL or THREAD_PRIORITY_HIGHEST for the process's input thread, to ensure that the application is responsive to the user. Background threads, particularly those that are processor intensive, can be set to THREAD_PRIORITY_BELOW_NORMAL or THREAD_PRIORITY_LOWEST, to ensure that they can be preempted when necessary. However, if you have a thread waiting for another thread with a lower priority to complete some task, be sure to block the execution of the waiting high-priority thread. To do this, use a wait function, critical section, or the **Sleep** function, **SleepEx**, or **SwitchToThread** function. This is preferable to having the thread execute a loop. Otherwise, the process may become deadlocked, because the thread with lower priority is never scheduled.

To determine the current priority level of a thread, use the **GetThreadPriority** function.

Base Priority

The priority level of a thread is determined by both the priority class of its process and its priority level. The priority class and priority level are combined to form the *base priority* of each thread.

The following table shows the base priority levels for combinations of priority class and priority value.

	Process Priority Class	Thread Priority Level
1	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
1	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
2	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
3	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
4	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
4	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
5	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
5	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
5	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
6	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
6	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
6	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
7	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
7	Background NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
7	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
8	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
8	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
8	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
8	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
9	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
9	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
9	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
10	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
10	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL

11	Foreground NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
11	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
11	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
12	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
12	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
13	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
14	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
15	HIGH_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	IDLE_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	BELOW_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
15	ABOVE_NORMAL_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL
16	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_IDLE
22	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_LOWEST
23	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_BELOW_NORMAL
24	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_NORMAL
25	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_ABOVE_NORMAL
26	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_HIGHEST
31	REALTIME_PRIORITY_CLASS	THREAD_PRIORITY_TIME_CRITICAL

Context Switches

The scheduler maintains a queue of executable threads for each priority level. These are known as *ready threads*. When a processor becomes available, the system performs a *context switch*. The steps in a context switch are:

1. Save the context of the thread that just finished executing.
2. Place the thread that just finished executing at the end of the queue for its priority.
3. Find the highest priority queue that contains ready threads.
4. Remove the thread at the head of the queue, load its context, and execute it.

The following classes of threads are not ready threads.

- Threads created with the CREATE_SUSPENDED flag
- Threads halted during execution with the **SuspendThread** or **SwitchToThread** function
- Threads waiting for a synchronization object or input.

Until threads that are suspended or blocked become ready to run, the scheduler does not allocate any processor time to them, regardless of their priority.

The most common reasons for a context switch are:

- The time slice has elapsed.
- A thread with a higher priority has become ready to run.
- A running thread needs to wait.

When a running thread needs to wait, it relinquishes the remainder of its time slice.

Priority Boosts

Each thread has a *dynamic priority*. This is the priority the scheduler uses to determine which thread to execute. Initially, a thread's dynamic priority is the same as its base priority. The system can boost and lower the dynamic priority, to ensure that it is responsive and that no threads are starved for processor time. The system does not boost the priority of threads with a base priority level between 16 and 31. Only threads with a base priority between 0 and 15 receive dynamic priority boosts.

The system boosts the dynamic priority of a thread to enhance its responsiveness as follows:

- When a process that uses NORMAL_PRIORITY_CLASS is brought to the foreground, the scheduler boosts the priority class of the process associated with the foreground window, so that it is greater than or equal to the priority class of any background processes. The priority class returns to its original setting when the process is no longer in the foreground.

  Windows NT/2000: The user can control the boosting of processes that use NORMAL_PRIORITY_CLASS through the System control panel application.
- When a window receives input, such as timer messages, mouse messages, or keyboard input, the scheduler boosts the priority of the thread that owns the window.
- When the wait conditions for a blocked thread are satisfied, the scheduler boosts the priority of the thread. For example, when a wait operation associated with disk or keyboard I/O finishes, the thread receives a priority boost.

  Windows NT/2000: You can disable the priority-boosting feature by calling the **SetProcessPriorityBoost** or **SetThreadPriorityBoost** function. To determine whether this feature has been disabled, call the **GetProcessPriorityBoost** or **GetThreadPriorityBoost** function.

After raising a thread's dynamic priority, the scheduler reduces that priority by one level each time the thread completes a time slice, until the thread drops back to its base priority. A thread's dynamic priority is never less than its base priority.

Priority Inversion

Priority inversion occurs when two or more threads with different priorities are in contention to be scheduled. Consider a simple case with three threads: thread 1, thread 2, and thread 3. Thread 1 is high priority and becomes ready to be scheduled. Thread 2, a low-priority thread, is executing code in a critical section. Thread 1, the high-priority thread, begins waiting for a shared resource from thread 2. Thread 3 has medium

priority. Thread 3 receives all the processor time, because the high-priority thread (thread 1) is waiting for shared resources from the low-priority thread (thread 2). Thread 2 won't leave the critical section, because does not have the highest priority and won't be scheduled.

- **Windows NT/2000:** The scheduler solves this problem by randomly boosting the priority of the ready threads (in this case, the low-priority lock-holders). The low-priority threads run long enough to exit the critical section, and the high-priority thread can enter the critical section. If the low-priority thread doesn't get enough CPU time to exit the critical section the first time, it will get another chance during the next round of scheduling.
- **Windows 95:** If a high-priority thread is dependent on a low-priority thread that will not be allowed to run because a medium priority thread is getting all of the CPU time, the system recognizes that the high-priority thread is dependent on the low-priority thread. It will boost the low-priority thread's priority up to the priority of the high-priority thread. This will allow the thread that formerly had the lowest priority to run and release the high-priority thread that was waiting for it.

Multiple Processors

Windows NT uses a symmetric multiprocessing (SMP) model to schedule threads on multiple processors. With this model, any thread can be assigned to any processor. Therefore, scheduling threads on a computer with multiple processors is similar to scheduling threads on a computer with a single processor. However, the scheduler has a pool of processors, so that it can schedule threads to run concurrently. Scheduling is still determined by thread priority. However, on a multiprocessor computer, you can also affect scheduling by setting thread affinity and thread ideal processor, as discussed here.

Thread Affinity

Thread affinity forces a thread to run on a specific subset of processors. Use the **SetProcessAffinityMask** function to specify thread affinity for all threads of the process. To set the thread affinity for a single thread, use the **SetThreadAffinityMask** function. The thread affinity must be a subset of the process affinity. You can obtain the current process affinity by calling the **GetProcessAffinityMask** function.

Setting thread affinity should generally be avoided, because it can interfere with the scheduler's ability to schedule threads effectively across processors. This can decrease the performance gains produced by parallel processing. An appropriate use of thread affinity is testing each processor.

Thread Ideal Processor

When you specify a *thread ideal processor*, the scheduler runs the thread on the specified processor when possible. Use the **SetThreadIdealProcessor** function to specify a preferred processor for a thread. This does not guarantee that the ideal processor will be chosen, but provides a useful hint to the scheduler.

Multiple Threads

Each process is started with a single thread, but can create additional threads from any of its threads.

Creating Threads

The **CreateThread** function creates a new thread for a process. The creating thread must specify the starting address of the code that the new thread is to execute. Typically, the starting address is the name of a function defined in the program code. This function takes a single parameter and returns a DWORD value. A process can have multiple threads simultaneously executing the same function.

The following example demonstrates how to create a new thread that executes the locally defined function, ThreadFunc.

```
DWORD WINAPI ThreadFunc( LPVOID lpParam )
{
    char szMsg[80];

    wsprintf( szMsg, "ThreadFunc: Parameter = %d\n", *lpParam );
    MessageBox( NULL, szMsg, "Thread created.", MB_OK );

    return 0;
}
VOID main( VOID )
{
    DWORD dwThreadId, dwThrdParam = 1;
    HANDLE hThread;
    hThread = CreateThread(
        NULL,                        // no security attributes
        0,                           // use default stack size
        ThreadFunc,                  // thread function
        &dwThrdParam,                // argument to thread function
        0,                           // use default creation flags
        &dwThreadId);                // returns the thread identifier

    // Check the return value for success.

    if (hThread == NULL)
      ErrorExit( "CreateThread failed." );

    CloseHandle( hThread );
}
```

For simplicity, this example passes a pointer to a DWORD value as an argument to the thread function. This could be a pointer to any type of data or structure, or it could be

omitted altogether by passing a NULL pointer and deleting the references to the parameter in ThreadFunc.

It is risky to pass the address of a local variable if the creating thread exits before the new thread, because the pointer becomes invalid. Instead, either pass a pointer to dynamically allocated memory or make the creating thread wait for the new thread to terminate. Data can also be passed from the creating thread to the new thread using global variables. With global variables, it is usually necessary to synchronize access by multiple threads. For more information about synchronization, see *Synchronizing Execution of Multiple Threads*.

In processes where a thread might create multiple threads to execute the same code, it is inconvenient to use global variables. For example, a process that enables the user to open several files at the same time can create a new thread for each file, with each of the threads executing the same thread function. The creating thread can pass the unique information (such as the file name) required by each instance of the thread function as an argument. You cannot use a single global variable for this purpose, but you could use a dynamically allocated string buffer.

The creating thread can use the arguments to **CreateThread** to specify the following:

- The security attributes for the handle to the new thread. These security attributes include an inheritance flag that determines whether the handle can be inherited by child processes. The security attributes also include a security descriptor, which the system uses to perform access checks on all subsequent uses of the thread's handle before access is granted.
- The initial stack size of the new thread. The thread's stack is allocated automatically in the memory space of the process; the system increases the stack as needed and frees it when the thread terminates.
- A creation flag that enables you to create the thread in a suspended state. When suspended, the thread does not run until the **ResumeThread** function is called.

You can also create a thread by calling the **CreateRemoteThread** function. This function is used by debugger processes to create a thread that runs in the address space of the process being debugged.

Thread Stack Size

Each new thread receives its own stack space, consisting of both committed and reserved memory. By default, each thread uses 1 MB of reserved memory, and one page of committed memory. The system will commit one page blocks from the reserved stack memory as needed, until the stack cannot grow any father. To specify a different default stack size, use the STACKSIZE statement in the module definition (.DEF) file. Your linker may also support a command-line option for setting the stack size. For more information, see the documentation included with your linker.

To increase the amount of stack space which is to be initially committed for a thread, specify the value in the *dwStackSize* parameter of the **CreateThread** function. This value is rounded to the nearest page and used to set the initial size of the committed

memory. The call to **CreateThread** will fail if there is not enough memory to commit the number of bytes you request. If the *dwStackSize* value is smaller than the default size, the new thread uses the same size as the thread that created it.

The stack is freed when the thread terminates.

Thread Handles and Identifiers

When a new thread is created by the **CreateThread** or **CreateRemoteThread** function, a handle to the thread is returned. By default, this handle has full access rights, and—subject to security access checking—can be used in any of the functions that accept a thread handle. This handle can be inherited by child processes, depending on the inheritance flag specified when it is created. The handle can be duplicated by **DuplicateHandle**, which enables you to create a thread handle with a subset of the access rights. The handle is valid until closed, even after the thread it represents has been terminated.

The **CreateThread** and **CreateRemoteThread** functions also return an identifier that uniquely identifies the thread throughout the system. A thread can use the **GetCurrentThreadId** function to get its own thread identifier. The identifiers are valid from the time the thread is created until the thread has been terminated.

Windows 2000: If you have a thread identifier, you can get the thread handle by calling the **OpenThread** function. **OpenThread** enables you to specify the handle's access rights and whether it can be inherited.

Windows NT 4.0 and earlier, Windows 95/98: The Win32 API does not provide a way to get the thread handle from the thread identifier. If the handles were made available this way, the owning process could fail because another process unexpectedly performed an operation on one of its threads, such as suspending it, resuming it, adjusting its priority, or terminating it. Instead, you must request the handle from the thread creator or the thread itself.

A thread can use the **GetCurrentThread** function to retrieve a *pseudo handle* to its own thread object. This pseudo handle is valid only for the calling process; it cannot be inherited or duplicated for use by other processes. To get the real handle to the thread, given a pseudo handle, use the **DuplicateHandle** function.

Suspending Thread Execution

A thread can suspend and resume the execution of another thread using the **SuspendThread** and **ResumeThread** functions. While a thread is suspended, it is not scheduled for time on the processor.

The **SuspendThread** function is not particularly useful for synchronization because it does not control the point in the code at which the thread's execution is suspended. However, you might want to suspend a thread in a situation where you are waiting for user input that could cancel the work the thread is performing. If the user input cancels the work, have the thread exit; otherwise, call **ResumeThread**.

If a thread is created in a suspended state (with the CREATE_SUSPENDED flag), it does not begin to execute until another thread calls **ResumeThread** with a handle to the suspended thread. This can be useful for initializing the thread's state before it begins to execute. See *Using a Multithreaded Multiple Document Interface Application* for an example that uses this method to modify the thread's priority before it can run. Suspending a thread at creation can be useful for one-time synchronization, because this ensures that the suspended thread will execute the starting point of its code when you call **ResumeThread**.

A thread can temporarily yield its execution for a specified interval by calling the **Sleep** or **SleepEx** functions. This is useful particularly in cases where the thread responds to user interaction, because it can delay execution long enough to allow users to observe the results of their actions. During the sleep interval, the thread is not scheduled for time on the processor.

The **SwitchToThread** function is similar to **Sleep** and **SleepEx**, except that you cannot specify the interval. **SwitchToThread** allows the thread to give up its time slice.

Synchronizing Execution of Multiple Threads

To avoid race conditions and deadlocks, it is necessary to synchronize access by multiple threads to shared resources. Synchronization is also necessary to ensure that interdependent code is executed in the proper sequence.

The Win32 API provides a number of objects whose handles can be used to synchronize multiple threads. These objects include:

- Console input buffers
- Events
- Mutexes
- Processes
- Semaphores
- Threads
- Timers

The state of each of these objects is either signaled or not signaled. When you specify a handle to any of these objects in a call to one of the wait functions, the execution of the calling thread is blocked until the state of the specified object becomes signaled.

Some of these objects are useful in blocking a thread until some event occurs. For example, a console input buffer handle is signaled when there is unread input, such as a keystroke or mouse button click. Process and thread handles are signaled when the process or thread terminates. This allows a process, for example, to create a child process and then block its own execution until the new process has terminated.

Other objects are useful in protecting shared resources from simultaneous access. For example, multiple threads can each have a handle to a mutex object. Before accessing a shared resource, the threads must call one of the wait functions to wait for the state of

the mutex to be signaled. When the mutex becomes signaled, only one waiting thread is released to access the resource. The state of the mutex is immediately reset to not signaled so any other waiting threads remain blocked. When the thread is finished with the resource, it must set the state of the mutex to signaled to allow other threads to access the resource.

For the threads of a single process, critical-section objects provide a more efficient means of synchronization than mutexes. A critical section is used like a mutex to enable one thread at a time to use the protected resource. A thread can use the **EnterCriticalSection** function to request ownership of a critical section. If it is already owned by another thread, the requesting thread is blocked. A thread can use the **TryEnterCriticalSection** function to request ownership of a critical section, without blocking upon failure to obtain the critical section. After it receives ownership, the thread is free to use the protected resource. The execution of the other threads of the process is not affected unless they attempt to enter the same critical section.

The **WaitForInputIdle** function makes a thread wait until a specified process is initialized and waiting for user input with no input pending. Calling **WaitForInputIdle** can be useful for synchronizing parent and child processes, because **CreateProcess** returns without waiting for the child process to complete its initialization.

For more information, see *Synchronization*.

Multiple Threads and GDI Objects

To enhance performance, access to graphical device interface (GDI) objects (such as palettes, device contexts, regions, and the like) is not serialized. This creates a potential danger for processes that have multiple threads sharing these objects. For example, if one thread deletes a GDI object while another thread is using it, the results are unpredictable. This danger can be avoided simply by not sharing GDI objects. If sharing is unavoidable (or desirable), the application must provide its own mechanisms for synchronizing access. For more information about synchronizing access, see *Synchronizing Execution of Multiple Threads*.

Thread Local Storage

All threads of a process share the virtual address space and the global variables of that process. The local variables of a thread function are local to each thread that runs the function. However, the static or global variables used by that function have the same value for all threads. With thread local storage (TLS), you can create a unique copy of a variable for each thread. Using TLS, one thread allocates an index that can be used by any thread of the process to retrieve its unique copy.

Use the following steps to implement TLS:

1. Use the **TlsAlloc** function during process or dynamic-link library (DLL) initialization to allocate a TLS index.
2. For each thread that needs to use the TLS index, allocate dynamic storage, then use the **TlsSetValue** function to associate the index with a pointer to the dynamic storage.

3. When you need a thread to access its storage, specify the TLS index in a call to the **TlsGetValue** function to retrieve the pointer.
4. When each thread no longer needs the dynamic storage that it has associated with a TLS index, it must free the index. When all threads have finished using a TLS index, use the **TlsFree** function to free the index.

The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexes available in each process. This minimum is guaranteed to be at least 64 for all systems.

Windows 2000: There is a limit of 1088 TLS indexes per process.

Windows NT 4.0 and earlier: There is a limit of 64 TLS indexes per process.

It is ideal to use TLS in a DLL. Perform the initial TLS operations in the **DllMain** function in the context of the process or thread attaching to the DLL. When a new process attaches to the DLL, call **TlsAlloc** in the entry-point function to allocate a TLS index for that process. Then store the TLS index in a global variable that is private to each attached process. When a new thread attaches to the DLL, allocate dynamic memory for that thread in the entry-point function, and use **TlsSetValue** with the TLS index from **TlsAlloc** to save private data to the index. Then you can use the TLS index in a call to **TlsGetValue** to access the private data for the calling thread from within any function in the DLL. When a process detaches from the DLL, call **TlsFree**.

For an example illustrating the use of thread local storage, see *Using Thread Local Storage*.

Creating Windows in Threads

Any thread can create a window. The thread that creates the window owns the window and its associated message queue. Therefore, the thread must provide a message loop to process the messages in its message queue. In addition, you must use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx** in that thread, rather than the other wait functions, so that it can process messages. Otherwise, the system can become deadlocked when the thread is sent a message while it is waiting.

The **AttachThreadInput** function can be used to allow a set of threads to share the same input state. By sharing input state, the threads share their concept of the active window. By doing this, one thread can always activate another thread's window. This function is also useful for sharing focus state, mouse capture state, keyboard state, and window Z-order state among windows created by different threads whose input state is shared.

Terminating a Thread

A thread executes until one of the following events occurs:

- The thread calls the **ExitThread** function.
- Any thread of the process calls the **ExitProcess** function.
- The thread function returns.
- Any thread calls the **TerminateThread** function with a handle to the thread.

- Any thread calls the **TerminateProcess** function with a handle to the process.

The **GetExitCodeThread** function returns the termination status of a thread. While a thread is executing, its termination status is STILL_ACTIVE. When a thread terminates, its termination status changes from STILL_ACTIVE to the exit code of the thread. The exit code is either the value specified in the call to **ExitThread**, **ExitProcess**, **TerminateThread**, or **TerminateProcess**, or the value returned by the thread function.

When a thread terminates, the state of the thread object changes to signaled, releasing any other threads that had been waiting for the thread to terminate. For more about synchronization, see *Synchronizing Execution of Multiple Threads*.

If a thread is terminated by **ExitThread**, the system calls the entry-point function of each attached DLL with a value indicating that the thread is detaching from the DLL (unless you call the **DisableThreadLibraryCalls** function). If a thread is terminated by **ExitProcess**, the DLL entry-point functions are invoked once, to indicate that the process is detaching. DLLs are not notified when a thread is terminated by **TerminateThread** or **TerminateProcess**. For more information about DLLs, see *Dynamic Link Libraries*.

Warning The **TerminateThread** and **TerminateProcess** functions should be used only in extreme circumstances, since they do not allow threads to clean up, do not notify attached DLLs, and do not free the initial stack.

The following steps provide a better solution:

- Create an event object using the **CreateEvent** function.
- Create the threads.
- Each thread monitors the event state by calling the **WaitForSingleObject** function. Use a wait time-out interval of zero.
- Each thread terminates its own execution when the event is set to the signaled state (**WaitForSingleObject** returns WAIT_OBJECT_0).

Thread Times

The **GetThreadTimes** function obtains timing information for a thread. It returns the thread creation time, how much time the thread has been executing in kernel mode, and how much time the thread has been executing in user mode. These times do not include time spent executing system threads or waiting in a suspended or blocked state. If the thread has exited, **GetThreadTimes** returns the thread exit time.

Thread Security and Access Rights

Windows NT/Windows 2000 security enables you to control access to thread objects. For more information about security, see *Access-Control Model*.

You can specify a security descriptor for a thread when you call the **CreateProcess**, **CreateProcessAsUser**, **CreateProcessWithLogonW**, **CreateThread**, or

CreateRemoteThread function. To retrieve a thread's security descriptor, call the **GetSecurityInfo** function. To change a thread's security descriptor, call the **SetSecurityInfo** function.

The handle returned by the **CreateThread** function has THREAD_ALL_ACCESS access to the thread object. When you call the **GetCurrentThread** function, the system returns a pseudohandle with the maximum access that the thread's security descriptor allows the caller.

The valid access rights for thread objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the following thread-specific access rights.

Value	Meaning
SYNCHRONIZE	A standard right required to wait for the thread to exit.
THREAD_ALL_ACCESS	Specifies all possible access rights for a thread object.
THREAD_DIRECT_IMPERSONATION	Required for a server thread that impersonates a client.
THREAD_GET_CONTEXT	Required to read the context of a thread using **GetThreadContext**.
THREAD_IMPERSONATE	Required to use a thread's security information directly without calling it by using a communication mechanism that provides impersonation services.
THREAD_QUERY_INFORMATION	Required to read certain information from the thread object.
THREAD_SET_CONTEXT	Required to write the context of a thread.
THREAD_SET_INFORMATION	Required to set certain information in the thread object.
THREAD_SET_THREAD_TOKEN	Required to set the impersonation token for a thread.
THREAD_SUSPEND_RESUME	Required to suspend or resume a thread.
THREAD_TERMINATE	Required to terminate a thread.

You can request the ACCESS_SYSTEM_SECURITY access right to a thread object if you want to read or write the object's SACL. For more information, see *Access-Control Lists (ACLs)* and *SACL Access Right*.

Child Processes

A *child process* is a process that is created by another process, called the *parent process*.

Creating Processes

The **CreateProcess** function creates a new process, which runs independently of the creating process. However, for simplicity, the relationship is referred to as a parent-child relationship.

The following code fragment demonstrates how to create a process.

```
void main( VOID )
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    ZeroMemory( &si, sizeof(si) );
    si.cb = sizeof(si);

    // Start the child process.
    if( !CreateProcess( NULL, // No module name (use command line).
        "MyChildProcess", // Command line.
        NULL,             // Process handle not inheritable.
        NULL,             // Thread handle not inheritable.
        FALSE,            // Set handle inheritance to FALSE.
        0,                // No creation flags.
        NULL,             // Use parent's environment block.
        NULL,             // Use parent's starting directory.
        &si,              // Pointer to STARTUPINFO structure.
        &pi )             // Pointer to PROCESS_INFORMATION structure.
    )
    {
        ErrorExit( "CreateProcess failed." );
    }

    // Wait until child process exits.
    WaitForSingleObject( pi.hProcess, INFINITE );

    // Close process and thread handles.
    CloseHandle( pi.hProcess );
    CloseHandle( pi.hThread );
}
```

If **CreateProcess** succeeds, it returns a **PROCESS_INFORMATION** structure containing handles and identifiers for the new process and its primary thread. The thread and process handles are created with full access rights, although access can be restricted if you specify security descriptors. When you no longer need these handles, close them by using the **CloseHandle** function.

You can also create a process using the **CreateProcessAsUser** function. This function allows you to specify the security context of the user account in which the process will execute.

Setting Window Properties Using STARTUPINFO

A parent process can specify properties associated with the main window of its child process. The **CreateProcess** function takes a pointer to a **STARTUPINFO** structure as one of its parameters. Use the members of this structure to specify characteristics of the child process's main window. The **dwFlags** member contains a bit field that determines which other members of the structure are used. This allows you to specify values for any subset of the window properties. The system uses default values for the properties you do not specify. The **dwFlags** member can also force a feedback cursor to be displayed during the initialization of the new process.

For GUI processes, the **STARTUPINFO** structure specifies the default values to be used the first time the new process calls the **CreateWindow** and **ShowWindow** functions to create and display an overlapped window. The following default values can be specified:

- The width and height, in pixels, of the window created by **CreateWindow**
- The location, in screen coordinates of the window created by **CreateWindow**
- The *nCmdShow* parameter of **ShowWindow**

For console processes, use the **STARTUPINFO** structure to specify window properties only when creating a new console (either using **CreateProcess** with CREATE_NEW_CONSOLE or with the **AllocConsole** function). The **STARTUPINFO** structure can be used to specify the following console window properties:

- The size of the new console window, in character cells
- The location of the new console window, in screen coordinates
- The size, in character cells, of the new console's screen buffer
- The text and background color attributes of the new console's screen buffer
- The title of the new console's window

Process Handles and Identifiers

When a new process is created by the **CreateProcess** function, handles of the new process and its primary thread are returned. These handles are created with full access rights, and—subject to security access checking—can be used in any of the functions that accept thread or process handles. These handles can be inherited by child processes, depending on the inheritance flag specified when they are created. The handles are valid until closed, even after the process or thread they represent has been terminated.

The **CreateProcess** function also returns an identifier that uniquely identifies the process throughout the system. A process can use the **GetCurrentProcessId** function to get its own process identifier. The identifier is valid from the time the process is created until the process has been terminated.

If you have a process identifier, you can get the process handle by calling the **OpenProcess** function. **OpenProcess** enables you to specify the handle's access rights and whether it can be inherited.

A process can use the **GetCurrentProcess** function to retrieve a pseudo handle to its own process object. This pseudo handle is valid only for the calling process; it cannot be inherited or duplicated for use by other processes. To get the real handle to the process, call the **DuplicateHandle** function.

Obtaining Additional Process Information

The Win32 API provides functions for obtaining information about processes. Some of these functions can be used only for the calling process, because they do not take a process handle as a parameter. You can use functions that take a process handle to obtain information about other processes.

- To obtain the command-line string for the current process, use the **GetCommandLine** function.
- To parse a Unicode command-line string obtained from the Unicode version of **GetCommandLine**, use the **CommandLineToArgvW** function.
- To retrieve the **STARTUPINFO** structure specified when the current process was created, use the **GetStartupInfo** function.
- To obtain the version information from the executable header, use the **GetProcessVersion** function.
- To obtain the full path and file name for the executable file containing the process code, use the **GetModuleFileName** function.
- To obtain the count of handles to graphical user interface (GUI) objects in use, use the **GetGuiResources** function.
- To determine whether a process is being debugged, use the **IsDebuggerPresent** function.
- To retrieve accounting information for all I/O operations performed by the process, use the **GetProcessIoCounters** function.

Inheritance

A child process can inherit several properties and resources from its parent process. You can also prevent a child process from inheriting properties from its parent process. The following can be inherited:

- Open handles returned by the **CreateFile** function. This includes handles to files, console input buffers, console screen buffers, named pipes, serial communication devices, and mailslots.
- Open handles to process, thread, mutex, event, semaphore, named-pipe, anonymous-pipe, and file-mapping objects.
- Environment variables.
- The current directory.

- The console, unless the process is detached or a new console is created. A child console process also inherits the parent's standard handles, as well as access to the input buffer and the active screen buffer.

The child process does not inherit the following:

- Priority class.
- Handles returned by **LocalAlloc**, **GlobalAlloc**, **HeapCreate**, and **HeapAlloc**.
- Pseudo handles, as in the handles returned by the **GetCurrentProcess** or **GetCurrentThread** function. These handles are valid only for the calling process.
- DLL module handles returned by the **LoadLibrary** function.
- GDI or USER handles, such as **HBITMAP** or **HMENU**.

Inheriting Handles

To cause a handle to be inherited, you must do two things:

- Specify that the handle is to be inherited when you create, open, or duplicate the handle.
- Specify that inheritable handles are to be inherited when you call the **CreateProcess** function.

This allows a child process to inherit some of its parent's handles, but not inherit others. For example, creation functions such as **CreateProcess** and **CreateFile** take a security attributes argument that determines whether the handle can be inherited. Open functions such as **OpenMutex** and **OpenEvent** take a handle inheritance flag that determines whether the handle can be inherited. The **DuplicateHandle** function takes a handle inheritance flag that determines whether the handle can be inherited.

When a child process is created, the *fInheritHandles* parameter of **CreateProcess** determines whether the inheritable handles of the parent process are inherited by the child process. An inherited handle refers to the same object in the child process as it does in the parent process. It also has the same value and access privileges. Therefore, when one process changes the state of the object, the change affects both processes. To use a handle, the child process must retrieve the handle value and "know" the object to which it refers. Usually, the parent process communicates this information to the child process through its command line, environment block, or some form of interprocess communication.

The **DuplicateHandle** function is useful if a process has an inheritable open handle that you do not want to be inherited by the child process. In this case, use **DuplicateHandle** to open a duplicate of the handle that cannot be inherited, then use the **CloseHandle** function to close the inheritable handle. You can also use the **DuplicateHandle** function to open an inheritable duplicate of a handle that cannot be inherited.

Inheriting Environment Variables

A child process inherits the environment variables of its parent process by default. However, **CreateProcess** enables the parent process to specify a different block of environment variables. For more information, see *Environment Variables*.

Inheriting the Current Directory

The **GetCurrentDirectory** function retrieves the current directory of the calling process. A child process inherits the current directory of its parent process by default. However, **CreateProcess** enables the parent process to specify a different current directory for the child process. To change the current directory of the calling process, use the **SetCurrentDirectory** function.

Environment Variables

Every process has an environment block that contains a set of environment variables and their values. The command processor provides the **set** command to display its environment block or to create new environment variables. Programs started by the command processor inherit the command processor's environment variables.

By default, a child process inherits the environment variables of its parent process. However, you can specify a different environment for the child process by creating a new environment block and passing a pointer to it as a parameter to the **CreateProcess** function.

The **GetEnvironmentStrings** function returns a pointer to the environment block of the calling process. This should be treated as a read-only block; do not modify it directly. Instead, use the **SetEnvironmentVariable** function to change an environment variable. When you are finished with the environment block obtained from **GetEnvironmentStrings**, call the **FreeEnvironmentStrings** function to free the block.

The **GetEnvironmentVariable** function determines whether a specified variable is defined in the environment of the calling process, and, if so, what its value is.

For more information, see the examples in *Changing Environment Variables*.

Terminating a Process

A process executes until one of the following events occurs:

- Any thread of the process calls the **ExitProcess** function. This terminates all threads of the process.
- The primary thread of the process returns. The primary thread can avoid terminating other threads by explicitly calling **ExitThread** before it returns. One of the remaining threads can still call **ExitProcess** to ensure that all threads are terminated.
- The last thread of the process terminates.
- Any thread calls the **TerminateProcess** function with a handle to the process. This terminates all threads of the process, without allowing them to clean up or save data.

- For console processes, the default handler function calls **ExitProcess** when the console receives a CTRL+C or CTRL+BREAK signal. All console processes attached to the console receive these signals. Detached processes and GUI processes are not affected by CTRL+C or CTRL+BREAK signals. For more information, see **SetConsoleCtrlHandler**.
- The user shuts down the system or logs off. Use the **SetProcessShutdownParameters** function to specify shutdown parameters, such as when a process should terminate relative to the other processes in the system. The **GetProcessShutdownParameters** function retrieves the current shutdown priority of the process and other shutdown flags.

When a process is terminated, all threads of the process are terminated immediately with no chance to run additional code. This means that the process does not execute code in termination handler blocks. For more information, see *Structured Exception Handling*.

The **GetExitCodeProcess** function returns the termination status of a process. While a process is executing, its termination status is STILL_ACTIVE. When a process terminates, its termination status changes from STILL_ACTIVE to the exit code of the process. The exit code is either the value specified in the call to **ExitProcess** or **TerminateProcess**, or the value returned by the main or **WinMain** function of the process. If a process is terminated due to a fatal exception, the exit code is the value of the exception that caused the termination. In addition, this value is used as the exit code for all the threads that were executing when the exception occurred.

When a process terminates, the state of the process object becomes signaled, releasing any threads that had been waiting for the process to terminate. For more about synchronization, see *Synchronizing Execution of Multiple Threads*.

Open handles to files or other resources are closed automatically when a process terminates. However, the objects themselves exist until all open handles to them are closed. This means that an object remains valid after a process closes, if another process has a handle to it.

If a process is terminated by **ExitProcess**, the system calls the entry-point function of each attached DLL with a value indicating that the process is detaching from the DLL. DLLs are not notified when a process is terminated by **TerminateProcess**. For more information about DLLs, see *Dynamic Link Libraries*.

Warning The **TerminateProcess** function should be used only in extreme circumstances, since it does not allow threads to clean up or save data and does not notify attached DLLs.

If you need to have one process terminate another process, the following steps provide a better solution:

- Have both processes call the **RegisterWindowMessage** function to create a private message.
- One process can terminate the other process by broadcasting the private message using the **BroadcastSystemMessage** function as follows:

```
BroadcastSystemMessage(
    BSF_IGNORECURRENTTASK, // do not send message to this process
    BSM_APPLICATIONS,      // broadcast only to applications
    private message,       // message registered in previous step
    wParam,                // message-specific value
    lParam );              // message-specific value
```

- The process receiving the private message calls **ExitProcess** to terminate its execution.

Note When the system is terminating a process, it does not terminate any child processes that the process has created.

Process Times

The **GetProcessTimes** function obtains timing information for a process. It returns the process creation time, how much time the process has been executing in kernel mode, and how much time the process has been executing in user mode. These times do not include time spent executing system threads or waiting in a suspended or blocked state. If the process has exited, **GetProcessTimes** returns the process exit time.

Process Security and Access Rights

Windows NT/Windows 2000 security enables you to control access to process objects. For more information about security, see *Access-Control Model*.

You can specify a security descriptor for a process when you call the **CreateProcess**, **CreateProcessAsUser**, or **CreateProcessWithLogonW** function. To retrieve a process's security descriptor, call the **GetSecurityInfo** function. To change a process's security descriptor, call the **SetSecurityInfo** function.

The handle returned by the **CreateProcess** function has PROCESS_ ALL_ACCESS access to the process object. When you call the **OpenProcess** function, the system checks the requested access rights against the DACL in the process's security descriptor. When you call the **GetCurrentProcess** function, Windows NT returns a pseudohandle with the maximum access that the DACL allows to the caller.

The valid access rights for process objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the following process-specific access rights.

Value	Meaning
PROCESS_ALL_ACCESS	Specifies all possible access rights for a process object.
PROCESS_CREATE_PROCESS	Required to create a process.
PROCESS_CREATE_THREAD	Required to create a thread.
PROCESS_DUP_HANDLE	Required to duplicate a handle.
PROCESS_QUERY_INFORMATION	Required to retrieve certain information about a process, such as its priority class.
PROCESS_SET_QUOTA	Required to set memory limits.
PROCESS_SET_INFORMATION	Required to set certain information about a process, such as its priority class.
PROCESS_TERMINATE	Required to terminate a process.
PROCESS_VM_OPERATION	Required to perform an operation on the address space of a process.
PROCESS_VM_READ	Required to read memory in a process.
PROCESS_VM_WRITE	Required to write to memory in a process.
SYNCHRONIZE	A standard right required to wait for the process to terminate.

You can request the ACCESS_SYSTEM_SECURITY access right to a process object if you want to read or write the object's SACL. For more information, see *Access-Control Lists (ACLs)* and *SACL Access Right*.

Process Working Set

The *working set* of a program is a collection of those pages in its virtual address space that have been recently referenced. It includes both shared and private data. The shared data includes pages that contain all instructions your application executes, including those in your DLLs and the system DLLs. As the working set size increases, memory demand increases.

A process has an associated minimum working set size and maximum working set size. Each time you call **CreateProcess**, it reserves the minimum working set size for the process. The virtual memory manager attempts to keep enough memory for the minimum working set resident when the process is active, but keeps no more than the maximum size.

To get the requested minimum and maximum sizes of the working set for your application, call the **GetProcessWorkingSetSize** function.

The system sets the default working set sizes. You can also modify the working set sizes using the **SetProcessWorkingSetSize** function. Setting these values is not a guarantee

that the memory will be reserved or resident. Be careful about requesting too large a minimum or maximum working set size, because doing so can degrade system performance.

Thread Pooling

There are many applications that create threads that spend a great deal of time in the sleeping state waiting for an event to occur. Other threads may enter a sleeping state only to be awakened periodically to poll for a change or update status information. *Thread pooling* enables you to use threads more efficiently by providing your application with a pool of worker threads that are managed by the system. One thread monitors the status of all wait operations queued to the thread pool. When a wait operation has completed, a worker thread from the thread pool executes the corresponding callback function.

You can also queue work items that are not related to a wait operation to the thread pool. To request that a work item be handled by a thread in the thread pool, call the **QueueUserWorkItem** function. This function takes a parameter to the function that will be called by the thread selected from the thread pool. There is no way to cancel a work item after it has been queued.

Timer-queue timers and registered wait operations also use the thread pool. Their callback functions are queued to the thread pool. You can also use the **BindIoCompletionCallback** function to queue a callback function to a worker thread.

The thread pool is created the first time you call **QueueUserWorkItem** or **BindIoCompletionCallback**, or when a timer-queue timer or registered wait operation queues a callback function. The number of threads that can be created in the thread pool is limited only by available memory. Each thread uses the default stack size and runs at the default priority. Each thread can handle up to 63 wait operations.

There are two types of worker threads in the thread pool: I/O and non-I/O. An *I/O worker thread* is a thread that waits in an alertable wait state. Work items are queued to I/O worker threads as asynchronous procedure calls (APC). You should queue a work item to an I/O worker thread if it should be executed in a thread that does not exit if there are pending asynchronous I/O requests or in a thread that waits in an alertable state. These threads can be used by work items that initiate asynchronous I/O completion requests.

A *non-I/O worker thread* waits on I/O completion ports. Using non-I/O worker threads is more efficient than using I/O worker threads. Therefore, you should use non-I/O worker threads whenever possible.

To use thread pooling, the work items and all the functions they call must be thread pool safe. A safe function does not assume that thread executing it is a dedicated or persistent thread. In general, you should avoid thread local storage and queing asynchronous calls that require a persistent thread, such as the **RegNotifyChangeKeyValue** function. However, such functions can be queued to a persistent worker thread using **QueueUserWorkItem** with the WT_EXECUTEINPERSISTENTIOTHREAD option.

Job Objects

A *job object* allows groups of processes to be managed as a unit. Job objects are namable, securable, sharable objects that control attributes of the processes associated with them. Operations performed on the job object affect all processes associated with the job object.

To create a job object, use the **CreateJobObject** function. When the job is created, there are no associated processes. To associate a process with a job, use the **AssignProcessToJobObject** function. After you associate a process with a job, the association cannot be broken. By default, processes created by a process associated with a job (child processes) are associated with the job. If the job has the extended limit JOB_OBJECT_LIMIT_BREAKAWAY_OK and the process was created with the CREATE_BREAKAWAY_FROM_JOB flag, its child processes are not associated with the job. If the job has the extended limit JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK, no child processes are associated with the job.

A job can enforce limits on each associated process, such as the working set size, process priority, end-of-job time limit, and so on. To set limits for a job object, use the **SetInformationJobObject** function. If a process associated with a job attempts to increase its working set size or process priority, the function calls are silently ignored.

The job object also records basic accounting information for all its associated processes, including those that have terminated. To retrieve this accounting information, use the **QueryInformationJobObject** function.

To terminate all processes currently associated with a job object, use the **TerminateJobObject** function.

To close a job object handle, use the **CloseHandle** function. The job object is destroyed when its last handle has been closed. If there are running processes still associated with the job when it is destroyed, they will continue to run even after the job is destroyed.

If a tool is to manage a process tree that uses job objects, both the tool and the members of the process tree must cooperate. Use one of the following options:

- The tool could use the JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK limit. If the tool uses this limit, it cannot monitor an entire process tree. The tool can monitor only the processes it adds to the job. If these processes create child processes, they are not associated with the job. In this option, child processes can be associated with other job objects.
- The tool could use the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit. If the tool uses this limit, it can monitor the entire process tree, except for those processes that any member of the tree explicitly breaks away from the tree. A member of the tree can create a child process in a new job object by calling the **CreateProcess** function with the CREATE_BREAKAWAY_FROM_JOB flag, then calling the **AssignProcessToJobObject** function. Otherwise, the member must handle cases in which **AssignProcessToJobObject** fails.

The CREATE_BREAKAWAY_FROM_JOB flag has no effect if the tree is not being monitored by the tool. Therefore, this is the preferred option, but it requires advance knowledge of the processes being monitored.

- The tool could prevent breakaways of any kind. In this option, the tool can monitor the entire process tree. However, if a process associated with the job tries to call **AssignProcessToJobObject**, the call will fail. If the process was not designed to be associated with a job, this failure may be unexpected.

Job Object Security and Access Rights

Windows NT/Windows 2000 security enables you to control access to job objects. For more information about security, see *Access-Control Model.*

You can specify a security descriptor for a job object when you call the **CreateJobObject** function. To get or set the security descriptor for a job object, call the **GetNamedSecurityInfo**, **SetNamedSecurityInfo**, **GetSecurityInfo**, or **SetSecurityInfo** function.

The handle returned by **CreateJobObject** has JOB_OBJECT_ALL_ACCESS access to the job object. When you call the **OpenJobObject** function, the system checks the requested access rights against the object's security descriptor.

The valid access rights for job objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights, in addition to the following job-specific access rights.

Value	Meaning
JOB_OBJECT_ASSIGN_PROCESS	Required to call the **AssignProcessToJobObject** function to assign processes to the job object.
JOB_OBJECT_SET_ATTRIBUTES	Required to call the **SetInformationJobObject** function to set the attributes of the job object.
JOB_OBJECT_QUERY	Required to call the **QueryInformationJobObject** function to query job object attributes and accounting information.
JOB_OBJECT_TERMINATE	Required to call the **TerminateJobObject** function to terminate all processes in the job object.
JOB_OBJECT_SET_SECURITY_ATTRIBUTES	Required to call the **SetInformationJobObject** function with the JobObjectSecurityLimitInformation information class to set security limitations on the processes associated with the job object.
JOB_OBJECT_ALL_ACCESS	Combines all valid job object access rights.

You can request the ACCESS_SYSTEM_SECURITY access right to a job object if you want to read or write the object's SACL. For more information, see *Access-Control Lists (ACLs)* and *SACL Access Right*.

Fibers

A *fiber* is a unit of execution that must be manually scheduled by the application. Fibers run in the context of the threads that schedule them. Each thread can schedule multiple fibers. In general, fibers do not provide advantages over a well-designed multithreaded application. However, using fibers can make it easier to port applications that were designed to schedule their own threads.

From a system standpoint, a fiber assumes the identity of the thread that created it. For example, if a fiber accesses thread local storage (TLS), it is accessing the thread local storage of the thread that created it. In addition, if a fiber calls the **ExitThread** function, the thread that created it exits. However, a fiber does not have all the same state information associated with it as that associated with a thread. The only state information maintained for a fiber is its stack, a subset of its registers, and the fiber data provided during fiber creation. The saved registers are the set of registers typically preserved across a function call.

Fibers are not preemptively scheduled. You schedule a fiber by switching to it from another fiber. The system still schedules threads to run. When a thread running fibers is preempted, its currently running fiber is preempted. The fiber runs when its thread runs.

Before scheduling the first fiber, call the **ConvertThreadToFiber** function to create an area in which to save fiber state information. The calling thread is now the currently executing fiber. The stored state information for this fiber includes the fiber data passed as an argument to **ConvertThreadToFiber**.

The **CreateFiber** function is used to create a new fiber from an existing fiber; the call requires the stack size, the starting address, and the fiber data. The starting address is typically a user-supplied function, called the fiber function, that takes one parameter (the fiber data) and does not return a value. If your fiber function returns, the thread running the fiber exits. To execute any fiber created with **CreateFiber,** call the **SwitchToFiber** function. You can call **SwitchToFiber** with the address of a fiber created by a different thread. To do this, you must have the address returned to the other thread when it called **CreateFiber** and you must use proper synchronization.

A fiber can retrieve the fiber data by calling the **GetFiberData** macro. A fiber can retrieve the fiber address at any time by calling the **GetCurrentFiber** macro.

To clean up the data associated with a fiber, call the **DeleteFiber** function. You must take care when calling **DeleteFiber**. If you call **DeleteFiber** for a fiber created by another thread, you can cause the other thread to terminate abnormally. If **DeleteFiber** is called from the currently running fiber, its thread calls **ExitThread**.

Process and Thread Reference

Process and Thread Functions

AssignProcessToJobObject

The **AssignProcessToJobObject** function associates a process with an existing job object.

```
BOOL AssignProcessToJobObject(
  HANDLE hJob,      // handle to job
  HANDLE hProcess   // handle to process
);
```

Parameters

hJob
: [in] Handle to the job object to which the process will be associated. The **CreateJobObject** or **OpenJobObject** function returns this handle. The handle must have the JOB_OBJECT_ASSIGN_PROCESS access right associated with it. For more information, see *Job Object Security and Access Rights.*

hProcess
: [in] Handle to the process to associate with the job object. The process must not already be assigned to a job. The handle must have PROCESS_SET_QUOTA and PROCESS_TERMINATE access to the process. For more information, see *Process Security and Access Rights*.

  Terminal Services: All processes within a job must run within the same session.

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

After you associate a process with a job object using **AssignProcessToJobObject**, the process is subject to the limits set for the job. To set limits for a job, use the **SetInformationJobObject** function.

If the job has a user-mode time limit, and the time limit has been exhausted, **AssignProcessToJobObject** fails and the specified process is terminated. If the time limit would be exceeded by associating the process, **AssignProcessToJobObject** still succeeds. However, the time limit violation will be reported. If the job has an active

process limit, and the limit would be exceeded by associating this process, **AssignProcessToJobObject** fails, and the specified process is terminated.

Memory operations performed by a process associated with a job that has a memory limit are subject to the memory limit. Memory operations performed by the process before it was associated with the job are not examined by **AssignProcessToJobObject**.

If the process is already running and the job has security limitations, **AssignProcessToJobObject** may fail. For example, if the primary token of the process contains the local administrators group, but the job object has the security limitation JOB_OBJECT_SECURITY_NO_ADMIN, the function fails. If the job has the security limitation JOB_OBJECT_SECURITY_ONLY_TOKEN, the process must be created suspended. To create a suspended process, call the **CreateProcess** function with the CREATE_SUSPENDED flag.

A process can be associated only with a single job. A process inherits limits from the job it is associated with and adds its accounting information to the job. If a process is associated with a job, all processes it creates are associated with that job by default. To create a process that is not part of the same job, call the **CreateProcess** function with the CREATE_BREAKAWAY_FROM_JOB flag.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, *Processes and Threads Overview*, **CreateJobObject**, **CreateProcess**, **OpenJobObject**, **SetInformationJobObject**

AttachThreadInput

The **AttachThreadInput** function attaches the input processing mechanism of one thread to that of another thread.

```
BOOL AttachThreadInput(
  DWORD idAttach,     // thread to attach
  DWORD idAttachTo,   // thread to attach to
  BOOL fAttach        // attach or detach
);
```

Parameters

idAttach

[in] Specifies the identifier of the thread to be attached to another thread. The thread to be attached cannot be a system thread.

idAttachTo

[in] Specifies the identifier of the thread to be attached to. This thread cannot be a system thread.

A thread cannot attach to itself. Therefore, *idAttachTo* cannot equal *idAttach*.

fAttach

[in] Specifies whether to attach or detach the threads. If this parameter is TRUE, the two threads are attached. If the parameter is FALSE, the threads are detached.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. There is no extended error information; do not call **GetLastError**.

Remarks

Windows created in different threads typically process input independently of each other. That is, they have their own input states (focus, active, capture windows, key state, queue status, and so on), and they are not synchronized with the input processing of other threads. By using the **AttachThreadInput** function, a thread can attach its input processing to another thread. This also allows threads to share their input states, so they can call the **SetFocus** function to set the keyboard focus to a window of a different thread. This also allows threads to get key-state information. These capabilities are not generally possible.

The **AttachThreadInput** function fails if either of the specified threads 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. The **AttachThreadInput** function also fails if a journal record hook is installed. Journal record hooks attach all input queues together.

Note that key state, which can be ascertained by calls to the **GetKeyState** or **GetKeyboardState** function, is reset after a call to **AttachThreadInput**.

Windows NT/2000: You cannot attach a thread to a thread in another desktop.

Requirements

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

Processes and Threads Overview, *Process and Thread Functions*, **GetCurrentThreadId**, **GetKeyState**, **GetKeyboardState**, **GetWindowThreadProcessId**, **SetFocus**

BindIoCompletionCallback

The **BindIoCompletionCallback** function queues a callback function to a non-I/O worker thread from the thread pool.

```
BOOL BindIoCompletionCallback(
  HANDLE FileHandle,                          // handle to file
  LPOVERLAPPED_COMPLETION_ROUTINE Function,   // callback
  ULONG Flags                                 // reserved
);
```

Parameters

FileHandle

[in] Handle to a file opened for overlapped I/O completion. This handle is returned by the **CreateFile** function, with FILE_FLAG_OVERLAPPED flag.

Function

[in] Pointer to the function to be executed in a non-I/O worker thread when the I/O operation is complete. For more information about the completion routine, see **FileIOCompletionRoutine**.

Flags

Reserved; must be zero.

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

Be sure that the thread that initiates the request does not terminate before the request is completed. Also, if a function in a DLL is queued to a worker thread, be sure that the function has completed execution before the DLL is unloaded. Otherwise, the request is canceled.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **FileIOCompletionRoutine**

CommandLineToArgvW

The **CommandLineToArgvW** function parses a Unicode command-line string. It returns a pointer to a set of Unicode argument strings and a count of arguments, similar to the standard C run-time **argv** and **argc** values. The function provides a way to obtain a Unicode set of **argv** and **argc** values from a Unicode command-line string.

```
LPWSTR * CommandLineToArgvW(
  LPCWSTR lpCmdLine,  // pointer to a command-line string
  int *pNumArgs       // receives the argument count
);
```

Parameters

lpCmdLine

[in] Pointer to a null-terminated Unicode command-line string. An application will usually directly pass on the value returned by a call to the **GetCommandLineW** function.

If this parameter is the empty string, "", the function returns the path to the current executable file.

pNumArgs

[out] Pointer to an integer variable that receives the count of arguments parsed.

Return Values

If the function succeeds, the return value is a non-NULL pointer to the constructed argument list, which is an array of Unicode strings.

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

Remarks

It is the caller's responsibility to free the memory used by the argument list when it is no longer needed. To free the memory, use a single call to the **GlobalFree** function.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in shellapi.h; include windows.h.
Library: Use shell32.lib.
Unicode: Declared only as Unicode.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **GetCommandLine**, **GlobalFree**

ConvertThreadToFiber

The **ConvertTheadToFiber** function converts the current thread into a fiber. You must convert a thread into a fiber before you can schedule other fibers.

```
LPVOID ConvertThreadToFiber(
  LPVOID lpParameter  // fiber data for new fiber
);
```

Parameters

lpParameter

[in] Specifies a single variable that is passed to the fiber. The fiber can retrieve this value by using the **GetFiberData** macro.

Return Values

If the function succeeds, the return value is the address of the fiber.

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

Remarks

Only fibers can execute other fibers. If a thread needs to execute a fiber, it must call **ConvertThreadToFiber** to create an area in which to save fiber state information. The thread is now the current fiber. The state information for this fiber includes the fiber data specified by *lpParameter*.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **GetFiberData**

CreateFiber

The **CreateFiber** function allocates a fiber object, assigns it a stack, and sets up execution to begin at the specified start address, typically the fiber function. This function does not schedule the fiber.

```
LPVOID CreateFiber(
  DWORD dwStackSize,                        // initial stack size
  LPFIBER_START_ROUTINE lpStartAddress,     // fiber function
  LPVOID lpParameter                        // fiber argument
);
```

Parameters

dwStackSize

[in] Specifies the size, in bytes, of the stack for the new fiber. If zero is specified, the stack size defaults to the same size as that of the main thread. The function fails if it cannot commit *dwStackSize* bytes. Note that the system increases the stack size dynamically, if necessary. For more information, see *Thread Stack Size*.

lpStartAddress

[in] Pointer to the application-defined function of type LPFIBER_START_ROUTINE to be executed by the fiber and represents the starting address of the fiber. Execution of the newly created fiber does not begin until another fiber calls the **SwitchToFiber** function with this address. For more information of the fiber callback function, see **FiberProc**.

lpParameter

[in] Specifies a single argument that is passed to the fiber. This value can be retrieved by the fiber using the **GetFiberData** macro.

Return Values

If the function succeeds, the return value is the address of the fiber.

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

Remarks

Before a thread can schedule a fiber using the **SwitchToFiber** function, it must call the **ConvertThreadToFiber** function so there is a fiber associated with the thread.

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **ConvertThreadToFiber**, **FiberProc**, **GetFiberData**, **SwitchToFiber**

CreateJobObject

The **CreateJobObject** function creates or opens a job object.

```
HANDLE CreateJobObject(
  LPSECURITY_ATTRIBUTES lpJobAttributes,  // SD
  LPCTSTR lpName                          // job name
);
```

Parameters

lpJobAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies the security descriptor for the job object and determines whether child processes can inherit the returned handle. If *lpJobAttributes* is NULL, the job object gets a default security descriptor and the handle cannot be inherited.

lpName

[in] Pointer to a null-terminated string specifying the name of the job. The name is limited to MAX_PATH characters. Name comparison is case-sensitive.

If *lpName* is NULL, the job is created without a name.

If *lpName* matches the name of an existing event, semaphore, mutex, waitable timer, or file-mapping object, the function fails and the **GetLastError** function returns ERROR_INVALID_HANDLE. This occurs because these objects share the same name space.

Terminal Services: The name can have a "Global\" or "Local\" prefix to explicitly create the object in the global or session name space. The remainder of the name can contain any character except the backslash character (\). For more information, see *Kernel Object Name Spaces*.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The remainder of the name can contain any character except the backslash character.

Return Values

If the function succeeds, the return value is a handle to the job object. The handle has JOB_OBJECT_ALL_ACCESS access to the job object. If the object existed before the function call, the function returns a handle to the existing job object and **GetLastError** returns ERROR_ALREADY_EXISTS.

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

Remarks

When a job is created, its accounting information is initialized to zero, all limits are inactive, and there are no associated processes. To associate a process with a job, use the **AssignProcessToJobObject** function. To set limits for a job, use the **SetInformationJobObject** function. To query accounting information, use the **QueryInformationJobObject** function.

To close a job object handle, use the **CloseHandle** function. The job is destroyed when its last handle has been closed. If there are running processes still associated with the job when it is destroyed, they will continue to run even after the job is destroyed.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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.

Processes and Threads Overview, *Process and Thread Functions*, **AssignProcessToJobObject**, **CloseHandle**, **QueryInformationJobObject**, **SECURITY_ATTRIBUTES**, **SetInformationJobObject**

CreateProcess

The **CreateProcess** function creates a new process and its primary thread. The new process runs the specified executable file.

To create a process that runs in a different security context, use the **CreateProcessAsUser** or **CreateProcessWithLogonW** function.

```
BOOL CreateProcess(
  LPCTSTR lpApplicationName,                 // name of executable module
  LPTSTR lpCommandLine,                      // command line string
```

```
  LPSECURITY_ATTRIBUTES lpProcessAttributes, // SD
  LPSECURITY_ATTRIBUTES lpThreadAttributes,  // SD
  BOOL bInheritHandles,                      // handle inheritance option
  DWORD dwCreationFlags,                     // creation flags
  LPVOID lpEnvironment,                      // new environment block
  LPCTSTR lpCurrentDirectory,                // current directory name
  LPSTARTUPINFO lpStartupInfo,               // startup information
  LPPROCESS_INFORMATION lpProcessInformation // process information
);
```

Parameters

lpApplicationName

[in] Pointer to a null-terminated string that specifies the module to execute.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path.

The *lpApplicationName* parameter can be NULL. In that case, the module name must be the first white-space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe files\sub dir\program name
c:\program files\sub.exe dir\program name
c:\program files\sub dir\program.exe name
c:\program files\sub dir\program name.exe

The specified module can be a Win32-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer.

Windows NT/2000: If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments. A 16-bit application is one that executes as a VDM or WOW process.

lpCommandLine

[in] Pointer to a null-terminated string that specifies the command line to execute. The system adds a null character to the command line, trimming the string if necessary, to indicate which file was actually used.

Windows NT/2000: The Unicode version of this function, **CreateProcessW**, will fail if this parameter is a const string.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use **GetCommandLine** to retrieve the entire command line. C runtime processes can use the **argc** and **argv** arguments. Note that it is a common practice to repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white-space-delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory for the parent process.
3. **Windows 95/98:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is System32.
4. **Windows NT/2000:** The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

lpProcessAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpProcessAttributes* is NULL, the handle cannot be inherited.

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new process. If *lpProcessAttributes* is NULL, the process gets a default security descriptor.

lpThreadAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the main thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor.

bInheritHandles

[in] Indicates whether the new process inherits handles from the calling process. If TRUE, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

dwCreationFlags

[in] Specifies additional flags that control the priority class and the creation of the process. The following creation flags can be specified in any combination, except as noted.

Value	Meaning
CREATE_BREAKAWAY_FROM_JOB	**Windows 2000:** The child processes of a process associated with a job are not associated with the job. If the calling process is not associated with a job, this flag has no effect. If the calling process is associated with a job, the job must set the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit or **CreateProcess** will fail.
CREATE_DEFAULT_ERROR_MODE	The new process does not inherit the error mode of the calling process. Instead, **CreateProcess** gives the new process the current default error mode. An application sets the current default error mode by calling **SetErrorMode**. This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for **CreateProcess** is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.
CREATE_FORCE_DOS	**Windows NT/2000:** This flag is valid only when starting a 16-bit bound application. If set, the system will force the application to run as an MS-DOS-based application rather than as an OS/2-based application.
CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the DETACHED_PROCESS flag.

CREATE_NEW_PROCESS_GROUP	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the *lpProcessInformation* parameter. Process groups are used by the **GenerateConsoleCtrlEvent** function to enable sending a CTRL+C or CTRL+BREAK signal to a group of console processes.
CREATE_NO_WINDOW	**Windows NT/2000:** This flag is valid only when starting a console application. If set, the console application is run without a console window.
CREATE_SEPARATE_WOW_VDM	**Windows NT/2000:** This flag is valid only when starting a 16-bit Windows-based application. If set, the new process runs in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based applications run as threads in a single, shared VDM. The advantage of running separately is that a crash only terminates the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows-based applications that are run in separate VDMs have separate input queues. That means that if one application stops responding momentarily, applications in separate VDMs continue to receive input. The disadvantage of running separately is that it takes significantly more memory to do so. You should use this flag only if the user requests that 16-bit applications should run in them own VDM.
CREATE_SHARED_WOW_VDM	**Windows NT/2000:** The flag is valid only when starting a 16-bit Windows-based application. If the DefaultSeparateVDM switch in the Windows section of WIN.INI is TRUE, this flag causes the **CreateProcess** function to override the switch and run the new process in the shared Virtual DOS Machine.
CREATE_SUSPENDED	The primary thread of the new process is created in a suspended state, and does not run until the **ResumeThread** function is called.
CREATE_UNICODE_ENVIRONMENT	Indicates the format of the *lpEnvironment* parameter. If this flag is set, the environment block pointed to by *lpEnvironment* uses Unicode characters. Otherwise, the environment block uses ANSI characters.

DEBUG_PROCESS	If this flag is set, the calling process is treated as a debugger, and the new process is debugged. The system notifies the debugger of all debug events that occur in the process being debugged. If you create a process with this flag set, only the calling thread (the thread that called **CreateProcess**) can call the **WaitForDebugEvent** function. **Windows 95/98:** This flag is not valid if the new process is a 16-bit application.
DEBUG_ONLY_THIS_PROCESS	If this flag is not set and the calling process is being debugged, the new process becomes another process being debugged by the calling process's debugger. If the calling process is not a process being debugged, no debugging-related actions occur.
DETACHED_PROCESS	For console processes, the new process does not have access to the console of the parent process. The new process can call the **AllocConsole** function at a later time to create a new console. This flag cannot be used with the CREATE_NEW_CONSOLE flag.

The *dwCreationFlags* parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. If none of the following priority class flags is specified, the priority class defaults to NORMAL_PRIORITY_CLASS unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process. You can specify one of the following values:

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than NORMAL_PRIORITY_CLASS but lower than HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than IDLE_PRIORITY_CLASS but lower than NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the system. Use extreme care when using the high-priority class, because a CPU-bound application with a high-priority class can use nearly all available cycles.

IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

lpEnvironment

[in] Pointer to an environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the form:

```
name=value
```

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

If an application provides an environment block, rather than passing NULL for this parameter, the current directory information of the system drives is not automatically propagated to the new process. For a discussion of this situation and how to handle it, see the following Remarks section.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, set the *dwCreationFlags* field's CREATE_UNICODE_ENVIRONMENT flag. Otherwise, do not set this flag.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

lpCurrentDirectory

[in] Pointer to a null-terminated string that specifies the current drive and directory for the child process. The string must be a full path and file name that includes a drive letter. If this parameter is NULL, the new process will have the same current drive and directory as the calling process. This option is provided primarily for shells that need to start an application and specify its initial drive and working directory.

lpStartupInfo
[in] Pointer to a **STARTUPINFO** structure that specifies how the main window for the new process should appear.

lpProcessInformation
[out] Pointer to a **PROCESS_INFORMATION** structure that receives identification information about the new process.

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 **CreateProcess** function is used to run a new program. The **WinExec** and **LoadModule** functions are still available, but they are implemented as calls to **CreateProcess**.

In addition to creating a process, **CreateProcess** also creates a thread object. The thread is created with an initial stack whose size is described in the image header of the specified program's executable file. The thread begins execution at the image's entry point.

When created, the new process and the new thread handles receive full access rights. For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle to that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the thread.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the **OpenProcess** function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the **PROCESS_INFORMATION** structure.

When specifying an application name in the *lpApplicationName* or *lpCommandLine* strings, it doesn't matter whether the application name includes the file name extension, with one exception: an MS-DOS-based or Windows-based application whose file name extension is .com must include the .com extension.

The calling thread can use the **WaitForInputIdle** function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcess** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the **ExitProcess** function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs).

ExitProcess, **ExitThread**, **CreateThread**, **CreateRemoteThread**, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events at a time can happen in an address space, and the following restrictions apply.

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is finished for the process.
- Only one thread at a time can be in a DLL initialization or detach routine.
- The **ExitProcess** function does not return until there are no threads are in their DLL initialization or detach routines.

The created process remains in the system until all threads within the process have terminated and all handles to the process and any of its threads have been closed through calls to **CloseHandle**. The handles for both the process and the main thread must be closed through calls to **CloseHandle**. If these handles are not needed, it is best to close them immediately after the process is created.

When the last thread in a process terminates, the following events occur:

- All objects opened by the process are implicitly closed.
- The process's termination status (which is returned by **GetExitCodeProcess**) changes from its initial value of STILL_ACTIVE to the termination status of the last thread to terminate.
- The thread object of the main thread is set to the signaled state, satisfying any threads that were waiting on the object.
- The process object is set to the signaled state, satisfying any threads that were waiting on the object.

If the current directory on drive C is \MSVC\MFC, there is an environment variable called =C: whose value is C:\MSVC\MFC. As noted in the previous description of *lpEnvironment*, such current directory information for a system's drives does not automatically propagate to a new process when the **CreateProcess** function's *lpEnvironment* parameter is non-NULL. An application must manually pass the current directory information to the new process. To do so, the application must explicitly create the =X environment variable strings, get them into alphabetical order (because the system uses a sorted environment), and then put them into the environment block specified by *lpEnvironment*. Typically, they will go at the front of the environment block, due to the previously mentioned environment block sorting.

One way to obtain the current directory variable for a drive X is to call **GetFullPathName(**"X:",..**)**. That avoids an application having to scan the environment block. If the full path returned is X:\, there is no need to pass that value on as environment data, since the root directory is the default current directory for drive X of a new process.

The handle returned by the **CreateProcess** function has PROCESS_ALL_ACCESS access to the process object.

The current directory specified by the *lpcurrentDirectory* parameter is the current directory for the child process. The current directory specified in item 2 under the *lpCommandLine* parameter is the current directory for the parent process.

Note The command line that the operating system provides to a process is not necessarily identical to the command line that the calling process gives to the **CreateProcess** function.

Windows NT/2000: When a process is created with CREATE_NEW_PROCESS_GROUP specified, an implicit call to **SetConsoleCtrlHandler**(NULL,TRUE) is made on behalf of the new process; this means that the new process has CTRL+C disabled. This lets good shells handle CTRL+C themselves, and selectively pass that signal on to sub-processes. CTRL+BREAK is not disabled, and may be used to interrupt the process/process 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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **AllocConsole**, **CloseHandle**, **CreateProcessAsUser**, **CreateProcessWithLogonW**, **CreateRemoteThread**, **CreateThread**, **ExitProcess**, **ExitThread**, **GenerateConsoleCtrlEvent**, **GetCommandLine**, **GetEnvironmentStrings**, **GetExitCodeProcess**, **GetFullPathName**, **GetStartupInfo**, **GetSystemDirectory**, **GetWindowsDirectory**, **LoadModule**, **OpenProcess**, **PROCESS_INFORMATION**, **ResumeThread**, **SECURITY_ATTRIBUTES**, **SetConsoleCtrlHandler**, **SetErrorMode**, **STARTUPINFO**, **TerminateProcess**, **WaitForInputIdle**, **WaitForDebugEvent**, **WinExec**

CreateProcessAsUser

The **CreateProcessAsUser** function creates a new process and its primary thread. The new process then runs a specified executable file. The **CreateProcessAsUser** function is similar to the **CreateProcess** function, except that the new process runs in the security context of the user represented by the *hToken* parameter. By default, the new process is noninteractive, that is, it runs on a desktop that is not visible and cannot receive user input. Also, by default, the new process inherits the environment of the calling process, rather than the environment associated with the specified user.

The **CreateProcessWithLogonW** function is similar to **CreateProcessAsUser**, except that the caller does not need to call the **LogonUser** function to authenticate the user and get a token.

This function is also similar to the **SHCreateProcessAsUser** function.

```
BOOL CreateProcessAsUser(
  HANDLE hToken,                               // handle to user token
  LPCTSTR lpApplicationName,                   // name of executable module
  LPTSTR lpCommandLine,                        // command-line string
  LPSECURITY_ATTRIBUTES lpProcessAttributes,   // SD
  LPSECURITY_ATTRIBUTES lpThreadAttributes,    // SD
  BOOL bInheritHandles,                        // inheritance option
  DWORD dwCreationFlags,                       // creation flags
  LPVOID lpEnvironment,                        // new environment block
  LPCTSTR lpCurrentDirectory,                  // current directory name
  LPSTARTUPINFO lpStartupInfo,                 // startup information
  LPPROCESS_INFORMATION lpProcessInformation   // process information
);
```

Parameters

hToken

[in] Handle to a primary token that represents a user. The handle must have TOKEN_QUERY, TOKEN_DUPLICATE, and TOKEN_ASSIGN_PRIMARY access. For more information, see *Access Rights for Access-Token Objects*. The user represented by the token must have read and execute access to the application specified by the *lpApplicationName* or the *lpCommandLine* parameter.

If your process has the SE_TCB_NAME privilege, it can call the **LogonUser** function to get a primary token that represents a specified user.

Alternatively, you can call the **DuplicateTokenEx** function to convert an impersonation token into a primary token. This allows a server application that is impersonating a client to create a process that has the security context of the client.

Terminal Services: The process is run in the session specified in this token.

lpApplicationName

[in] Pointer to a null-terminated string that specifies the module to execute.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path.

This parameter can be NULL. In that case, the module name must be the first white-space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe files\sub dir\program name
c:\program files\sub.exe dir\program name
c:\program files\sub dir\program.exe name
c:\program files\sub dir\program name.exe

The specified module can be a Win32-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer. If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments. By default, all 16-bit Windows-based applications created by **CreateProcessAsUser** are run in a separate VDM (equivalent to CREATE_SEPARATE_WOW_VDM in **CreateProcess**).

lpCommandLine

[in] Pointer to a null-terminated string that specifies the command line to execute. The system adds a null character to the command line, trimming the string if necessary, to indicate which file was actually used.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use **GetCommandLine** to retrieve the entire command line. C runtime processes can use the **argc** and **argv** arguments. Note that it is a common practice to repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white-space-delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory.

3. The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is System32.
4. The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

lpProcessAttributes
[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new process and determines whether child processes can inherit the returned handle. If *lpProcessAttributes* is NULL, the process gets a default security descriptor and the handle cannot be inherited.

lpThreadAttributes
[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new process and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited.

bInheritHandles
[in] Indicates whether the new process inherits handles from the calling process. If TRUE, each inheritable open handle in the calling process is inherited by the new process. Inherited handles have the same value and access privileges as the original handles.

dwCreationFlags
[in] Specifies additional flags that control the priority class and the creation of the process. The following creation flags can be specified in any combination, except as noted.

Value	Meaning
CREATE_BREAKAWAY_FROM_JOB	**Windows 2000:** The child processes of a process associated with a job are not associated with the job. If the calling process is not associated with a job, this flag has no effect. If the calling process is associated with a job, the job must set the JOB_OBJECT_LIMIT_BREAKAWAY_OK limit or **CreateProcess** will fail.

CREATE_DEFAULT_ERROR_MODE	The new process does not inherit the error mode of the calling process. Instead, **CreateProcessAsUser** gives the new process the current default error mode. An application sets the current default error mode by calling **SetErrorMode**. This flag is particularly useful for multi-threaded shell applications that run with hard errors disabled. The default behavior for **CreateProcessAsUser** is for the new process to inherit the error mode of the caller. Setting this flag changes that default behavior.
CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the DETACHED_PROCESS flag.
CREATE_NEW_PROCESS_GROUP	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the *lpProcessInformation* parameter. Process groups are used by the **GenerateConsoleCtrlEvent** function to enable sending a CTRL+C or CTRL+BREAK signal to a group of console processes.
CREATE_SUSPENDED	The primary thread of the new process is created in a suspended state, and does not run until the **ResumeThread** function is called.
CREATE_UNICODE_ENVIRONMENT	Indicates the format of the *lpEnvironment* parameter. If this flag is set, the environment block pointed to by *lpEnvironment* uses Unicode characters. Otherwise, the environment block uses ANSI characters.
DEBUG_ONLY_THIS_PROCESS	If this flag is not set and the calling process is being debugged, the new process becomes another process being debugged by the calling process's debugger. If the calling process is not a process being debugged, no debugging-related actions occur.
DEBUG_PROCESS	If this flag is set, the calling process is treated as a debugger, and the new process is debugged. The system notifies the debugger of all debug events that occur in the process being debugged.
DETACHED_PROCESS	For console processes, the new process does not have access to the console of the parent process. The new process can call the **AllocConsole** function later to create a new console. This flag cannot be used with the CREATE_NEW_CONSOLE flag.

The *dwCreationFlags* parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. If none of the following priority class flags is specified, the priority class is NORMAL_PRIORITY_CLASS by default unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process. One of the following flags can be specified.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than NORMAL_PRIORITY_CLASS but lower than HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than IDLE_PRIORITY_CLASS but lower than NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a CPU-bound application with a high-priority class can use nearly all available cycles.
IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

lpEnvironment

[in] Pointer to an environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process.

If an application provides an environment block, rather than passing NULL for this parameter, the current directory information of the system drives is not automatically propagated to the new process. For a discussion of this situation and how to handle it, see the following Remarks section.

An environment block can contain either Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, set the *dwCreationFlags* field's CREATE_UNICODE_ENVIRONMENT flag. Otherwise, do not set this flag.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

To retrieve a copy of the environment block for a given user, use the **CreateEnvironmentBlock** function.

lpCurrentDirectory

[in] Pointer to a null-terminated string that specifies the current drive and directory for the new process. The string must be a full path and file name that includes a drive letter. If this parameter is NULL, the new process will have the same current drive and directory as the calling process. This option is provided primarily for shells that need to start an application and specify its initial drive and working directory.

lpStartupInfo

[in] Pointer to a **STARTUPINFO** structure that specifies how the main window for the new process should appear.

lpProcessInformation

[out] Pointer to a **PROCESS_INFORMATION** structure that receives identification information about the new process.

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

Typically, the process that calls the **CreateProcessAsUser** function must have the SE_ASSIGNPRIMARYTOKEN_NAME and SE_INCREASE_QUOTA_NAME privileges. However, if *hToken* is a restricted version of the caller's primary token, the SE_ASSIGNPRIMARYTOKEN_NAME privilege is not required. If the necessary privileges are not already enabled, **CreateProcessAsUser** enables them for the duration of the call.

CreateProcessAsUser must be able to open the primary token of the calling process for TOKEN_DUPLICATE and TOKEN_IMPERSONATE access.

By default, **CreateProcessAsUser** creates the new process on a noninteractive window station with a desktop that is not visible and cannot receive user input. To enable user

interaction with the new process, you must specify the name of the default interactive window station and desktop, "winsta0\default", in the **lpDesktop** member of the **STARTUPINFO** structure. In addition, before calling **CreateProcessAsUser**, you must change the discretionary access control list (DACL) of both the default interactive window station and the default desktop. The DACLs for the window station and desktop must grant access to the user or the logon session represented by the *hToken* parameter.

CreateProcessAsUser does not load the specified user's profile into the HKEY_USERS registry key. This means that access to information in the HKEY_CURRENT_USER registry key may not produce results consistent with a normal interactive logon. It is your responsibility to load the user's registry hive into HKEY_USERS with the **LoadUserProfile** function before calling **CreateProcessAsUser**.

If the *lpEnvironment* parameter is NULL, the new process inherits the environment of the calling process. **CreateProcessAsUser** does not automatically modify the environment block to include environment variables specific to the user represented by *hToken*. For example, the USERNAME and USERDOMAIN variables are inherited from the calling process if *lpEnvironment* is NULL. It is your responsibility to prepare the environment block for the new process and specify it in *lpEnvironment*.

CreateProcessAsUser allows you to access the specified directory and executable image in the security context of the caller or the target user. By default, **CreateProcessAsUser** accesses the directory and executable image in the security context of the caller. In this case, if the caller does not have access to the directory and executable image, the function fails. To access the directory and executable image using the security context of the target user, specify *hToken* in a call to the **ImpersonateLoggedOnUser** function before calling **CreateProcessAsUser**.

When created, the new process and the new thread handles receive full access rights (PROCESS_ALL_ACCESS and THREAD_ALL_ACCESS). For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process is not able to use the handle to gain access to the process or thread.

If the *lpProcessAttributes* parameter is NULL, the default security descriptor for the user referenced in the *hToken* parameter will be used. This security descriptor may not allow access for the caller, in which case the process may not be opened again once it is run. The handle returned in the **PROCESS_INFORMATION** structure is valid and will continue to have full access rights. This is also true for thread attributes.

Handles in **PROCESS_INFORMATION** must be closed with **CloseHandle** when they are no longer needed.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the **OpenProcess** function to open a handle to the process. The initial thread in the process is also

assigned a thread identifier. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the **PROCESS_INFORMATION** structure.

When specifying an application name in the *lpApplicationName* or *lpCommandLine* strings, it doesn't matter whether the application name includes the file name extension, with one exception: an MS-DOS-based or Windows-based application whose file name extension is .com must include the .com extension.

The calling thread can use the **WaitForInputIdle** function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcessAsUser** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the **ExitProcess** function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated without an opportunity to execute any additional code (including the thread termination code of attached DLLs).

ExitProcess, **ExitThread**, **CreateThread**, **CreateRemoteThread**, and a process that is starting (as the result of a call by **CreateProcessAsUser**) are serialized between each other within a process. Only one of these events can happen at a time, and the following restrictions apply:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is finished for the process.
- Only one thread at a time can be in a DLL initialization or detach routine.
- The **ExitProcess** function does not return until there are no threads executing DLL initialization or detach routines.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use advapi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **CreateEnvironmentBlock**, **CreateProcess**, **CreateProcessWithLogonW**, **CreateRemoteThread**, **CreateThread**, **ExitProcess**, **ExitThread**,

GetEnvironmentStrings, **GetExitCodeProcess**, **GetStartupInfo**, **ImpersonateLoggedOnUser**, **PROCESS_INFORMATION**, **SECURITY_ATTRIBUTES**, **SetErrorMode**, **SHCreateProcessAsUser**, **STARTUPINFO**, **WaitForInputIdle**

CreateProcessWithLogonW

The **CreateProcessWithLogonW** function creates a new process and its primary thread. The new process then runs the specified executable file in the security context of the specified credentials (user, domain, and password). It can optionally load the user profile of the specified user.

The **CreateProcessWithLogonW** function is similar to the **CreateProcessAsUser** function, except that the caller does not need to call the **LogonUser** function to authenticate the user and get a token.

```
BOOL CreateProcessWithLogonW(
  LPCWSTR lpUsername,                 // user's name
  LPCWSTR lpDomain,                   // user's domain
  LPCWSTR lpPassword,                 // user's password
  DWORD dwLogonFlags,                 // logon option
  LPCWSTR lpApplicationName,          // executable module name
  LPWSTR lpCommandLine,               // command-line string
  DWORD dwCreationFlags,              // creation flags
  LPVOID lpEnvironment,               // new environment block
  LPCWSTR lpCurrentDirectory,         // current directory name
  LPSTARTUPINFOW lpStartupInfo,       // startup information
  LPPROCESS_INFORMATION lpProcessInfo // process information
);
```

Parameters

lpUsername

[in] Pointer to a null-terminated string that specifies the name of the user. This is the name of the user account to log on to. If you use the format *user@DNS_domain_name*, the *lpDomain* parameter should be NULL.

The user account must have Log On Locally permission on the local computer. This permission is granted to all users on workstations and servers, but only to administrators on domain controllers.

lpDomain

[in] Pointer to a null-terminated string that specifies the name of the domain or server whose account database contains the *lpUsername* account.

If this parameter is NULL, **CreateProcessWithLogonW** attempts to validate the account using the local account database. If **CreateProcessWithLogonW** cannot find the account in the local account database, its trusted domains search their account databases until a match is found. Note that this parameter cannot be NULL unless you specify the user name in UPN format.

lpPassword

[in] Pointer to a null-terminated string that specifies the clear-text password for the *lpUsername* account.

dwLogonFlags

[in] Specifies the logon option. This parameter can be one of the following values.

Value	Meaning
LOGON_WITH_PROFILE	Log on with profile.
LOGON_NETCREDENTIALS_ONLY	Log on with only network credentials.

lpApplicationName

[in] Pointer to a null-terminated string that specifies the module to execute.

The string can specify the full path and file name of the module to execute or it can specify a partial name. In the case of a partial name, the function uses the current drive and current directory to complete the specification. The function will not use the search path.

This parameter can be NULL. In that case, the module name must be the first white-space-delimited token in the *lpCommandLine* string. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin; otherwise, the file name is ambiguous. For example, consider the string "c:\program files\sub dir\program name". This string can be interpreted in a number of ways. The system tries to interpret the possibilities in the following order:

c:\program.exe files\sub dir\program name
c:\program files\sub.exe dir\program name
c:\program files\sub dir\program.exe name
c:\program files\sub dir\program name.exe

The specified module can be a Win32-based application. It can be some other type of module (for example, MS-DOS or OS/2) if the appropriate subsystem is available on the local computer. If the executable module is a 16-bit application, *lpApplicationName* should be NULL, and the string pointed to by *lpCommandLine* should specify the executable module as well as its arguments. A 16-bit application is one that executes as a VDM or WOW process.

lpCommandLine

[in] Pointer to a null-terminated string that specifies the command line to execute. The system adds a null character to the command line, trimming the string if necessary, to indicate which file was actually used.

The *lpCommandLine* parameter can be NULL. In that case, the function uses the string pointed to by *lpApplicationName* as the command line.

If both *lpApplicationName* and *lpCommandLine* are non-NULL, **lpApplicationName* specifies the module to execute, and **lpCommandLine* specifies the command line. The new process can use **GetCommandLine** to retrieve the entire command line. C runtime processes can use the **argc** and **argv** arguments. Note that it is a common practice to repeat the module name as the first token in the command line.

If *lpApplicationName* is NULL, the first white-space-delimited token of the command line specifies the module name. If you are using a long file name that contains a space, use quoted strings to indicate where the file name ends and the arguments begin (see the explanation for the *lpApplicationName* parameter). If the file name does not contain an extension, .exe is appended. If the file name ends in a period (.) with no extension, or if the file name contains a path, .exe is not appended. If the file name does not contain a directory path, the system searches for the executable file in the following sequence:

1. The directory from which the application loaded.
2. The current directory.
3. The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is System32.
4. The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is System.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

dwCreationFlags

[in] Specifies how the process is created. The CREATE_DEFAULT_ERROR_MODE, CREATE_NEW_CONSOLE, and CREATE_NEW_PROCESS_GROUP flags are enabled by default. You can specify additional flags as noted.

Value	Meaning
CREATE_DEFAULT_ERROR_MODE	The new process does not inherit the error mode of the calling process. Instead, **CreateProcessWithLogonW** gives the new process the current default error mode. An application sets the current default error mode by calling **SetErrorMode**. This flag is enabled by default.
CREATE_NEW_CONSOLE	The new process has a new console, instead of inheriting the parent's console. This flag cannot be used with the DETACHED_PROCESS flag. This flag is enabled by default.

CREATE_NEW_PROCESS_GROUP	The new process is the root process of a new process group. The process group includes all processes that are descendants of this root process. The process identifier of the new process group is the same as the process identifier, which is returned in the *lpProcessInfo* parameter. Process groups are used by the **GenerateConsoleCtrlEvent** function to enable sending a CTRL+C or CTRL+BREAK signal to a group of console processes. This flag is enabled by default.
CREATE_SEPARATE_WOW_VDM	This flag is only valid starting a 16-bit Windows-based application. If set, the new process runs in a private Virtual DOS Machine (VDM). By default, all 16-bit Windows-based applications run in a single, shared VDM. The advantage of running separately is that a crash only terminates the single VDM; any other programs running in distinct VDMs continue to function normally. Also, 16-bit Windows-based applications that run in separate VDMs have separate input queues. That means that if one application stops responding momentarily, applications in separate VDMs continue to receive input.
CREATE_SUSPENDED	The primary thread of the new process is created in a suspended state, and does not run until the **ResumeThread** function is called.
CREATE_UNICODE_ENVIRONMENT	Indicates the format of the *lpEnvironment* parameter. If this flag is set, the environment block pointed to by *lpEnvironment* uses Unicode characters. Otherwise, the environment block uses ANSI characters.
CREATE_WITH_USERPROFILE	If this flag is set, the system loads the user's profile after the logon succeeds. Loading the profile can be time-consuming, so it is best to use this flag only if you must access the user's profile information.

The *dwCreationFlags* parameter also controls the new process's priority class, which is used to determine the scheduling priorities of the process's threads. If none of the following priority class flags is specified, the priority class is NORMAL_PRIORITY_CLASS by default unless the priority class of the creating process is IDLE_PRIORITY_CLASS or BELOW_NORMAL_PRIORITY_CLASS. In this case, the child process receives the default priority class of the calling process. One of the following flags can be specified:

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than NORMAL_PRIORITY_CLASS but lower than HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority higher than IDLE_PRIORITY_CLASS but lower than NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks. The threads of a high-priority class process preempt the threads of normal-priority or idle-priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a CPU-bound application with a high-priority class can use nearly all available cycles.
IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

lpEnvironment

[in] Pointer to an environment block for the new process. If this parameter is NULL, the new process uses the environment of the calling process.

An environment block consists of a null-terminated block of null-terminated strings. Each string is in the form:

```
name=value
```

Because the equal sign is used as a separator, it must not be used in the name of an environment variable.

If an application provides an environment block, rather than passing NULL for this parameter, the current directory information of the system drives is not automatically

propagated to the new process. For a discussion of this situation and how to handle it, see the following Remarks section.

An environment block can contain Unicode or ANSI characters. If the environment block pointed to by *lpEnvironment* contains Unicode characters, set the *dwCreationFlags* parameter's CREATE_UNICODE_ENVIRONMENT flag. Otherwise, do not set this flag.

Note that an ANSI environment block is terminated by two zero bytes: one for the last string, one more to terminate the block. A Unicode environment block is terminated by four zero bytes: two for the last string, two more to terminate the block.

To retrieve a copy of the environment block for a given user, use the **CreateEnvironmentBlock** function.

lpCurrentDirectory

[in] Pointer to a null-terminated string that specifies the current drive and directory for the new process. The string must be a full path and file name that includes a drive letter. If this parameter is NULL, the new process has the same current drive as the system service that creates the process. (This option is provided primarily for shells that need to start an application and specify its initial drive and working directory.)

lpStartupInfo

[in] Pointer to a **STARTUPINFO** structure that specifies how the main window for the new process should appear.

lpProcessInfo

[out] Pointer to a **PROCESS_INFORMATION** structure that receives identification information about the new process.

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

By default, **CreateProcessWithLogonW** creates the new process on a noninteractive window station with a desktop that is not visible and cannot receive user input. To enable user interaction with the new process, you must specify the name of the default interactive window station and desktop, "winsta0\default", in the **lpDesktop** member of the **STARTUPINFO** structure. In addition, before calling **CreateProcessWithLogonW**, you must change the discretionary access control list (DACL) of both the default interactive window station and the default desktop. The DACLs for the window station and desktop must grant access to the user represented by the *lpUsername* parameter.

CreateProcessWithLogonW does not load the specified user's profile into the **HKEY_USERS** registry key. This means that access to information in the **HKEY_CURRENT_USER** registry key may not produce results consistent with a normal interactive logon. It is your responsibility to load the user's registry hive into

HKEY_USERS, using the **LoadUserProfile** function, before calling **CreateProcessWithLogonW**.

If the *lpEnvironment* parameter is NULL, the new process inherits the environment of the calling process. **CreateProcessWithLogonW** does not automatically modify the environment block to include environment variables specific to the user. For example, the USERNAME and USERDOMAIN variables are inherited from the calling process if *lpEnvironment* is NULL. It is your responsibility to prepare the environment block for the new process and specify it in *lpEnvironment*.

When created, the new process and the new thread handles receive full access rights (PROCESS_ALL_ACCESS and THREAD_ALL_ACCESS). For either handle, if a security descriptor is not provided, the handle can be used in any function that requires an object handle of that type. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If access is denied, the requesting process cannot use the handle to gain access to the process or thread.

If the *lpProcessAttributes* parameter is NULL, the function uses the default security descriptor for the user. This security descriptor may not allow access for the caller, in which case the process may not be opened again after it is run. The handle returned in the **PROCESS_INFORMATION** structure is valid and will continue to have complete access. This is also true for thread attributes.

Handles in **PROCESS_INFORMATION** must be closed with **CloseHandle** when they are no longer needed.

The process is assigned a process identifier. The identifier is valid until the process terminates. It can be used to identify the process, or specified in the **OpenProcess** function to open a handle to the process. The initial thread in the process is also assigned a thread identifier. It can be specified in the **OpenThread** function to open a handle to the thread. The identifier is valid until the thread terminates and can be used to uniquely identify the thread within the system. These identifiers are returned in the **PROCESS_INFORMATION** structure.

When specifying an application name in the *lpApplicationName* or *lpCommandLine* strings, it doesn't matter whether the application name includes the file name extension, with one exception: an MS-DOS-based or Windows-based application whose file name extension is .com must include the .com extension.

The calling thread can use the **WaitForInputIdle** function to wait until the new process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronization between parent and child processes, because **CreateProcessWithLogonW** returns without waiting for the new process to finish its initialization. For example, the creating process would use **WaitForInputIdle** before trying to find a window associated with the new process.

The preferred way to shut down a process is by using the **ExitProcess** function, because this function sends notification of approaching termination to all DLLs attached to the process. Other means of shutting down a process do not notify the attached DLLs. Note that when a thread calls **ExitProcess**, other threads of the process are terminated

without an opportunity to execute any additional code (including the thread termination code of attached DLLs).

The **ExitProcess**, **ExitThread**, **CreateThread**, and **CreateRemoteThread** functions, and processes that are starting (as the result of a call by **CreateProcessWithLogonW**) are serialized within a process. Only one of these events can happen at a time, and the following restrictions apply:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is finished for the process.
- Only one thread at a time can be in a DLL initialization or detach routine.
- The **ExitProcess** function does not return until there are no threads executing DLL initialization or detach routines.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use advapi32.lib.
Unicode: Declared only as Unicode.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **CreateEnvironmentBlock**, **CreateProcessAsUser**, **CreateRemoteThread**, **CreateThread**, **ExitProcess**, **ExitThread**, **GetExitCodeProcess**, **OpenProcess**, **OpenThread**, **PROCESS_INFORMATION**, **SetErrorMode**, **STARTUPINFO**, **WaitForInputIdle**

CreateRemoteThread

The **CreateRemoteThread** function creates a thread that runs in the virtual address space of another process.

```
HANDLE CreateRemoteThread(
  HANDLE hProcess,                          // handle to process
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
  DWORD dwStackSize,                        // initial stack size
  LPTHREAD_START_ROUTINE lpStartAddress,    // thread function
  LPVOID lpParameter,                       // thread argument
  DWORD dwCreationFlags,                    // creation option
  LPDWORD lpThreadId                        // thread identifier
);
```

Parameters

hProcess

[in] Handle to the process in which the thread is to be created. The handle must have the PROCESS_CREATE_THREAD, PROCESS_VM_OPERATION, PROCESS_VM_WRITE, and PROCESS_VM_READ access rights. For more information, see *Process Security and Access Rights*.

lpThreadAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new thread and determines whether child processes can inherit the returned handle. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor and the handle cannot be inherited.

dwStackSize

[in] Specifies the initial commit size of the stack, in bytes. The system rounds this value to the nearest page. If this value is zero, or is smaller than the default commit size, the default is to use the same size as the calling thread. For more information, see *Thread Stack Size*.

lpStartAddress

[in] Pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and represents the starting address of the thread in the remote process. The function must exist in the remote process. For more information on the thread function, see **ThreadProc**.

lpParameter

[in] Specifies a single value passed to the thread function.

dwCreationFlags

[in] Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state and will not run until the **ResumeThread** function is called. If this value is zero, the thread runs immediately after creation.

lpThreadId

[out] Pointer to a variable that receives the thread identifier.

If this parameter is NULL, the thread identifier is not returned.

Return Values

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

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

Note that **CreateRemoteThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of **CreateProcess**, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Remarks

The **CreateRemoteThread** function causes a new thread of execution to begin in the address space of the specified process. The thread has access to all objects opened by the process.

The new thread handle is created with full access to the new thread. If a security descriptor is not provided, the handle may be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the **GetThreadPriority** and **SetThreadPriority** functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting for the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle**.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a **CreateProcess** call) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **CreateProcess**, **CreateThread**, **ExitProcess**, **ExitThread**, **GetThreadPriority**, **ResumeThread**, **SECURITY_ATTRIBUTES**, **SetThreadPriority**, **ThreadProc**

CreateThread

The **CreateThread** function creates a thread to execute within the virtual address space of the calling process.

To create a thread that runs in the virtual address space of another process, use the **CreateRemoteThread** function.

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // SD
  DWORD dwStackSize,                        // initial stack size
  LPTHREAD_START_ROUTINE lpStartAddress,    // thread function
  LPVOID lpParameter,                       // thread argument
  DWORD dwCreationFlags,                    // creation option
  LPDWORD lpThreadId                        // thread identifier
);
```

Parameters

lpThreadAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpThreadAttributes* is NULL, the handle cannot be inherited.

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new thread. If *lpThreadAttributes* is NULL, the thread gets a default security descriptor.

dwStackSize

[in] Specifies the initial commit size of the stack, in bytes. The system rounds this value to the nearest page. If this value is zero, or is smaller than the default commit size, the default is to use the same size as the calling thread. For more information, see *Thread Stack Size*.

lpStartAddress

[in] Pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread and represents the starting address of the thread. For more information on the thread function, see **ThreadProc**.

lpParameter

[in] Specifies a single parameter value passed to the thread.

dwCreationFlags

[in] Specifies additional flags that control the creation of the thread. If the CREATE_SUSPENDED flag is specified, the thread is created in a suspended state, and will not run until the **ResumeThread** function is called. If this value is zero, the thread runs immediately after creation. At this time, no other values are supported.

lpThreadId

[out] Pointer to a variable that receives the thread identifier.

Windows NT/2000: If this parameter is NULL, the thread identifier is not returned.

Windows 95/98: This parameter may not be NULL.

Return Values

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

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

Note that **CreateThread** may succeed even if *lpStartAddress* points to data, code, or is not accessible. If the start address is invalid when the thread runs, an exception occurs, and the thread terminates. Thread termination due to a invalid start address is handled as an error exit for the thread's process. This behavior is similar to the asynchronous nature of **CreateProcess**, where the process is created even if it refers to invalid or missing dynamic-link libraries (DLLs).

Windows 95/98: CreateThread succeeds only when it is called in the context of a 32-bit program. A 32-bit DLL cannot create an additional thread when that DLL is being called by a 16-bit program.

Remarks

The number of threads a process can create is limited by the available virtual memory. By default, every thread has one megabyte of stack space. Therefore, you can create at most 2028 threads. If you reduce the default stack size, you can create more threads. However, your application will have better performance if you create one thread per processor and build queues of requests for which the application maintains the context information. A thread would process all requests in a queue before processing requests in the next queue.

The new thread handle is created with THREAD_ALL_ACCESS to the new thread. If a security descriptor is not provided, the handle can be used in any function that requires a thread object handle. When a security descriptor is provided, an access check is performed on all subsequent uses of the handle before access is granted. If the access check denies access, the requesting process cannot use the handle to gain access to the thread. If the thread impersonates a client, then calls **CreateThread** with a NULL security descriptor, the thread object created has a default security descriptor which allows access only to the impersonation token's TokenDefaultDacl owner or members. For more information, see *Thread Security and Access Rights*.

The thread execution begins at the function specified by the *lpStartAddress* parameter. If this function returns, the **DWORD** return value is used to terminate the thread in an implicit call to the **ExitThread** function. Use the **GetExitCodeThread** function to get the thread's return value.

The thread is created with a thread priority of THREAD_PRIORITY_NORMAL. Use the **GetThreadPriority** and **SetThreadPriority** functions to get and set the priority value of a thread.

When a thread terminates, the thread object attains a signaled state, satisfying any threads that were waiting on the object.

The thread object remains in the system until the thread has terminated and all handles to it have been closed through a call to **CloseHandle**.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means that the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the C run-time libraries should use the **beginthread** and **endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **CreateProcess**, **CreateRemoteThread**, **ExitProcess**, **ExitThread**, **GetExitCodeThread**, **GetThreadPriority**, **ResumeThread**, **SetThreadPriority**, **SECURITY_ATTRIBUTES**, **ThreadProc**

DeleteFiber

The **DeleteFiber** function deletes an existing fiber.

```
VOID DeleteFiber(
  LPVOID lpFiber   // pointer to the fiber to delete
);
```

Parameters

lpFiber

[in] Specifies the address of the fiber to delete.

Return Values

This function does not return a value.

Remarks

The **DeleteFiber** function deletes all data associated with the fiber. This data includes the stack, a subset of the registers, and the fiber data. If the currently running fiber calls **DeleteFiber**, the **ExitThread** function is called and the thread terminates. If the currently running fiber is deleted by another thread, the thread associated with the fiber is likely to terminate abnormally because the fiber stack has been freed.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, Process and Thread Functions, **ExitThread**

ExitProcess

The **ExitProcess** function ends a process and all its threads.

```
VOID ExitProcess(
  UINT uExitCode   // exit code for all threads
);
```

Parameters

uExitCode

[in] Specifies the exit code for the process, and for all threads that are terminated as a result of this call. Use the **GetExitCodeProcess** function to retrieve the process's exit value. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

Return Values

This function does not return a value.

Remarks

ExitProcess is the preferred method of ending a process. This function provides a clean process shutdown. This includes calling the entry-point function of all attached dynamic-link libraries (DLLs) with a value indicating that the process is detaching from the DLL. If

a process terminates by calling **TerminateProcess**, the DLLs that the process is attached to are not notified of the process termination.

After all attached DLLs have executed any process termination value, this function terminates the current process.

Terminating a process causes the following:

1. All of the object handles opened by the process are closed.
2. All of the threads in the process terminate their execution.
3. The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.
4. The states of all threads of the process become signaled, satisfying any threads that had been waiting for the threads to terminate.
5. The termination status of the process changes from STILL_ACTIVE to the exit value of the process.

Terminating a process does not cause child processes to be terminated.

Terminating a process does not necessarily remove the process object from the operating system. A process object is deleted when the last handle to the process is closed.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a call by **CreateProcess**) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

Requirements

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.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **CreateRemoteThread**, **CreateThread**, **ExitThread**, **GetExitCodeProcess**, **GetExitCodeThread**, **OpenProcess**, **TerminateProcess**

ExitThread

The **ExitThread** function ends a thread.

```
VOID ExitThread(
  DWORD dwExitCode   // exit code for this thread
);
```

Parameters

dwExitCode

[in] Specifies the exit code for the calling thread. Use the **GetExitCodeThread** function to retrieve a thread's exit code.

Return Values

This function does not return a value.

Remarks

ExitThread is the preferred method of exiting a thread. When this function is called (either explicitly or by returning from a thread procedure), the current thread's stack is deallocated and the thread terminates. The entry-point function of all attached dynamic-link libraries (DLLs) is invoked with a value indicating that the thread is detaching from the DLL.

If the thread is the last thread in the process when this function is called, the thread's process is also terminated.

The state of the thread object becomes signaled, releasing any other threads that had been waiting for the thread to terminate. The thread's termination status changes from STILL_ACTIVE to the value of the *dwExitCode* parameter.

Terminating a thread does not necessarily remove the thread object from the operating system. A thread object is deleted when the last handle to the thread is closed.

The **ExitProcess**, **ExitThread**, **CreateThread**, **CreateRemoteThread** functions, and a process that is starting (as the result of a **CreateProcess** call) are serialized between each other within a process. Only one of these events can happen in an address space at a time. This means the following restrictions hold:

- During process startup and DLL initialization routines, new threads can be created, but they do not begin execution until DLL initialization is done for the process.
- Only one thread in a process can be in a DLL initialization or detach routine at a time.
- **ExitProcess** does not return until no threads are in their DLL initialization or detach routines.

A thread that uses functions from the C run-time libraries should use the **_beginthread** and **_endthread** C run-time functions for thread management rather than **CreateThread** and **ExitThread**. Failure to do so results in small memory leaks when **ExitThread** is called.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **CreateRemoteThread**, **CreateThread**, **ExitProcess**, **FreeLibraryAndExitThread**, **GetExitCodeThread**, **OpenThread**, **TerminateThread**

FiberProc

The **FiberProc** function is an application-defined function used with the **CreateFiber** function. It serves as the starting address for a fiber. The **LPFIBER_START_ROUTINE** type defines a pointer to this callback function. **FiberProc** is a placeholder for the application-defined function name.

```
VOID CALLBACK FiberProc(
  PVOID lpParameter   // fiber data
);
```

Parameters

lpParameter

[in] Receives the fiber data passed to the function using the *lpParameter* parameter of the **CreateFiber** function.

Return Values

This function does not return a value.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

Processes and Threads Overview, *Process and Thread Functions*, **CreateFiber**

FreeEnvironmentStrings

The **FreeEnvironmentStrings** function frees a block of environment strings.

```
BOOL FreeEnvironmentStrings(
  LPTSTR lpszEnvironmentBlock  // environment strings
);
```

Parameters

lpszEnvironmentBlock
[in] Pointer to a block of environment strings. The pointer to the block must be obtained by calling the **GetEnvironmentStrings** 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

When **GetEnvironmentStrings** is called, it allocates memory for a block of environment strings. When the block is no longer needed, it should be freed by calling **FreeEnvironmentStrings**.

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.

Processes and Threads Overview, *Process and Thread Functions*, **GetEnvironmentStrings**

GetCommandLine

The **GetCommandLine** function returns a pointer to the command-line string for the current process.

```
LPTSTR GetCommandLine(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is a pointer to the command-line string for the current process.

Remarks

ANSI console processes written in C can use the *argc* and *argv* arguments of the **main** function to access the command-line arguments. ANSI GUI applications can use the *lpCmdLine* parameter of the **WinMain** function to access the command-line string, excluding the program name. The reason that **main** and **WinMain** cannot return Unicode strings is that *argc*, *argv*, and *lpCmdLine* use the **LPSTR** data type for parameters, not the **LPTSTR** data type. The **GetCommandLine** function can be used to access Unicode strings, because it uses the **LPTSTR** data type.

To convert the command line to an *argv* style array of strings, call the **CommandLineToArgvW** function.

Note The command line that the operating system provides to a process is not necessarily identical to the COMMAND line that the calling process gives to the **CreateProcess** 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 and Windows NT/2000.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CommandLineToArgvW**, **CreateProcess**, **WinMain**

GetCurrentProcess

The **GetCurrentProcess** function returns a pseudo handle for the current process.

```
HANDLE GetCurrentProcess(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is a pseudo handle to the current process.

Remarks

A pseudo handle is a special constant that is interpreted as the current process handle. The calling process can use this handle to specify its own process whenever a process handle is required. Pseudo handles are not inherited by child processes.

This handle has the maximum possible access to the process object. For systems that support security descriptors, this is the maximum access allowed by the security descriptor for the calling process. For systems that do not support security descriptors, this is PROCESS_ALL_ACCESS. For more information, see *Process Security and Access Rights*.

A process can create a "real" handle to itself that is valid in the context of other processes, or that can be inherited by other processes, by specifying the pseudo handle as the source handle in a call to the **DuplicateHandle** function. A process can also use the **OpenProcess** function to open a real handle to itself.

The pseudo handle need not be closed when it is no longer needed. Calling the **CloseHandle** function with a pseudo handle has no effect. If the pseudo handle is duplicated by **DuplicateHandle**, the duplicate handle must be closed.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **DuplicateHandle**, **GetCurrentProcessId**, **GetCurrentThread**, **OpenProcess**

GetCurrentProcessId

The **GetCurrentProcessId** function returns the process identifier of the calling process.

```
DWORD GetCurrentProcessId(VOID);
```

Parameters
This function has no parameters.

Return Values
The return value is the process identifier of the calling process.

Remarks
Until the process terminates, the process identifier uniquely identifies the process throughout the system.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements
Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also
Processes and Threads Overview, *Process and Thread Functions*, **GetCurrentProcess**, **OpenProcess**

GetCurrentThread

The **GetCurrentThread** function returns a pseudo handle for the current thread.

```
HANDLE GetCurrentThread(VOID);
```

Parameters
This function has no parameters.

Return Values
The return value is a pseudo handle for the current thread.

Remarks
A pseudo handle is a special constant that is interpreted as the current thread handle. The calling thread can use this handle to specify itself whenever a thread handle is required. Pseudo handles are not inherited by child processes.

This handle has the maximum possible access to the thread object. For systems that support security descriptors, this is the maximum access allowed by the security

descriptor for the calling process. For systems that do not support security descriptors, this is THREAD_ALL_ACCESS.

The function cannot be used by one thread to create a handle that can be used by other threads to refer to the first thread. The handle is always interpreted as referring to the thread that is using it. A thread can create a "real" handle to itself that can be used by other threads, or inherited by other processes, by specifying the pseudo handle as the source handle in a call to the **DuplicateHandle** function.

The pseudo handle need not be closed when it is no longer needed. Calling the **CloseHandle** function with this handle has no effect. If the pseudo handle is duplicated by **DuplicateHandle**, the duplicate handle must be closed.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **DuplicateHandle**, **GetCurrentProcess**, **GetCurrentThreadId**, **OpenThread**

GetCurrentThreadId

The **GetCurrentThreadId** function returns the thread identifier of the calling thread.

```
DWORD GetCurrentThreadId(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is the thread identifier of the calling thread.

Remarks

Until the thread terminates, the thread identifier uniquely identifies the thread throughout 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **GetCurrentThread**, **OpenThread**

GetEnvironmentStrings

The **GetEnvironmentStrings** function returns the address of the environment block for the current process. This function replaces the **GetDOSEnvironment** function.

```
LPVOID GetEnvironmentStrings(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is a pointer to an environment block for the current process.

Remarks

Do not use the return value of **GetEnvironmentStrings** to get or set environment variables. Instead, use the **GetEnvironmentVariable** and **SetEnvironmentVariable** functions to access the environment variables within this block. When the block is no longer needed, it should be freed by calling **FreeEnvironmentStrings**.

A process can use this function's return value to specify the environment address used by the **CreateProcess** 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

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **GetEnvironmentVariable**, **SetEnvironmentVariable**, **FreeEnvironmentStrings**

GetEnvironmentVariable

The **GetEnvironmentVariable** function retrieves the value of the specified variable from the environment block of the calling process. The value is in the form of a null-terminated string of characters.

```
DWORD GetEnvironmentVariable(
  LPCTSTR lpName,   // environment variable name
  LPTSTR lpBuffer,  // buffer for variable value
  DWORD nSize       // size of buffer
);
```

Parameters

lpName
[in] Pointer to a null-terminated string that specifies the environment variable.

lpBuffer
[out] Pointer to a buffer to receive the value of the specified environment variable.

nSize
[in] Specifies the size, in characters, of the buffer pointed to by the *lpBuffer* parameter.

Return Values

If the function succeeds, the return value is the number of characters stored into the buffer pointed to by *lpBuffer*, not including the terminating null character.

If the specified environment variable name was not found in the environment block for the current process, the return value is zero.

If the buffer pointed to by *lpBuffer* is not large enough, the return value is the buffer size, in characters, required to hold the value string and its terminating null character.

Requirements

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

Processes and Threads Overview, *Process and Thread Functions*, **GetEnvironmentStrings**, **SetEnvironmentVariable**

GetExitCodeProcess

The **GetExitCodeProcess** function retrieves the termination status of the specified process.

```
BOOL GetExitCodeProcess(
  HANDLE hProcess,      // handle to the process
  LPDWORD lpExitCode    // termination status
);
```

Parameters

hProcess

[in] Handle to the process.

Windows NT/2000: The handle must have PROCESS_QUERY_INFORMATION access. For more information, see *Process Security and Access Rights*.

lpExitCode

[out] Pointer to a variable to receive the process termination status.

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 process has not terminated, the termination status returned is STILL_ACTIVE. If the process has terminated, the termination status returned may be one of the following:

- The exit value specified in the **ExitProcess** or **TerminateProcess** function.
- The return value from the **main** or **WinMain** function of the process.
- The exception value for an unhandled exception that caused the process to terminate.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **ExitProcess**, **ExitThread**, **TerminateProcess**, **WinMain**

GetExitCodeThread

The **GetExitCodeThread** function retrieves the termination status of the specified thread.

```
BOOL GetExitCodeThread(
  HANDLE hThread,       // handle to the thread
  LPDWORD lpExitCode    // termination status
);
```

Parameters

hThread

[in] Handle to the thread.

Windows NT/2000: The handle must have THREAD_QUERY_INFORMATION access. For more information, see *Thread Security and Access Rights*.

lpExitCode

[out] Pointer to a variable to receive the thread termination status.

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 thread has not terminated, the termination status returned is STILL_ACTIVE. If the thread has terminated, the termination status returned may be one of the following:

- The exit value specified in the **ExitThread** or **TerminateThread** function.
- The return value from the thread function.
- The exit value of the thread's process.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Processes and Threads Overview, *Process and Thread Functions*, **ExitThread**, **GetExitCodeProcess**, **OpenThread**, **TerminateThread**

GetGuiResources

The **GetGuiResources** function returns the count of handles to graphical user interface (GUI) objects in use by the specified process.

```
DWORD GetGuiResources (
  HANDLE hProcess,  // handle to process
  DWORD uiFlags     // GUI object type
);
```

Parameters

hProcess

[in] Handle to the process. The handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see *Process Security and Access Rights*.

uiFlags

[in] Specifies the GUI object type. This parameter can be one of the following values.

Value	Meaning
GR_GDIOBJECTS	Return the count of GDI objects.
GR_USEROBJECTS	Return the count of USER objects.

Return Values

If the function succeeds, the return value is the count of handles to GUI objects in use by the process. If no GUI objects are in use, the return value is zero.

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

Remarks

A process without a graphical user interface does not use GUI resources, therefore, **GetGuiResources** will return zero.

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

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **GetCurrentProcess**, **OpenProcess**

GetPriorityClass

The **GetPriorityClass** function returns the priority class for the specified process. This value, together with the priority value of each thread of the process, determines each thread's base priority level.

```
DWORD GetPriorityClass(
  HANDLE hProcess   // handle to process
);
```

Parameters

hProcess

[in] Handle to the process.

Windows NT/2000: The handle must have PROCESS_QUERY_INFORMATION access. For more information, see *Process Security and Access Rights*.

Return Values

If the function succeeds, the return value is the priority class of the specified process.

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

The process's priority class is one of the following values.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Indicates a process that performs time-critical tasks that must be executed immediately for it to run correctly. The threads of a high-priority class process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class CPU-bound application can use nearly all available cycles.

IDLE_PRIORITY_CLASS	Indicates a process whose threads run only when the system is idle and are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Indicates a normal process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Indicates a process that has the highest possible priority. The threads of a real-time priority class process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

Remarks

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The operating system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

For a table that shows the base priority levels for each combination of priority class and thread priority value, see the **SetPriorityClass** 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.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **GetThreadPriority**, **SetPriorityClass**, **SetThreadPriority**

GetProcessAffinityMask

The **GetProcessAffinityMask** function obtains a process affinity mask for the specified process and the system affinity mask for the system.

A process affinity mask is a bit vector in which each bit represents the processors that a process is allowed to run on. A system affinity mask is a bit vector in which each bit represents the processors that are configured into a system.

A process affinity mask is a proper subset of a system affinity mask. A process is only allowed to run on the processors configured into a system.

```
BOOL GetProcessAffinityMask(
  HANDLE hProcess,                    // handle to process
  PDWORD_PTR lpProcessAffinityMask,   // process affinity mask
  PDWORD_PTR lpSystemAffinityMask     // system affinity mask
);
```

Parameters

hProcess
[in] Handle to the process whose affinity mask is desired.

Windows NT/2000: This handle must have PROCESS_QUERY_INFORMATION access. For more information, see *Process Security and Access Rights*.

lpProcessAffinityMask
[out] Pointer to a variable that receives the affinity mask for the specified process.

lpSystemAffinityMask
[out] Pointer to a variable that receives the affinity mask for the system.

Return Values

If the function succeeds, the return value is nonzero.

Windows NT/2000: Upon success, the function sets the **DWORD** variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the appropriate affinity masks.

Windows 95/98: Upon success, the function sets the **DWORD** variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* to the value one.

If the function fails, the return value is zero, and the values of the **DWORD** variables pointed to by *lpProcessAffinityMask* and *lpSystemAffinityMask* are undefined. 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Processes and Threads Overview, *Process and Thread Functions*, **SetProcessAffinityMask**, **SetThreadAffinityMask**

GetProcessIoCounters

The **GetProcessIoCounters** function retrieves accounting information for all I/O operations performed by the specified process.

```
BOOL GetProcessIoCounters(
  HANDLE hProcess,             // handle to process
  PIO_COUNTERS lpIoCounters  // I/O accouting information
);
```

Parameters

hProcess
[in] Handle to the process. The handle must have the PROCESS_QUERY access right.

lpIoCounters
[out] Pointer to an **IO_COUNTERS** structure that receives the I/O accounting information for the process.

Return Value

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: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **IO_COUNTERS**

GetProcessPriorityBoost

The **GetProcessPriorityBoost** function returns the priority boost control state of the specified process.

```
BOOL GetProcessPriorityBoost(
  HANDLE hProcess,             // handle to process
  PBOOL pDisablePriorityBoost  // priority boost state
);
```

Parameters

hProcess

[in] Handle to the process. This handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see *Process Security and Access Rights*.

pDisablePriorityBoost

[out] Pointer to a variable that receives the priority boost control state. A value of TRUE indicates that dynamic boosting is disabled. A value of FALSE indicates normal behavior.

Return Values

If the function succeeds, the return value is nonzero. In that case, the variable pointed to by the *pDisablePriorityBoost* parameter receives the priority boost control state.

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 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **SetProcessPriorityBoost**

GetProcessShutdownParameters

The **GetProcessShutdownParameters** function retrieves shutdown parameters for the currently calling process.

```
BOOL GetProcessShutdownParameters(
  LPDWORD lpdwLevel, // shutdown priority
  LPDWORD lpdwFlags  // shutdown flag
);
```

Parameters

lpdwLevel

[out] Pointer to a variable that receives the shutdown priority level. Higher levels shut down first. System level shutdown orders are reserved for system components. Higher numbers shut down first. Following are the level conventions:

Value	Meaning
000–0FF	System reserved last shutdown range.
100–1FF	Application reserved last shutdown range.
200–2FF	Application reserved "in between" shutdown range.
300–3FF	Application reserved first shutdown range.
400–4FF	System reserved first shutdown range.

All processes start at shutdown level 0x280.

lpdwFlags

[out] Pointer to a variable that receives the shutdown flags. This parameter can be the following value.

Value	Meaning
SHUTDOWN_NORETRY	If this process takes longer than the specified timeout to shut down, do not display a retry dialog box for the user. Instead, just cause the process to directly exit.

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: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **SetProcessShutdownParameters**

GetProcessTimes

The **GetProcessTimes** function obtains timing information about a specified process.

```
BOOL GetProcessTimes(
  HANDLE hProcess,           // handle to process
  LPFILETIME lpCreationTime, // process creation time
  LPFILETIME lpExitTime,     // process exit time
```

```
  LPFILETIME lpKernelTime,    // process kernel-mode time
  LPFILETIME lpUserTime       // process user-mode time
);
```

Parameters

hProcess

[in] Handle to the process whose timing information is sought. This handle must be created with PROCESS_QUERY_INFORMATION access. For more information, see *Process Security and Access Rights*.

lpCreationTime

[out] Pointer to a **FILETIME** structure that receives the creation time of the process.

lpExitTime

[out] Pointer to a **FILETIME** structure that receives the exit time of the process. If the process has not exited, the content of this structure is undefined.

lpKernelTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the process has executed in kernel mode. The time that each of the threads of the process has executed in kernel mode is determined, and then all of those times are summed together to obtain this value.

lpUserTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the process has executed in user mode. The time that each of the threads of the process has executed in user mode is determined, and then all of those times are summed together to obtain this value.

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

All times are expressed using **FILETIME** data structures. Such a structure contains two 32-bit values that combine to form a 64-bit count of 100-nanosecond time units.

Process creation and exit times are points in time expressed as the amount of time that has elapsed since midnight on January 1, 1601 at Greenwich, England. The Win32 API provides several functions that an application can use to convert such values to more generally useful forms.

Process kernel mode and user mode times are amounts of time. For example, if a process has spent one second in kernel mode, this function will fill the **FILETIME** structure specified by *lpKernelTime* with a 64-bit value of ten million. That is the number of 100-nanosecond units in one second.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **FILETIME**, **FileTimeToDosDateTime**, **FileTimeToLocalFileTime**, **FileTimeToSystemTime**

GetProcessVersion

The **GetProcessVersion** function obtains the major and minor version numbers of the system on which a specified process expects to run.

```
DWORD GetProcessVersion(
  DWORD ProcessId  // process identifier
);
```

Parameters

ProcessId

[in] Process identifier that specifies the process of interest. A value of zero specifies the calling process.

Return Values

If the function succeeds, the return value is the version of the system on which the process expects to run. The high word of the return value contains the major version number. The low word of the return value contains the minor version number.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. The function fails if *ProcessId* is an invalid value.

Remarks

The **GetProcessVersion** function performs less quickly when *ProcessId* is nonzero, specifying a process other than the calling process.

The version number returned by this function is the version number stamped in the image header of the .exe file the process is running. Linker programs set this value.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*

GetProcessWorkingSetSize

The **GetProcessWorkingSetSize** function obtains the minimum and maximum working set sizes of a specified process.

The "working set" of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The size of a process' working set is specified in bytes. The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

```
BOOL GetProcessWorkingSetSize(
  HANDLE hProcess,                    // handle to the process
  PSIZE_T lpMinimumWorkingSetSize,   // minimum working set size
  PSIZE_T lpMaximumWorkingSetSize    // maximum working set size
);
```

Parameters

hProcess

[in] Handle to the process whose working set sizes will be obtained. The handle must have the PROCESS_QUERY_INFORMATION access right. For more information, see *Process Security and Access Rights*.

lpMinimumWorkingSetSize

[out] Pointer to a variable that receives the minimum working set size of the specified process. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

lpMaximumWorkingSetSize

[out] Pointer to a variable that receives the maximum working set size of the specified process. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active when memory is in short supply.

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: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*,
SetProcessWorkingSetSize

GetStartupInfo

The **GetStartupInfo** function retrieves the contents of the **STARTUPINFO** structure that was specified when the calling process was created.

```
VOID GetStartupInfo(
  LPSTARTUPINFO lpStartupInfo   // startup information
);
```

Parameters

lpStartupInfo
[out] Pointer to a **STARTUPINFO** structure that receives the startup information.

Return Values

This function does not return a value.

Remarks

The **STARTUPINFO** structure was specified by the process that created the calling process. It can be used to specify properties associated with the main window of the calling process.

Requirements

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.

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **STARTUPINFO**

GetThreadPriority

The **GetThreadPriority** function returns the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base-priority level.

```
int GetThreadPriority(
  HANDLE hThread    // handle to thread
);
```

Parameters

hThread

[in] Handle to the thread.

Windows NT/2000: The handle must have THREAD_QUERY_INFORMATION access. For more information, see *Thread Security and Access Rights*.

Return Values

If the function succeeds, the return value is the thread's priority level.

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

The thread's priority level is one of the following values:

Priority	Meaning
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority for the priority class.
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority for the priority class.
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority for the priority class.
THREAD_PRIORITY_IDLE	Indicates a base-priority level of 1 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base-priority level of 16 for REALTIME_PRIORITY_CLASS processes.

(continued)

(continued)

Priority	Meaning
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority for the priority class.
THREAD_PRIORITY_NORMAL	Indicates normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL	Indicates a base-priority level of 15 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base-priority level of 31 for REALTIME_PRIORITY_CLASS processes.

Remarks

Every thread has a base-priority level determined by the thread's priority value and the priority class of its process. The operating system uses the base-priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level will scheduling of threads at a lower level take place.

For a table that shows the base-priority levels for each combination of priority class and thread priority value, refer to the **SetPriorityClass** 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **GetPriorityClass**, **OpenThread**, **SetPriorityClass**, **SetThreadPriority**

GetThreadPriorityBoost

The **GetThreadPriorityBoost** function returns the priority boost control state of the specified thread.

```
BOOL GetThreadPriorityBoost(
  HANDLE hThread,                // handle to thread
  PBOOL pDisablePriorityBoost    // priority boost state
);
```

Parameters

hThread

[in] Handle to the thread. This thread must have THREAD_QUERY_INFORMATION access. For more information, see *Thread Security and Access Rights*.

pDisablePriorityBoost

[out] Pointer to a variable that receives the priority boost control state. A value of TRUE indicates that dynamic boosting is disabled. A value of FALSE indicates normal behavior.

Return Values

If the function succeeds, the return value is nonzero. In that case, the variable pointed to by the *pDisablePriorityBoost* parameter receives the priority boost control state.

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 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **OpenThread**, **SetThreadPriorityBoost**

GetThreadTimes

The **GetThreadTimes** function obtains timing information about a specified thread.

```
BOOL GetThreadTimes(
  HANDLE hThread,             // handle to thread
  LPFILETIME lpCreationTime,  // thread creation time
  LPFILETIME lpExitTime,      // thread exit time
  LPFILETIME lpKernelTime,    // thread kernel-mode time
  LPFILETIME lpUserTime       // thread user-mode time
);
```

Parameters

hThread

[in] Handle to the thread whose timing information is sought. This handle must be created with THREAD_QUERY_INFORMATION access. For more information, see *Thread Security and Access Rights.*

lpCreationTime

[out] Pointer to a **FILETIME** structure that receives the creation time of the thread.

lpExitTime

[out] Pointer to a **FILETIME** structure that receives the exit time of the thread. If the thread has not exited, the content of this structure is undefined.

lpKernelTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the thread has executed in kernel mode.

lpUserTime

[out] Pointer to a **FILETIME** structure that receives the amount of time that the thread has executed in user mode.

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

All times are expressed using **FILETIME** data structures. Such a structure contains two 32-bit values that combine to form a 64-bit count of 100-nanosecond time units.

Thread creation and exit times are points in time expressed as the amount of time that has elapsed since midnight on January 1, 1601 at Greenwich, England. The Win32 API provides several functions that an application can use to convert such values to more generally useful forms; see *Time Functions*, particularly those noted in the following See Also section.

Thread kernel mode and user mode times are amounts of time. For example, if a thread has spent one second in kernel mode, this function will fill the **FILETIME** structure specified by *lpKernelTime* with a 64-bit value of ten million. That is the number of 100-nanosecond units in one second.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **FILETIME**, **FileTimeToDosDateTime**, **FileTimeToLocalFileTime**, **FileTimeToSystemTime**, **OpenThread**

OpenJobObject

The **OpenJobObject** function opens an existing job object.

```
HANDLE OpenJobObject(
  DWORD dwDesiredAccess,  // access right
  BOOL bInheritHandles,   // inheritance state
  LPCTSTR lpName          // job name
);
```

Parameters

dwDesiredAccess

[in] Specifies the desired access mode to the job object. This parameter can be one or more of the following values.

Value	Meaning
MAXIMUM_ALLOWED	Specifies maximum access rights to the job object that are valid for the caller.
JOB_OBJECT_ASSIGN_PROCESS	Specifies the assign process access right to the object. Allows processes to be assigned to the job.
JOB_OBJECT_SET_ATTRIBUTES	Specifies the set attribute access right to the object. Allows job object attributes to be set.
JOB_OBJECT_QUERY	Specifies the query access right to the object. Allows job object attributes and accounting information to be queried.
JOB_OBJECT_TERMINATE	Specifies the terminate access right to the object. Allows termination of all processes in the job object.
JOB_OBJECT_SET_SECURITY_ATTRIBUTES	Specifies the security attributes access right to the object. Allows security limitations on all processes in the job object to be set.
JOB_OBJECT_ALL_ACCESS	Specifies the full access right to the job object.

bInheritHandles

[in] Specifies whether the returned handle is inherited when a new process is created. If this parameter is TRUE, the new process inherits the handle.

lpName
[in] Pointer to a null-terminated string specifying the name of the job to be opened. Name comparisons are case sensitive.

Return Values

If the function succeeds, the return value is a handle to the job. The handle provides the requested access to the job.

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

Remarks

To associate a process with a job, use the **AssignProcessToJobObject** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

Processes and Threads Overview, *Process and Thread Functions*, **AssignProcessToJobObject**

OpenProcess

The **OpenProcess** function returns a handle to an existing process object.

```
HANDLE OpenProcess(
  DWORD dwDesiredAccess,  // access flag
  BOOL bInheritHandle,    // handle inheritance option
  DWORD dwProcessId       // process identifier
);
```

Parameters

dwDesiredAccess
[in] Specifies the access to the process object. For operating systems that support security checking, this access is checked against any security descriptor for the target process. This parameter can be STANDARD_RIGHTS_REQUIRED or one or more of the following values:

Value	Description
PROCESS_ALL_ACCESS	Specifies all possible access flags for the process object.
PROCESS_CREATE_PROCESS	Used internally.
PROCESS_CREATE_THREAD	Enables using the process handle in the **CreateRemoteThread** function to create a thread in the process.
PROCESS_DUP_HANDLE	Enables using the process handle as either the source or target process in the **DuplicateHandle** function to duplicate a handle.
PROCESS_QUERY_INFORMATION	Enables using the process handle in the **GetExitCodeProcess** and **GetPriorityClass** functions to read information from the process object.
PROCESS_SET_QUOTA	Enables using the process handle in the **AssignProcessToJobObject** and **SetProcessWorkingSetSize** functions to set memory limits.
PROCESS_SET_INFORMATION	Enables using the process handle in the **SetPriorityClass** function to set the priority class of the process.
PROCESS_TERMINATE	Enables using the process handle in the **TerminateProcess** function to terminate the process.
PROCESS_VM_OPERATION	Enables using the process handle in the **VirtualProtectEx** and **WriteProcessMemory** functions to modify the virtual memory of the process.
PROCESS_VM_READ	Enables using the process handle in the **ReadProcessMemory** function to read from the virtual memory of the process.
PROCESS_VM_WRITE	Enables using the process handle in the **WriteProcessMemory** function to write to the virtual memory of the process.
SYNCHRONIZE	**Windows NT/2000:** Enables using the process handle in any of the wait functions to wait for the process to terminate.

bInheritHandle

[in] Specifies whether the returned handle can be inherited by a new process created by the current process. If TRUE, the handle is inheritable.

dwProcessId
[in] Specifies the identifier of the process to open.

Return Values

If the function succeeds, the return value is an open handle to the specified process.

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

Remarks

The handle returned by the **OpenProcess** function can be used in any function that requires a handle to a process, such as the wait functions, provided the appropriate access rights were requested.

When you are finished with the handle, be sure to close it using the **CloseHandle** 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.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **AssignProcessToJobObject**, **CloseHandle**, **CreateProcess**, **CreateRemoteThread**, **DuplicateHandle**, **GetCurrentProcess**, **GetCurrentProcessId**, **GetExitCodeProcess**, **GetPriorityClass**, **ReadProcessMemory**, **SetPriorityClass**, **SetProcessWorkingSetSize**, **TerminateProcess**, **VirtualProtectEx**, **WriteProcessMemory**

OpenThread

The **OpenThread** function returns a handle to an existing thread object.

```
HANDLE OpenThread(
  DWORD dwDesiredAccess,  // access right
  BOOL bInheritHandle,    // handle inheritance option
  DWORD dwThreadId        // thread identifier
);
```

Parameters

dwDesiredAccess

[in] Specifies the desired access to the thread object. For operating systems that support security checking, this access level is checked against any security descriptor for the target thread. This parameter can be STANDARD_RIGHTS_REQUIRED or any combination of the following values.

Value	Description
SYNCHRONIZE	Enables the use of the thread handle in any of the wait functions.
THREAD_ALL_ACCESS	Specifies all possible access flags for the thread object.
THREAD_GET_CONTEXT	Enables the use of the thread handle in the **GetThreadContext** function.
THREAD_QUERY_INFORMATION	Enables the use of the thread handle to read certain information from the thread object, such as the exit code (see **GetExitCodeThread**).
THREAD_SET_CONTEXT	Enables the use of the thread handle in the **SetThreadContext** function.
THREAD_SET_INFORMATION	Enables the use of the thread handle to set certain information for the thread object.
THREAD_SET_THREAD_TOKEN	Enables the use of the thread handle in the **SetTokenInformation** function to set the thread token.
THREAD_SUSPEND_RESUME	Enables the use of the thread handle in the **SuspendThread** or **ResumeThread** function to suspend and resume the thread.
THREAD_TERMINATE	Enables the use of the thread handle in the **TerminateThread** function to terminate the thread.

bInheritHandle

[in] Indicates whether the returned handle is to be inherited by a new process created by the current process. If this parameter is TRUE, the new process will inherit the handle.

dwThreadId

[in] Specifies the identifier of the thread to open.

Return Values

If the function succeeds, the return value is an open handle to the specified process.

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

Remarks

The handle returned by **OpenThread** can be used in any function that requires a handle to a thread, such as the wait functions, provided you requested the appropriate access rights. The handle is granted access to the thread object only to the extent it was specified in the *dwDesiredAccess* parameter.

When you are finished with the handle, be sure to close it by using the **CloseHandle** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CloseHandle**, **GetExitCodeThread**, **GetThreadContext**, **ResumeThread**, **SetThreadContext**, **SetTokenInformation**, **SuspendThread**, **TerminateThread**

QueryInformationJobObject

The **QueryInformationJobObject** function obtains limit and job state information from the job object.

```
BOOL QueryInformationJobObject(
  HANDLE hJob,                              // handle to job
  JOBOBJECTINFOCLASS JobObjectInfoClass,    // information class
  LPVOID lpJobObjectInfo,                   // limit information
  DWORD cbJobObjectInfoLength,              // limit information size
  LPDWORD lpReturnLength                    // data written
);
```

Parameters

hJob

[in] Handle to the job whose information is being queried. The **CreateJobObject** or **OpenJobObject** function returns this handle. The handle must have the JOB_OBJECT_QUERY access right associated with it. For more information, see *Job Object Security and Access Rights*.

If this value is NULL and the calling process is associated with a job, the job associated with the calling process is used.

JobObjectInfoClass

[in] Specifies the information class for limits to be queried. This parameter can be one of the following values.

Value	Meaning
JobObjectBasicAccountingInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION** structure.
JobObjectBasicAndIoAccountingInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION** structure.
JobObjectBasicLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JobObjectBasicProcessIdList	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_PROCESS_ID_LIST** structure.
JobObjectBasicUIRestrictions	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_UI_RESTRICTIONS** structure.
JobObjectExtendedLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** structure.
JobObjectSecurityLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_SECURITY_LIMIT_INFORMATION** structure.

lpJobObjectInfo

[out] Receives the limit information. The format of this data depends on the value of the *JobObjectInfoClass* parameter.

cbJobObjectInfoLength

[in] Specifies the count, in bytes, of the job information being queried.

lpReturnLength

[out] Pointer to a variable that receives the length of data written to the structure pointed to by the *lpJobObjectInfo* parameter. If you do not want to receive this information, specify NULL.

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 **QueryInformationJobObject** to obtain the current limits, modify them, then use the **SetInformationJobObject** function to set new limits.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION**, **JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION**, **JOBOBJECT_BASIC_LIMIT_INFORMATION**, **JOBOBJECT_BASIC_PROCESS_ID_LIST**, **JOBOBJECT_BASIC_UI_RESTRICTIONS**, **JOBOBJECT_EXTENDED_LIMIT_INFORMATION**, **JOBOBJECT_SECURITY_LIMIT_INFORMATION**, **SetInformationJobObject**

QueueUserWorkItem

The **QueueUserWorkItem** function queues a work item to a worker thread in the thread pool.

```
BOOL QueueUserWorkItem(
  LPTHREAD_START_ROUTINE Function,  // starting address
  PVOID Context,                    // function data
  ULONG Flags                       // worker options
);
```

Parameters

Function
[in] Pointer to the application-defined function of type **LPTHREAD_START_ROUTINE** to be executed by the thread in the thread pool. This value represents the starting address of the thread. For more information, see **ThreadProc**.

Context
[in] Specifies a single parameter value that will be passed to the thread function.

Flags
[in] This parameter can be one or more of the following values:

Value	Meaning
WT_EXECUTEDEFAULT	By default, the callback function is queued to a non-I/O worker thread.
WT_EXECUTEINIOTHREAD	The callback function is queued to an I/O worker thread. This flag should be used if the function should be executed in a thread that handles pending asynchronous I/O requests before exiting.
	The callback function is queued as an APC. Be sure to address reentrancy issues if the function performs an alertable wait operation.
WT_EXECUTEINPERSISTENTIOTHREAD	The callback function is queued to a thread that never terminates.
	Note that currently no worker thread is truly persistent, although I/O worker threads do not terminate if there are any pending I/O requests.
WT_EXECUTELONGFUNCTION	Specifies that the callback function can perform a long wait. This flags helps the system to decide if it should create a new thread. This flag can be used only with the WT_EXECUTEINIOTHREAD flag.

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 function in a DLL is queued to a worker thread, be sure that the function has completed execution before the DLL is unloaded.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **ThreadProc**

ResumeThread

The **ResumeThread** function decrements a thread's suspend count. When the suspend count is decremented to zero, the execution of the thread is resumed.

```
DWORD ResumeThread(
  HANDLE hThread   // handle to thread
);
```

Parameters

hThread

[in] Handle to the thread to be restarted.

Windows NT/2000: The handle must have THREAD_SUSPEND_RESUME access to the thread. For more information, see *Thread Security and Access Rights*.

Return Values

If the function succeeds, the return value is the thread's previous suspend count.

If the function fails, the return value is –1. To get extended error information, call **GetLastError**.

Remarks

The **ResumeThread** function checks the suspend count of the subject thread. If the suspend count is 0, the thread is not currently suspended. Otherwise, the subject thread's suspend count is decremented. If the resulting value is 0, then the execution of the subject thread is resumed.

If the return value is 0, the specified thread was not suspended. If the return value is 1, the specified thread was suspended but was restarted. If the return value is greater than 1, the specified thread is still suspended.

Note that while reporting debug events, all threads within the reporting process are frozen. Debuggers are expected to use the **SuspendThread** and **ResumeThread** functions to limit the set of threads that can execute within a process. By suspending all threads in a process except for the one reporting a debug event, it is possible to "single step" a single thread. The other threads are not released by a continue operation if they are suspended.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **OpenThread**, **SuspendThread**

SetEnvironmentVariable

The **SetEnvironmentVariable** function sets the value of an environment variable for the current process.

```
BOOL SetEnvironmentVariable(
  LPCTSTR lpName,  // environment variable name
  LPCTSTR lpValue  // new value for variable
);
```

Parameters

lpName

[in] Pointer to a null-terminated string that specifies the environment variable whose value is being set. The operating system creates the environment variable if it does not exist and *lpValue* is not NULL.

lpValue

[in] Pointer to a null-terminated string containing the new value of the specified environment variable. If this parameter is NULL, the variable is deleted from the current process's environment.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

Processes and Threads Overview, *Process and Thread Functions*, **GetEnvironmentVariable**

SetInformationJobObject

The **SetInformationJobObject** function is used to set limits for a job object.

```
BOOL SetInformationJobObject(
  HANDLE hJob,                             // handle to job
  JOBOBJECTINFOCLASS JobObjectInfoClass,   // information class
  LPVOID lpJobObjectInfo,                  // limit information
  DWORD cbJobObjectInfoLength              // size of limit information
);
```

Parameters

hJob

[in] Handle to the job whose limits are being set. The **CreateJobObject** or **OpenJobObject** function returns this handle. The handle must have the JOB_OBJECT_SET_ATTRIBUTES access right associated with it. For more information, see *Job Object Security and Access Rights*.

JobObjectInfoClass

[in] Specifies the information class for limits to be set. This parameter can be one of the following values.

Value	Meaning
JobObjectAssociateCompletionPortInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_ASSOCIATE_COMPLETION_PORT** structure.
JobObjectBasicLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JobObjectBasicUIRestrictions	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_BASIC_UI_RESTRICTIONS** structure.
JobObjectEndOfJobTimeInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_END_OF_JOB_TIME_INFORMATION** structure.
JobObjectExtendedLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** structure.
JobObjectSecurityLimitInformation	The *lpJobObjectInfo* parameter is a pointer to a **JOBOBJECT_SECURITY_LIMIT_INFORMATION** structure. The *hJob* handle must have the JOB_OBJECT_SET_SECURITY_ATTRIBUTES access right associated with it.

lpJobObjectInfo
[in] Specifies the limits to be set for the job. The format of this data depends on the value of *JobObjectInfoClass*.

cbJobObjectInfoLength
[in] Specifies the count, in bytes, of the job information being set.

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 the **SetInformationJobObject** function to set several limits in a single call. If you want to establish the limits one at a time or change a subset of the limits, call the **QueryInformationJobObject** function to obtain the current limits, modify these limits, and then call **SetInformationJobObject**.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Processes and Threads Overview, *Process and Thread Functions*, **JOBOBJECT_ASSOCIATE_COMPLETION_PORT**, **JOBOBJECT_BASIC_LIMIT_INFORMATION**, **JOBOBJECT_BASIC_UI_RESTRICTIONS**, **JOBOBJECT_END_OF_JOB_TIME_INFORMATION**, **JOBOBJECT_EXTENDED_LIMIT_INFORMATION**, **JOBOBJECT_SECURITY_LIMIT_INFORMATION** , **QueryInformationJobObject**

SetPriorityClass

The **SetPriorityClass** function sets the priority class for the specified process. This value together with the priority value of each thread of the process determines each thread's base priority level.

```
BOOL SetPriorityClass(
  HANDLE hProcess,        // handle to process
  DWORD dwPriorityClass   // priority class
);
```

Parameters

hProcess

[in] Handle to the process.

Windows NT/2000: The handle must have the PROCESS_SET_INFORMATION access right. For more information, see *Process Security and Access Rights*.

dwPriorityClass

[in] Specifies the priority class for the process. This parameter can be one of the following values.

Priority	Meaning
ABOVE_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above NORMAL_PRIORITY_CLASS but below HIGH_PRIORITY_CLASS.
BELOW_NORMAL_PRIORITY_CLASS	**Windows 2000:** Indicates a process that has priority above IDLE_PRIORITY_CLASS but below NORMAL_PRIORITY_CLASS.
HIGH_PRIORITY_CLASS	Specify this class for a process that performs time-critical tasks that must be executed immediately. The threads of the process preempt the threads of normal or idle priority class processes. An example is the Task List, which must respond quickly when called by the user, regardless of the load on the operating system. Use extreme care when using the high-priority class, because a high-priority class application can use nearly all available CPU time.
IDLE_PRIORITY_CLASS	Specify this class for a process whose threads run only when the system is idle. The threads of the process are preempted by the threads of any process running in a higher priority class. An example is a screen saver. The idle-priority class is inherited by child processes.
NORMAL_PRIORITY_CLASS	Specify this class for a process with no special scheduling needs.
REALTIME_PRIORITY_CLASS	Specify this class for a process that has the highest possible priority. The threads of the process preempt the threads of all other processes, including operating system processes performing important tasks. For example, a real-time process that executes for more than a very brief interval can cause disk caches not to flush or cause the mouse to be unresponsive.

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

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. The **SetThreadPriority** function enables setting the base priority level of a thread relative to the priority class of its process. For more information, see *Scheduling Priorities*.

Requirements

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.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **CreateThread**, **GetPriorityClass**, **GetThreadPriority**, **SetThreadPriority**

SetProcessAffinityMask

The **SetProcessAffinityMask** function sets a processor affinity mask for the threads of a specified process.

A process affinity mask is a bit vector in which each bit represents the processor on which the threads of the process are allowed to run.

The value of the process affinity mask must be a proper subset of the mask values obtained by the **GetProcessAffinityMask** function.

```
BOOL SetProcessAffinityMask(
  HANDLE hProcess,                  // handle to process
  DWORD_PTR dwProcessAffinityMask   // process affinity mask
);
```

Parameters

hProcess

[in] Handle to the process whose affinity mask is to be set. This handle must have the PROCESS_SET_INFORMATION access right. For more information, see *Process Security and Access Rights*.

dwProcessAffinityMask
[in] Specifies an affinity mask for the threads of the process.

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

Process affinity is inherited by any process that you start with the **CreateProcess** function.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **GetProcessAffinityMask**

SetProcessPriorityBoost

The **SetProcessPriorityBoost** function disables the ability of the system to temporarily boost the priority of the threads of the specified process.

```
BOOL SetProcessPriorityBoost(
  HANDLE hProcess,          // handle to process
  BOOL DisablePriorityBoost // priority boost state
);
```

Parameters

hProcess
[in] Handle to the process. This handle must have the PROCESS_SET_INFORMATION access right. For more information, see *Process Security and Access Rights*.

DisablePriorityBoost
[in] Specifies the priority boost control state. A value of TRUE indicates that dynamic boosting is to be disabled. A value of FALSE restores normal behavior.

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 thread is running in one of the dynamic priority classes, the system temporarily boosts the thread's priority when it is taken out of a wait state. If **SetProcessPriorityBoost** is called with the *DisablePriorityBoost* parameter set to TRUE, its threads' priorities are not boosted. This setting affects all existing threads and any threads subsequently created by the process. To restore normal behavior, call **SetProcessPriorityBoost** with *DisablePriorityBoost* set to FALSE.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **GetProcessPriorityBoost**

SetProcessShutdownParameters

The **SetProcessShutdownParameters** function sets shutdown parameters for the currently calling process. This function sets a shutdown order for a process relative to the other processes in the system.

```
BOOL SetProcessShutdownParameters(
  DWORD dwLevel,   // shutdown priority
  DWORD dwFlags    // shutdown options
);
```

Parameters

dwLevel

[in] Specifies the shutdown priority for a process relative to other processes in the system. The system shuts down processes from high *dwLevel* values to low. The highest and lowest shutdown priorities are reserved for system components. This parameter must be in the following range of values:

Value	Meaning
000–0FF	System reserved last shutdown range.
100–1FF	Application reserved last shutdown range.
200–2FF	Application reserved "in between" shutdown range.
300–3FF	Application reserved first shutdown range.
400–4FF	System reserved first shutdown range.

All processes start at shutdown level 0x280.

dwFlags

[in] This parameter can be the following value.

Value	Meaning
SHUTDOWN_NORETRY	Specifies whether to retry the shutdown if the specified time-out period expires. If this flag is specified, the system terminates the process without displaying a retry dialog box for the user.

Return Values

If the function is succeeds, the return value is nonzero.

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

Remarks

Applications running in the system security context do not get shut down by the operating system. They get notified of shutdown or logoff through the callback function installable via **SetConsoleCtrlHandler**. They also get notified in the order specified by the *dwLevel* parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **GetProcessShutdownParameters**, **SetConsoleCtrlHandler**

SetProcessWorkingSetSize

The **SetProcessWorkingSetSize** function sets the minimum and maximum working set sizes for a specified process.

The working set of a process is the set of memory pages currently visible to the process in physical RAM memory. These pages are resident and available for an application to use without triggering a page fault. The size of the working set of a process is specified in bytes. The minimum and maximum working set sizes affect the virtual memory paging behavior of a process.

```
BOOL SetProcessWorkingSetSize(
  HANDLE hProcess,                  // handle to process
  SIZE_T dwMinimumWorkingSetSize,   // minimum working set size
  SIZE_T dwMaximumWorkingSetSize    // maximum working set size
);
```

Parameters

hProcess

[in] Handle to the process whose working set sizes is to be set.

Windows NT/2000: The handle must have PROCESS_SET_QUOTA access rights. For more information, see *Process Security and Access Rights*.

dwMinimumWorkingSetSize

[in] Specifies a minimum working set size for the process. The virtual memory manager attempts to keep at least this much memory resident in the process whenever the process is active.

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value –1, the function temporarily trims the working set of the specified process to zero. This essentially swaps the process out of physical RAM memory.

dwMaximumWorkingSetSize

[in] Specifies a maximum working set size for the process. The virtual memory manager attempts to keep no more than this much memory resident in the process whenever the process is active and memory is in short supply.

If both *dwMinimumWorkingSetSize* and *dwMaximumWorkingSetSize* have the value –1, the function temporarily trims the working set of the specified process to zero. This essentially swaps the process out of physical RAM memory.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. Call **GetLastError** to obtain extended error information.

Remarks

The working set of the specified process can be emptied by specifying the value –1 for both the minimum and maximum working set sizes.

If the values of either *dwMinimumWorkingSetSize* or *dwMaximumWorkingSetSize* are greater than the process' current working set sizes, the specified process must have the SE_INC_BASE_PRIORITY_NAME privilege. Users in the Administrators and Power Users groups generally have this privilege. For more information about security privileges, see *Privileges*.

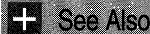
The operating system allocates working set sizes on a first-come, first-served basis. For example, if an application successfully sets 40 megabytes as its minimum working set size on a 64-megabyte system, and a second application requests a 40-megabyte working set size, the operating system denies the second application's request.

Using the **SetProcessWorkingSetSize** function to set an application's minimum and maximum working set sizes does not guarantee that the requested memory will be reserved, or that it will remain resident at all times. When the application is idle, or a low-memory situation causes a demand for memory, the operating system can reduce the application's working set. An application can use the **VirtualLock** function to lock ranges of the application's virtual address space in memory; however, that can potentially degrade the performance of the system.

When you increase the working set size of an application, you are taking away physical memory from the rest of the system. This can degrade the performance of other applications and the system as a whole. It can also lead to failures of operations that require physical memory to be present; for example, creating processes, threads, and kernel pool. Thus, you must use the **SetProcessWorkingSetSize** function carefully. You must always consider the performance of the whole system when you are designing an application.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.



Processes and Threads Overview, *Process and Thread Functions*, **GetProcessWorkingSetSize**, **VirtualLock**

SetThreadAffinityMask

The **SetThreadAffinityMask** function sets a processor affinity mask for a specified thread.

A thread affinity mask is a bit vector in which each bit represents the processors that a thread is allowed to run on.

A thread affinity mask must be a proper subset of the process affinity mask for the containing process of a thread. A thread is only allowed to run on the processors its process is allowed to run on.

```
DWORD_PTR SetThreadAffinityMask (
  HANDLE hThread,                    // handle to thread
  DWORD_PTR dwThreadAffinityMask     // thread affinity mask
);
```

Parameters

hThread

[in] Handle to the thread whose affinity mask is to be set.

Windows NT/2000: This handle must have the THREAD_SET_INFORMATION access right associated with it. For more information, see *Thread Security and Access Rights*.

dwThreadAffinityMask

Windows NT/2000: [in] Specifies an affinity mask for the thread.

Windows 95/98: [in] This value must be 1.

Return Values

If the function succeeds, the return value is nonzero.

Windows NT/2000: The return value is the thread's previous affinity mask.

Windows 95/98: The return value is 1. To succeed, *hThread* must be valid and *dwThreadAffinityMask* must be 1.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **GetProcessAffinityMask**, **OpenThread**, **SetThreadIdealProcessor**

SetThreadIdealProcessor

The **SetThreadIdealProcessor** function is used to specify a preferred processor for a thread. The system schedules threads on their preferred processors whenever possible.

```
DWORD SetThreadIdealProcessor(
  HANDLE hThread,          // handle to the thread
  DWORD dwIdealProcessor   // ideal processor number
);
```

Parameters

hThread

[in] Handle to the thread whose preferred processor is to be set. The handle must have the THREAD_SET_INFORMATION access right associated with it. For more information, see *Thread Security and Access Rights*.

dwIdealProcessor

[in] Specifies the number of the preferred processor for the thread. A value of MAXIMUM_PROCESSORS tells the system that the thread has no preferred processor.

Return Values

If the function succeeds, the return value is the previous preferred processor or MAXIMUM_PROCESSORS if the thread does not have a preferred processor.

If the function fails, the return value is –1. To get extended error information, call **GetLastError**.

Remarks

You can use the **GetSystemInfo** function to determine the number of processors on the computer. You can also use the **GetProcessAffinityMask** function to check the processors on which the thread is allowed to run. Note that **GetProcessAffinityMask** returns a bit mask whereas **SetThreadIdealProcessor** uses an integer value to represent the processor.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **GetProcessAffinityMask**, **GetSystemInfo**, **OpenThread**, **SetThreadAffinityMask**

SetThreadPriority

The **SetThreadPriority** function sets the priority value for the specified thread. This value, together with the priority class of the thread's process, determines the thread's base priority level.

```
BOOL SetThreadPriority(
  HANDLE hThread, // handle to the thread
  int nPriority   // thread priority level
);
```

Parameters

hThread

[in] Handle to the thread whose priority value is to be set.

Windows NT/2000: The handle must have the THREAD_SET_INFORMATION access right associated with it. For more information, see *Thread Security and Access Rights*.

nPriority

[in] Specifies the priority value for the thread. This parameter can be one of the following values:

Priority	Meaning
THREAD_PRIORITY_ABOVE_NORMAL	Indicates 1 point above normal priority for the priority class.
THREAD_PRIORITY_BELOW_NORMAL	Indicates 1 point below normal priority for the priority class.
THREAD_PRIORITY_HIGHEST	Indicates 2 points above normal priority for the priority class.
THREAD_PRIORITY_IDLE	Indicates a base priority level of 1 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 16 for REALTIME_PRIORITY_CLASS processes.

(continued)

(continued)

Priority	Meaning
THREAD_PRIORITY_LOWEST	Indicates 2 points below normal priority for the priority class.
THREAD_PRIORITY_NORMAL	Indicates normal priority for the priority class.
THREAD_PRIORITY_TIME_CRITICAL	Indicates a base priority level of 15 for IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, or HIGH_PRIORITY_CLASS processes, and a base priority level of 31 for REALTIME_PRIORITY_CLASS processes.

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

Every thread has a base priority level determined by the thread's priority value and the priority class of its process. The system uses the base priority level of all executable threads to determine which thread gets the next slice of CPU time. Threads are scheduled in a round-robin fashion at each priority level, and only when there are no executable threads at a higher level does scheduling of threads at a lower level take place.

The **SetThreadPriority** function enables setting the base priority level of a thread relative to the priority class of its process. For example, specifying THREAD_PRIORITY_HIGHEST in a call to **SetThreadPriority** for a thread of an IDLE_PRIORITY_CLASS process sets the thread's base priority level to 6. For a table that shows the base priority levels for each combination of priority class and thread priority value, see *Scheduling Priorities*.

For IDLE_PRIORITY_CLASS, BELOW_NORMAL_PRIORITY_CLASS, NORMAL_PRIORITY_CLASS, ABOVE_NORMAL_PRIORITY_CLASS, and HIGH_PRIORITY_CLASS processes, the system dynamically boosts a thread's base priority level when events occur that are important to the thread. REALTIME_PRIORITY_CLASS processes do not receive dynamic boosts.

All threads initially start at THREAD_PRIORITY_NORMAL. Use the **GetPriorityClass** and **SetPriorityClass** functions to get and set the priority class of a process. Use the **GetThreadPriority** function to get the priority value of a thread.

Use the priority class of a process to differentiate between applications that are time critical and those that have normal or below normal scheduling requirements. Use thread priority values to differentiate the relative priorities of the tasks of a process. For

example, a thread that handles input for a window could have a higher priority level than a thread that performs intensive calculations for the CPU.

When manipulating priorities, be very careful to ensure that a high-priority thread does not consume all of the available CPU time. A thread with a base priority level above 11 interferes with the normal operation of the operating system. Using REALTIME_PRIORITY_CLASS may cause disk caches to not flush, hang the mouse, and so on.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **GetPriorityClass**, **GetThreadPriority**, **SetPriorityClass**

SetThreadPriorityBoost

The **SetThreadPriorityBoost** function disables the ability of the system to temporarily boost the priority of a thread.

```
BOOL SetThreadPriorityBoost(
  HANDLE hThread,              // handle to thread
  BOOL DisablePriorityBoost    // priority boost state
);
```

Parameters

hThread

[in] Handle to the thread whose priority is to be boosted. This thread must have the THREAD_SET_INFORMATION access right associated with it. For more information, see *Thread Security and Access Rights.*

DisablePriorityBoost

[in] Specifies the priority boost control state. A value of TRUE indicates that dynamic boosting is to be disabled. A value of FALSE restores normal behavior.

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 thread is running in one of the dynamic priority classes, the system temporarily boosts the thread's priority when it is taken out of a wait state. If **SetThreadPriorityBoost** is called with the *DisablePriorityBoost* parameter set to TRUE, the thread's priority is not boosted. To restore normal behavior, call **SetThreadPriorityBoost** with *DisablePriorityBoost* set to FALSE.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **OpenThread**, **GetThreadPriorityBoost**

Sleep

The **Sleep** function suspends the execution of the current thread for a specified interval.

To enter an alertable wait state, use the **SleepEx** function.

```
VOID Sleep(
  DWORD dwMilliseconds   // sleep time
);
```

Parameters

dwMilliseconds
[in] Specifies the time, in milliseconds, for which to suspend execution. A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. A value of INFINITE causes an infinite delay.

Return Values

This function does not return a value.

Remarks

A thread can relinquish the remainder of its time slice by calling this function with a sleep time of zero milliseconds.

You have to be careful when using **Sleep** and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **Sleep** with infinite delay, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than **Sleep**.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development.*

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **MsgWaitForMultipleObjects**, **MsgWaitForMultipleObjectsEx**, **SleepEx**

SleepEx

The **SleepEx** function causes the current thread to enter a wait state until one of the following occurs:

- An I/O completion callback function is called
- An asynchronous procedure call (APC) is queued to the thread.
- The time-out interval elapses

```
DWORD SleepEx(
  DWORD dwMilliseconds,  // time-out interval
  BOOL bAlertable        // early completion option
);
```

Parameters

dwMilliseconds

[in] Specifies the time, in milliseconds, that the delay is to occur. A value of zero causes the thread to relinquish the remainder of its time slice to any other thread of equal priority that is ready to run. If there are no other threads of equal priority ready to run, the function returns immediately, and the thread continues execution. A value of INFINITE causes an infinite delay.

bAlertable

[in] Specifies whether the function may terminate early due to an I/O completion callback function or an APC. If *bAlertable* is FALSE, the function does not return until the time-out period has elapsed. If an I/O completion callback occurs, the function does not return and the I/O completion function is not executed. If an APC is queued to the thread, the function does not return and the APC function is not executed.

If *bAlertable* is TRUE and the thread that called this function is the same thread that called the extended I/O function (**ReadFileEx** or **WriteFileEx**), the function returns when either the time-out period has elapsed or when an I/O completion callback function occurs. If an I/O completion callback occurs, the I/O completion function is called. If an APC is queued to the thread (**QueueUserAPC**), the function returns when either the timer-out period has elapsed or when the APC function is called.

Return Values

The return value is zero if the specified time interval expired.

The return value is WAIT_IO_COMPLETION if the function returned due to one or more I/O completion callback functions. This can happen only if *bAlertable* is TRUE, and if the thread that called the **SleepEx** function is the same thread that called the extended I/O function.

Remarks

This function can be used with the **ReadFileEx** or **WriteFileEx** functions to suspend a thread until an I/O operation has been completed. These functions specify a completion routine that is to be executed when the I/O operation has been completed. For the completion routine to be executed, the thread that called the I/O function must be in an alertable wait state when the completion callback function occurs. A thread goes into an alertable wait state by calling either **SleepEx**, **MsgWaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, or **WaitForMultipleObjectsEx**, with the function's *bAlertable* parameter set to TRUE.

A thread can relinquish the remainder of its time slice by calling this function with a sleep time of zero milliseconds.

You have to be careful when using **SleepEx** and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses **SleepEx** with infinite delay, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than **SleepEx**.

! Requirements

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.

Processes and Threads Overview, Process and Thread Functions, **MsgWaitForMultipleObjects**, **MsgWaitForMultipleObjectsEx**, **QueueUserAPC**, **ReadFileEx**, **Sleep**, **WaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, **WriteFileEx**

SuspendThread

The **SuspendThread** function suspends the specified thread.

```
DWORD SuspendThread(
  HANDLE hThread   // handle to thread
);
```

Parameters

hThread

[in] Handle to the thread that is to be suspended.

Windows NT/2000: The handle must have THREAD_SUSPEND_RESUME access. For more information, see *Thread Security and Access Rights.*

Return Values

If the function succeeds, the return value is the thread's previous suspend count; otherwise, it is –1. To get extended error information, use the **GetLastError** function.

Remarks

If the function succeeds, execution of the specified thread is suspended and the thread's suspend count is incremented.

Suspending a thread causes the thread to stop executing user-mode (application) code.

Each thread has a suspend count (with a maximum value of MAXIMUM_SUSPEND_COUNT). If the suspend count is greater than zero, the thread is suspended; otherwise, the thread is not suspended and is eligible for execution. Calling **SuspendThread** causes the target thread's suspend count to be incremented. Attempting to increment past the maximum suspend count causes an error without incrementing the count.

The **ResumeThread** function decrements the suspend count of a suspended thread.

Requirements

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

Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **OpenThread**, **ResumeThread**

SwitchToFiber

The **SwitchToFiber** function schedules a fiber. The caller of must be a fiber.

```
VOID SwitchToFiber(
  LPVOID lpFiber  // fiber to schedule
);
```

Parameters

lpFiber
[in] Specifies the address of the fiber to schedule.

Return Values

This function does not return a value.

Remarks

You create fibers with **CreateFiber**. Before you can schedule fibers associated with a thread, you must call **ConvertThreadToFiber** to set up an area in which to save the fiber state information. The thread is now the currently executing fiber.

The **SwitchToFiber** function saves the state information of the current fiber and restores the state of the specified fiber. You can call **SwitchToFiber** with the address of a fiber created by a different thread. To do this, you must have the address returned to the other thread when it called **CreateFiber** and you must use proper synchronization.

Warning Avoid making the following call:

```
SwitchToFiber( GetCurrentFiber() );
```

This call may cause unpredictable problems.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CreateFiber**, **ConvertThreadToFiber**

SwitchToThread

The **SwitchToThread** function causes the calling thread to yield execution to another thread that is ready to run on the current processor. The operating system selects the thread to yield to.

```
BOOL SwitchToThread(VOID);
```

Parameters

This function has no parameters.

Return Values

If calling the **SwitchToThread** function causes the operating system to switch execution to another thread, the return value is nonzero.

If there are no other threads ready to execute, the operating system does not switch execution to another thread, and the return value is zero.

Remarks

The yield of execution is in effect for up to one thread-scheduling time slice. After that, the operating system reschedules execution for the yielding thread. The rescheduling is determined by the priority of the yielding thread and the status of other threads that are available to run.

Note The yield of execution is limited to the processor of the calling thread. The operating system will not switch execution to another processor, even if that processor is idle or is running a thread of lower priority.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **SuspendThread**

TerminateJobObject

The **TerminateJobObject** function terminates all processes currently associated with the job.

```
BOOL TerminateJobObject(
  HANDLE hJob,     // handle to job
  UINT uExitCode   // exit code
);
```

Parameters

hJob

[in] Handle to the job whose processes will be terminated. The **CreateJobObject** or **OpenJobObject** function returns this handle. This handle must have the JOB_OBJECT_TERMINATE access right. For more information, see *Job Object Security and Access Rights*.

uExitCode

[in] Specifies the exit code for the processes and threads terminated as a result of this call.

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 not possible for any of the processes associated with the job to postpone or handle the termination. It is as if **TerminateProcess** were called for each process associated with the job.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Processes and Threads Overview, *Process and Thread Functions*, **CreateJobObject**, **OpenJobObject**, **TerminateProcess**

TerminateProcess

The **TerminateProcess** function terminates the specified process and all of its threads.

```
BOOL TerminateProcess(
  HANDLE hProcess, // handle to the process
  UINT uExitCode   // exit code for the process
);
```

Parameters

hProcess

[in] Handle to the process to terminate.

Windows NT/2000: The handle must have PROCESS_TERMINATE access. For more information, see *Process Security and Access Rights*.

uExitCode

[in] Specifies the exit code for the process and for all threads terminated as a result of this call. Use the **GetExitCodeProcess** function to retrieve the process's exit value. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

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 **TerminateProcess** function is used to unconditionally cause a process to exit. Use it only in extreme circumstances. The state of global data maintained by dynamic-link libraries (DLLs) may be compromised if **TerminateProcess** is used rather than **ExitProcess**.

TerminateProcess causes all threads within a process to terminate, and causes a process to exit, but DLLs attached to the process are not notified that the process is terminating.

Terminating a process causes the following:

1. All of the object handles opened by the process are closed.
2. All of the threads in the process terminate their execution.
3. The state of the process object becomes signaled, satisfying any threads that had been waiting for the process to terminate.

4. The states of all threads of the process become signaled, satisfying any threads that had been waiting for the threads to terminate.
5. The termination status of the process changes from STILL_ACTIVE to the exit value of the process.

Terminating a process does not cause child processes to be terminated.

Terminating a process does not necessarily remove the process object from the system. A process object is deleted when the last handle to the process is closed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **ExitProcess**, **OpenProcess**, **GetExitCodeProcess**, **GetExitCodeThread**

TerminateThread

The **TerminateThread** function terminates a thread.

```
BOOL TerminateThread(
  HANDLE hThread,     // handle to thread
  DWORD dwExitCode    // exit code
);
```

Parameters

hThread

[in/out] Handle to the thread to terminate.

Windows NT/2000: The handle must have THREAD_TERMINATE access. For more information, see *Thread Security and Access Rights*.

dwExitCode

[in] Specifies the exit code for the thread. Use the **GetExitCodeThread** function to retrieve a thread's exit value.

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

TerminateThread is used to cause a thread to exit. When this occurs, the target thread has no chance to execute any user-mode code and its initial stack is not deallocated. DLLs attached to the thread are not notified that the thread is terminating.

TerminateThread is a dangerous function that should only be used in the most extreme cases. You should call **TerminateThread** only if you know exactly what the target thread is doing, and you control all of the code that the target thread could possibly be running at the time of the termination. For example, **TerminateThread** can result in the following problems:

- If the target thread owns a critical section, the critical section will not be released.
- If the target thread is executing certain kernel32 calls when it is terminated, the kernel32 state for the thread's process could be inconsistent.
- If the target thread is manipulating the global state of a shared DLL, the state of the DLL could be destroyed, affecting other users of the DLL.

A thread cannot protect itself against **TerminateThread**, other than by controlling access to its handles. The thread handle returned by the **CreateThread** and **CreateProcess** functions has THREAD_TERMINATE access, so any caller holding one of these handles can terminate your thread.

If the target thread is the last thread of a process when this function is called, the thread's process is also terminated.

The state of the thread object becomes signaled, releasing any other threads that had been waiting for the thread to terminate. The thread's termination status changes from STILL_ACTIVE to the value of the *dwExitCode* parameter.

Terminating a thread does not necessarily remove the thread object from the system. A thread object is deleted when the last thread handle is closed.

! Requirements

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.

+ See Also

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**, **CreateThread**, **ExitThread**, **GetExitCodeThread**, **OpenThread**

ThreadProc

The **ThreadProc** function is an application-defined function that serves as the starting address for a thread. Specify this address when calling the **CreateThread** or **CreateRemoteThread** function. The **LPTHREAD_START_ROUTINE** type defines a pointer to this callback function. **ThreadProc** is a placeholder for the application-defined function name.

```
DWORD WINAPI ThreadProc(
  LPVOID lpParameter   // thread data
);
```

Parameters

lpParameter

[in] Receives the thread data passed to the function using the *lpParameter* parameter of the **CreateThread** or **CreateRemoteThread** function.

Return Values

The function should return a value that indicates its success or failure.

Remarks

A process can obtain the return value of the **ThreadProc** of a thread it created with **CreateThread** by calling the **GetExitCodeThread** function. A process cannot obtain the return value from the **ThreadProc** of a thread it created with **CreateRemoteThread**.

Requirements

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.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **CreateThread**, **CreateRemoteThread**, **GetExitCodeThread**

TlsAlloc

The **TlsAlloc** function allocates a thread local storage (TLS) index. Any thread of the process can subsequently use this index to store and retrieve values that are local to the thread.

```
DWORD TlsAlloc(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a TLS index.

If the function fails, the return value is –1. To get extended error information, call **GetLastError**.

Remarks

The threads of the process can use the TLS index in subsequent calls to the **TlsFree**, **TlsSetValue**, or **TlsGetValue** functions.

TLS indexes are typically allocated during process or dynamic-link library (DLL) initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot. To store a value in its slot, a thread specifies the index in a call to **TlsSetValue**. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

The constant TLS_MINIMUM_AVAILABLE defines the minimum number of TLS indexes available in each process. This minimum is guaranteed to be at least 64 for all systems.

Windows 2000: There is a limit of 1088 TLS indexes per process.

Windows NT 4.0 and earlier: There is a limit of 64 TLS indexes per process.

TLS indexes are not valid across process boundaries. A DLL cannot assume that an index assigned in one process is valid in another process.

A DLL might use **TlsAlloc**, **TlsSetValue**, **TlsGetValue**, and **TlsFree** as follows:

- When a DLL attaches to a process, the DLL uses **TlsAlloc** to allocate a TLS index. The DLL then allocates some dynamic storage and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the initial thread of the process. The TLS index is stored in a global or static variable of the DLL.
- Each time the DLL attaches to a new thread of the process, the DLL allocates some dynamic storage for the new thread and uses the TLS index in a call to **TlsSetValue** to store the address in the TLS slot. This concludes the per-thread initialization for the new thread.
- Each time an initialized thread makes a DLL call requiring the data in its dynamic storage, the DLL uses the TLS index in a call to **TlsGetValue** to retrieve the address of the dynamic storage for that thread.

For additional information on thread local storage, see *Thread Local Storage*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **TlsFree**, **TlsGetValue**, **TlsSetValue**

TlsFree

The **TlsFree** function releases a thread local storage (TLS) index, making it available for reuse.

```
BOOL TlsFree(
  DWORD dwTlsIndex   // TLS index
);
```

Parameters

dwTlsIndex
[in] Specifies a TLS index that was allocated by the **TlsAlloc** 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

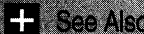
If the threads of the process have allocated dynamic storage and used the TLS index to store pointers to this storage, they should free the storage before calling **TlsFree**. The **TlsFree** function does not free any dynamic storage that has been associated with the TLS index. It is expected that DLLs call this function (if at all) only during their process detach routine.

For a brief discussion of typical uses of the TLS functions, see the Remarks section of the **TlsAlloc** 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 winbase.h; include windows.h.
Library: Use kernel32.lib.



Processes and Threads Overview, *Process and Thread Functions*, **TlsAlloc**, **TlsGetValue**, **TlsSetValue**

TlsGetValue

The **TlsGetValue** function retrieves the value in the calling thread's thread local storage (TLS) slot for a specified TLS index. Each thread of a process has its own slot for each TLS index.

```
LPVOID TlsGetValue(
  DWORD dwTlsIndex    // TLS index
);
```

Parameters

dwTlsIndex
[in] Specifies a TLS index that was allocated by the **TlsAlloc** function.

Return Values

If the function succeeds, the return value is the value stored in the calling thread's TLS slot associated with the specified index.

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

Note The data stored in a TLS slot can have a value of zero. In this case, the return value is zero and **GetLastError** returns NO_ERROR.

Remarks

TLS indexes are typically allocated by the **TlsAlloc** function during process or DLL initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot for that index. The storage slot for each thread is initialized to NULL. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

TlsSetValue and **TlsGetValue** were implemented with speed as the primary goal. These functions perform minimal parameter validation and error checking. In particular, this

function succeeds if *dwTlsIndex* is in the range 0 through (TLS_MINIMUM_AVAILABLE–1). It is up to the programmer to ensure that the index is valid.

Win32 functions that return indications of failure call **SetLastError** when they fail. They generally do not call **SetLastError** when they succeed. The **TlsGetValue** function is an exception to this general rule. The **TlsGetValue** function calls **SetLastError** to clear a thread's last error when it succeeds. That allows checking for the error-free retrieval of NULL 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **GetLastError**, **SetLastError**, **TlsAlloc**, **TlsFree**, **TlsSetValue**

TlsSetValue

The **TlsSetValue** function stores a value in the calling thread's thread local storage (TLS) slot for a specified TLS index. Each thread of a process has its own slot for each TLS index.

```
BOOL TlsSetValue(
  DWORD dwTlsIndex,   // TLS index
  LPVOID lpTlsValue   // value to store
);
```

Parameters

dwTlsIndex
[in] Specifies a TLS index that was allocated by the **TlsAlloc** function.

lpTlsValue
[in] Specifies the value to be stored in the calling thread's TLS slot specified by *dwTlsIndex*.

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

TLS indexes are typically allocated by the **TlsAlloc** function during process or DLL initialization. Once allocated, each thread of the process can use a TLS index to access its own TLS storage slot for that index. The storage slot for each thread is initialized to NULL. A thread specifies a TLS index in a call to **TlsSetValue**, to store a value in its slot. The thread specifies the same index in a subsequent call to **TlsGetValue**, to retrieve the stored value.

TlsSetValue and **TlsGetValue** were implemented with speed as the primary goal. These functions perform minimal parameter validation and error checking. In particular, this function succeeds if *dwTlsIndex* is in the range 0 through (TLS_MINIMUM_AVAILABLE–1). It is up to the programmer to ensure that the index is 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Processes and Threads Overview, *Process and Thread Functions*, **TlsAlloc**, **TlsFree**, **TlsGetValue**

UserHandleGrantAccess

The **UserHandleGrantAccess** function grants or denies access to a handle to a User object to a job that has a user-interface restriction. When access is granted, all processes associated with the job can subsequently recognize and use the handle. When access is denied, the processes can no longer use the handle. For more information see *User Objects*.

```
BOOL UserHandleGrantAccess(
  HANDLE hUserHandle,  // handle to User object
  HANDLE hJob,         // handle to job
  BOOL bGrant          // access granted or denied
);
```

Parameters

hUserHandle

[in] Handle to a User object.

hJob
[in] Handle to the job to be granted access to the User handle. The **CreateJobObject** or **OpenJobObject** function returns this handle.

bGrant
[in] Specifies whether access is to be denied or granted to *hUserHandle*. If *bGrant* is TRUE, all processes associated with the job can recognize and use the handle. If *bGrant* is FALSE, the processes can no longer use the handle.

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 **UserHandleGrantAccess** function can be called only from a process not associated with the job specified by the *hJob* parameter. The User handle must not be owned by a process or thread associated with the job.

To create user-interface restrictions, call the **SetInformationJobObject** function with the JobObjectBasicUIRestrictions job information class.

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

Processes and Threads Overview, *Process and Thread Functions*, **CreateJobObject**, **OpenJobObject**, **SetInformationJobObject**

WaitForInputIdle

The **WaitForInputIdle** function waits until the specified process is waiting for user input with no input pending, or until the time-out interval has elapsed.

```
DWORD WaitForInputIdle(
  HANDLE hProcess,       // handle to process
  DWORD dwMilliseconds   // time-out interval
);
```

Parameters

hProcess

[in] Handle to the process. If this process is a console application or does not have a message queue, **WaitForInputIdle** returns immediately.

dwMilliseconds

[in] Specifies the time-out interval, in milliseconds. If *dwMilliseconds* is INFINITE, the function does not return until the process is idle.

Return Values

The following table shows the possible return values:

Value	Meaning
0	The wait was satisfied successfully.
WAIT_TIMEOUT	The wait was terminated because the time-out interval elapsed.
–1	An error occurred. To get extended error information, use the **GetLastError** function.

Remarks

The **WaitForInputIdle** function enables a thread to suspend its execution until a specified process has finished its initialization and is waiting for user input with no input pending. This can be useful for synchronizing a parent process and a newly created child process. When a parent process creates a child process, the **CreateProcess** function returns without waiting for the child process to finish its initialization. Before trying to communicate with the child process, the parent process can use **WaitForInputIdle** to determine when the child's initialization has been completed. For example, the parent process should use **WaitForInputIdle** before trying to find a window associated with the child process.

The **WaitForInputIdle** function can be used at any time, not just during application startup.

Requirements

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

Processes and Threads Overview, *Process and Thread Functions*, **CreateProcess**

Process and Thread Structures

IO_COUNTERS

The **IO_COUNTERS** structure contains I/O accounting information for a process or a job object. For a job object, the counters include all operations performed by all processes that have ever been associated with the job, in addition to all processes currently associated with the job.

```
typedef struct _IO_COUNTERS {
    ULONGLONG  ReadOperationCount;
    ULONGLONG  WriteOperationCount;
    ULONGLONG  OtherOperationCount;
    ULONGLONG  ReadTransferCount;
    ULONGLONG  WriteTransferCount;
    ULONGLONG  OtherTransferCount;
} IO_COUNTERS;
typedef IO_COUNTERS *PIO_COUNTERS;
```

Members

ReadOperationCount
Specifies the number of read operations performed.

WriteOperationCount
Specifies the number of write operations performed.

OtherOperationCount
Specifies the number of I/O operations performed, other than read and write operations.

ReadTransferCount
Specifies the number of bytes read.

WriteTransferCount
Specifies the number of bytes written.

OtherTransferCount
Specifies the number of bytes transferred during operations other than read and write operations.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

Processes and Threads Overview, *Process and Thread Structures*, **GetProcessIoCounters**, **JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION**

JOBOBJECT_ASSOCIATE_COMPLETION_PORT

The **JOBOBJECT_ASSOCIATE_COMPLETION_PORT** structure contains information used to associate a completion port with a job. You can associate one completion port with a job. There is no way to terminate the association and no way to associate a different port with the job.

```
typedef struct _JOBOBJECT_ASSOCIATE_COMPLETION_PORT {
    PVOID CompletionKey;
    HANDLE CompletionPort;
} JOBOBJECT_ASSOCIATE_COMPLETION_PORT, PJOBOBJECT_ASSOCIATE_COMPLETION_PORT;
```

Members

CompletionKey

Specifies the value to use in the *dwCompletionKey* parameter of **PostQueuedCompletionStatus** when messages are sent on behalf of the job.

CompletionPort

Specifies the completion port to use in the *CompletionPort* parameter of the **PostQueuedCompletionStatus** function when messages are sent on behalf of the job.

Remarks

The job sends messages to the completion port when certain events occur. All messages are sent directly from the job as if the job had called the **PostQueuedCompletionStatus** function. A thread monitoring the completion port using the **GetQueuedCompletionStatus** function must pick up the messages. The thread receives information in the **GetQueuedCompletionStatus** parameters shown in the following table.

Parameter	Information Received
lpCompletionKey	The value specified in **CompletionKey** during the completion-port association. If a completion port is associated with multiple jobs, **CompletionKey** should help the caller determine which completion port is sending a message.
lpOverlapped	Message-specific value. For more information, see the following table of message identifiers.

LpNumberOfBytesTransfered	The message identifier that indicates which job-related event occurred. For more information, see the following table of message identifiers.

The following messages can be sent to the completion port.

Message Identifier	Description
JOB_OBJECT_MSG_END_OF_JOB_TIME	Indicates that the JOB_OBJECT_POST_AT_END_OF_JOB option is in effect and the end-of-job time limit has been reached. Upon posting this message, the time limit is canceled and the job's processes can continue to run. The value of *lpOverlapped* is NULL.
JOB_OBJECT_MSG_END_OF_PROCESS_TIME	Indicates that a process has exceeded a per-process time limit. The system sends this message after the process termination has been requested. The value of *lpOverlapped* is the identifier of the process that exceeded its limit.
JOB_OBJECT_MSG_ACTIVE_PROCESS_LIMIT	Indicates that the active process limit has been exceeded. The value of *lpOverlapped* is NULL.
JOB_OBJECT_MSG_ACTIVE_PROCESS_ZERO	Indicates that the active process count has been decremented to 0. For example, if the job currently has two active processes, the system sends this message after they both terminate. The value of *lpOverlapped* is NULL.
JOB_OBJECT_MSG_NEW_PROCESS	Indicates that a process has been added to the job. Processes added to a job at the time a completion port is associated are also reported. The value of *lpOverlapped* is the identifier of the process added to the job.
JOB_OBJECT_MSG_EXIT_PROCESS	Indicates that a process associated with the job has exited. The value of *lpOverlapped* is the identifier of the exiting process.

Message Identifier	Description
JOB_OBJECT_MSG_ABNORMAL_EXIT_PROCESS	Indicates that a process associated with the job exited with an exit code that indicates an abnormal exit (see the list following this table). The value of *lpOverlapped* is the identifier of the exiting process.
JOB_OBJECT_MSG_PROCESS_MEMORY_LIMIT	Indicates that a process associated with the job has exceeded its memory limit (if one is in effect). The value of *lpOverlapped* is the identifier of the process that has exceeded its limit. The system does not send this message if the process has not yet reported its process identifier.
JOB_OBJECT_MSG_JOB_MEMORY_LIMIT	Indicates that a process associated with the job caused the job to exceed the job-wide memory limit (if one is in effect). The value of *lpOverlapped* specifies the identifier of the process that has attempted to exceed the limit. The system does not send this message if the process has not yet reported its process identifier.

The following exit codes indicate an abnormal exit:

STATUS_ACCESS_VIOLATION
STATUS_ARRAY_BOUNDS_EXCEEDED
STATUS_BREAKPOINT
STATUS_CONTROL_C_EXIT
STATUS_DATATYPE_MISALIGNMENT
STATUS_FLOAT_DENORMAL_OPERAND
STATUS_FLOAT_DIVIDE_BY_ZERO
STATUS_FLOAT_INEXACT_RESULT
STATUS_FLOAT_INVALID_OPERATION
STATUS_FLOAT_MULTIPLE_FAULTS
STATUS_FLOAT_MULTIPLE_TRAPS
STATUS_FLOAT_OVERFLOW
STATUS_FLOAT_STACK_CHECK
STATUS_FLOAT_UNDERFLOW
STATUS_GUARD_PAGE_VIOLATION
STATUS_ILLEGAL_INSTRUCTION

STATUS_ILLEGAL_VLM_REFERENCE
STATUS_IN_PAGE_ERROR
STATUS_INVALID_DISPOSITION
STATUS_INTEGER_DIVIDE_BY_ZERO
STATUS_INTEGER_OVERFLOW
STATUS_NONCONTINUABLE_EXCEPTION
STATUS_PRIVILEGED_INSTRUCTION
STATUS_REG_NAT_CONSUMPTION
STATUS_SINGLE_STEP
STATUS_STACK_OVERFLOW

You must be cautious when using the JOB_OBJECT_MSG_NEW_PROCESS and JOB_OBJECT_MSG_EXIT_PROCESS messages, as race conditions may occur. For instance, if processes are actively starting and exiting within a job, and you are in the process of assigning a completion port to the job, you may miss messages for processes whose states change during the association of the completion port. For this reason, it is best to associate a completion port with a job when the job is inactive.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

+ See Also

Processes and Threads Overview, *Process and Thread Structures*, **PostQueuedCompletionStatus**, **QueryInformationJobObject**, **SetInformationJobObject**

JOBOBJECT_BASIC_ACCOUNTING_INFORMATION

The **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION** structure contains basic accounting information for a job object.

```
typedef struct _JOBOBJECT_BASIC_ACCOUNTING_INFORMATION {
    LARGE_INTEGER TotalUserTime;
    LARGE_INTEGER TotalKernelTime;
    LARGE_INTEGER ThisPeriodTotalUserTime;
    LARGE_INTEGER ThisPeriodTotalKernelTime;
    DWORD TotalPageFaultCount;
    DWORD TotalProcesses;
    DWORD ActiveProcesses;
    DWORD TotalTerminatedProcesses;
```

```
} JOBOBJECT_BASIC_ACCOUNTING_INFORMATION,
*PJOBOBJECT_BASIC_ACCOUNTING_INFORMATION;
```

Members

TotalUserTime

Specifies the total amount of user-mode execution time, in 100-nanosecond ticks, for all active processes associated with the job, as well as all terminated processes no longer associated with the job.

TotalKernelTime

Specifies the total amount of kernel-mode execution time, in 100-nanosecond ticks, for all active processes associated with the job, as well as all terminated processes no longer associated with the job.

ThisPeriodTotalUserTime

Specifies the total amount of user-mode execution time, in 100-nanosecond ticks, for all active processes associated with the job (as well as all terminated processes no longer associated with the job) since the last call that set a per-job user-mode time limit.

This member is set to 0 on creation of the job, and each time a per-job user-mode time limit is established.

ThisPeriodTotalKernelTime

Specifies the total amount of kernel-mode execution time, in 100-nanosecond ticks, for all active processes associated with the job (as well as all terminated processes no longer associated with the job) since the last call that set a per-job kernel-mode time limit.

This member is set to 0 on creation of the job, and each time a per-job kernel-mode time limit is established.

TotalPageFaultCount

Specifies the total number of page faults encountered by all active processes associated with the job, as well as all terminated processes no longer associated with the job.

TotalProcesses

Specifies the total number of processes associated with the job during its lifetime, including those that have terminated. For example, when a process is associated with a job, but the association fails because of a limit violation, this value is incremented.

ActiveProcesses

Specifies the total number of processes currently associated with the job. When a process is associated with a job, but the association fails because of a limit violation, this value is temporarily incremented. When the terminated process exits and all references to the process are released, this value is decremented.

TotalTerminatedProcesses

Specifies the total number of processes terminated because of a limit violation.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **QueryInformationJobObject**, **SetInformationJobObject**

JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION

The **JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION** structure contains basic accounting and I/O accounting information for a job object.

```
typedef struct JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION {
    JOBOBJECT_BASIC_ACCOUNTING_INFORMATION BasicInfo;
    IO_COUNTERS IoInfo;
} JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION,
*PJOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION;
```

Members

BasicInfo
A **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION** structure that specifies the basic accounting information for the job.

IoInfo
An **IO_COUNTERS** structure that specifies the I/O accounting information for the job. The structure includes information for all processes that have ever been associated with the job, in addition to the information for all processes currently associated with the job.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **IO_COUNTERS**, **JOBOBJECT_BASIC_ACCOUNTING_INFORMATION**, **QueryInformationJobObject**

JOBOBJECT_BASIC_LIMIT_INFORMATION

The **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure contains basic limit information for a job object.

```
typedef struct _JOBOBJECT_BASIC_LIMIT_INFORMATION {
    LARGE_INTEGER PerProcessUserTimeLimit;
    LARGE_INTEGER PerJobUserTimeLimit;
    DWORD LimitFlags;
    SIZE_T MinimumWorkingSetSize;
    SIZE_T MaximumWorkingSetSize;
    DWORD ActiveProcessLimit;
    ULONG_PTR Affinity;
    DWORD PriorityClass;
    DWORD SchedulingClass;
} JOBOBJECT_BASIC_LIMIT_INFORMATION, *PJOBOBJECT_BASIC_LIMIT_INFORMATION;
```

Members

PerProcessUserTimeLimit

Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_PROCESS_TIME. Specifies the per-process user-mode execution time limit, in 100-nanosecond ticks.

The system periodically checks to determine whether each process associated with the job has accumulated more user-mode time than the set limit. If it has, the process is terminated.

PerJobUserTimeLimit

Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_JOB_TIME. Specifies the per-job user-mode execution time limit, in 100-nanosecond ticks. The system adds the current time of the processes associated with the job to this limit. For example, if you set this limit to 1 minute, and the job has a process that has accumulated 5 minutes of user-mode time, the limit actually enforced is 6 minutes.

The system periodically checks to determine whether the sum of the user-mode execution time for all processes is greater than this end-of-job limit. If it is, the action specified in the **EndOfJobTimeAction** member of the **JOBOBJECT_END_OF_JOB_TIME_INFORMATION** structure is carried out. By default, all processes are terminated and the status code is set to ERROR_NOT_ENOUGH_QUOTA.

LimitFlags

Specifies the limit flags that are in effect. This member is a bit field that determines whether other structure members are used. Any combination of the following values can be specified.

Value	Meaning
JOB_OBJECT_LIMIT_ACTIVE_PROCESS	Establishes a maximum number of simultaneously active processes associated with the job.

(continued)

(continued)

Value	Meaning
JOB_OBJECT_LIMIT_AFFINITY	Causes all processes associated with the job to use the same processor affinity.
JOB_OBJECT_LIMIT_BREAKAWAY_OK	If any process associated with the job creates a child process using the CREATE_BREAKAWAY_FROM_JOB flag while this limit is in effect, the child process is not associated with the job. This limit requires use of a **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** structure. The **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.
JOB_OBJECT_LIMIT_DIE_ON_UNHANDLED_EXCEPTION	Forces a call to the **SetErrorMode** function with the SEM_NOGPFAULTERRORBOX flag for each process associated with the job. If an exception occurs and the system calls the **UnhandledExceptionFilter** function, the debugger will be given a chance to act. If there is no debugger, the functions returns EXCEPTION_EXECUTE_HANDLER. Normally, this will cause termination of the process with the exception code as the exit status.
JOB_OBJECT_LIMIT_JOB_MEMORY	Causes all processes associated with the job to limit the job-wide sum of their committed memory. When a process attempts to commit memory that would exceed the job-wide limit, it fails. If the job object is associated with a completion port, a JOB_OBJECT_MSG_JOB_MEMORY_LIMIT message is sent to the completion port.

Value	Meaning
JOB_OBJECT_LIMIT_JOB_TIME	Establishes a user-mode execution time limit for the job. This flag cannot be used with JOB_OBJECT_LIMIT _PRESERVE_JOB_TIME.
JOB_OBJECT_LIMIT_PRIORITY_CLASS	Causes all processes associated with the job to use the same priority class. For more information, see *Scheduling Priorities*.
JOB_OBJECT_LIMIT_PRESERVE_JOB_TIME	Preserves any job time limits you previously set. As long as this flag is set, you can establish a per-job time limit once, then alter other limits in subsequent calls. This flag cannot be used with JOB_OBJECT_ LIMIT_JOB_TIME.
JOB_OBJECT_LIMIT_PROCESS_MEMORY	Causes all processes associated with the job to limit their committed memory. When a process attempts to commit memory that would exceed the per-process limit, it fails. If the job object is associated with a completion port, a JOB_OBJECT_MSG_PROCESS_M EMORY_LIMIT message is sent to the completion port.
JOB_OBJECT_LIMIT_PROCESS_TIME	Establishes a user-mode execution time limit for each currently active process and for all future processes associated with the job.
JOB_OBJECT_LIMIT_SCHEDULING_CLASS	Causes all processes in the job to use the same scheduling class.
JOB_OBJECT_LIMIT_WORKINGSET	Causes all processes associated with the job to use the same minimum and maximum working set sizes.
JOB_OBJECT_LIMIT_SILENT_BREAKAWAY_OK	Allows any process associated with the job to create child processes that are not associated with the job.

(continued)

(continued)

Value	Meaning
	This limit requires use of a **JOBOBJECT_EXTENDED _LIMIT_INFORMATION** structure. The **BasicLimitInformation** member is a **JOBOBJECT_BASIC_LIMIT _INFORMATION** structure.

MinimumWorkingSetSize
Ignored unless the **LimitFlags** member specifies JOB_OBJECT_LIMIT_WORKINGSET. Specifies the minimum working set size for all processes associated with the job.

MaximumWorkingSetSize
Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_WORKINGSET. Specifies the maximum working set size for all processes associated with the job.

ActiveProcessLimit
Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_ACTIVE_PROCESS. Specifies the active process limit for the job. If you try to associate a process with a job, and this causes the active process count to exceed this limit, the process is terminated and the association fails.

Affinity
Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_AFFINITY. Specifies the processor affinity for all processes associated with the job. The affinity must be a proper subset of the system affinity mask obtained by calling the **GetProcessAffinityMask** function. The affinity of each thread is set to this value, but threads are free to subsequently set their affinity, as long as it is a subset of the specified affinity mask. Processes cannot set their own affinity mask.

PriorityClass
Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_PRIORITY_CLASS. Specifies the priority class for all processes associated with the job. Processes and threads cannot modify their priority class. The calling process must enable the SE_INC_BASE_PRIORITY_NAME privilege.

SchedulingClass
Ignored unless **LimitFlags** specifies JOB_OBJECT_LIMIT_SCHEDULING_CLASS. Specifies the scheduling class for all processes associated with the job.

The valid values are 0 to 9. Use 0 for the least favorable scheduling class relative to other threads, and 9 for the most favorable scheduling class relative to other threads. By default, this value is 5. To use a scheduling class greater than 5, the calling process must enable the SE_INC_BASE_PRIORITY_NAME privilege.

Remarks

Processes can still empty their working sets using the **SetProcessWorkingSetSize** function, even when JOB_OBJECT_LIMIT_WORKINGSET is used. However, you

cannot use **SetProcessWorkingSetSize** to change the minimum or maximum working set size.

The system increments the active process count when you attempt to associate a process with a job. If the limit is exceeded, the system decrements the active process count only when the process terminates and all handles to the process are closed. Therefore, if you have an open handle to a process that has been terminated in such a manner, you cannot associate any new processes until the handle is closed and the active process count is below the limit.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

+ See Also

Processes and Threads Overview, *Process and Thread Structures*, **GetProcessAffinityMask**, **JOBOBJECT_END_OF_JOB_TIME_INFORMATION**, **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** **QueryInformationJobObject**, **SetInformationJobObject**, **SetProcessWorkingSetSize**

JOBOBJECT_BASIC_PROCESS_ID_LIST

The **JOBOBJECT_BASIC_PROCESS_ID_LIST** structure contains the process identifier list for a job object.

```
typedef struct _JOBOBJECT_BASIC_PROCESS_ID_LIST {
    DWORD NumberOfAssignedProcesses;
    DWORD NumberOfProcessIdsInList;
    ULONG_PTR ProcessIdList[1];
} JOBOBJECT_BASIC_PROCESS_ID_LIST, *PJOBOBJECT_BASIC_PROCESS_ID_LIST;
```

Members

NumberOfAssignedProcesses
Specifies the number of process identifiers to be stored in **ProcessIdList**.

NumberOfProcessIdsInList
Specifies the number of process identifiers returned in the **ProcessIdList** buffer. If this number is less than **NumberOfAssignedProcesses**, increase the size of the buffer to accommodate the complete list.

ProcessIdList
Specifies the variable-length array of process identifiers returned by this call. Array elements 0 through **NumberOfProcessIdsInList**–1 contain valid process identifiers.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **QueryInformationJobObject**, **SetInformationJobObject**

JOBOBJECT_BASIC_UI_RESTRICTIONS

The **JOBOBJECT_BASIC_UI_RESTRICTIONS** structure contains basic user-interface restrictions for a job object.

```
typedef struct _JOBOBJECT_BASIC_UI_RESTRICTIONS {
    DWORD UIRestrictionsClass;
} JOBOBJECT_BASIC_UI_RESTRICTIONS, *PJOBOBJECT_BASIC_UI_RESTRICTIONS;
```

Members

UIRestrictionsClass

Specifies the restriction class for the user interface. This member can be one or more of the following values:

Value	Meaning
JOB_OBJECT_UILIMIT_DESKTOP	Prevents processes associated with the job from creating desktops and switching desktops using the **CreateDesktop** and **SwitchDesktop** functions.
JOB_OBJECT_UILIMIT_DISPLAYSETTINGS	Prevents processes associated with the job from calling the **ChangeDisplaySettings** function.
JOB_OBJECT_UILIMIT_EXITWINDOWS	Prevents processes associated with the job from calling the **ExitWindows** or **ExitWindowsEx** function.
JOB_OBJECT_UILIMIT_GLOBALATOMS	Prevents processes associated with the job from accessing global atoms. When this flag is used, each job has its own atom table.
JOB_OBJECT_UILIMIT_HANDLES	Prevents processes associated with the job from using USER handles owned by processes not associated with the same job.

JOB_OBJECT_UILIMIT_READCLIPBOARD	Prevents processes associated with the job from reading data from the clipboard.
JOB_OBJECT_UILIMIT_SYSTEMPARAMETERS	Prevents processes associated with the job from changing system parameters by using the **SystemParametersInfo** function.
JOB_OBJECT_UILIMIT_WRITECLIPBOARD	Prevents processes associated with the job from writing data to the clipboard.

Remarks

If you specify the JOB_OBJECT_UILIMIT_HANDLES flag, when a process associated with the job broadcasts messages, they are only sent to top-level windows owned by processes associated with the same job. In addition, hooks can be installed only on threads belonging to processes associated with the job.

To grant access to a User handle to a job that has a user-interface restriction, use the **UserHandleGrantAccess** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **ExitWindows**, **ExitWindowsEx** **QueryInformationJobObject**, **SetInformationJobObject**, **SystemParametersInfo**, **UserHandleGrantAccess**

JOBOBJECT_END_OF_JOB_TIME_INFORMATION

The **JOBOBJECT_END_OF_JOB_TIME_INFORMATION** structure specifies the action the system will perform when an end-of-job time limit is exceeded.

```
typedef struct _JOBOBJECT_END_OF_JOB_TIME_INFORMATION {
    DWORD EndOfJobTimeAction;
} JOBOBJECT_END_OF_JOB_TIME_INFORMATION, PJOBOBJECT_END_OF_JOB_TIME_INFORMATION;
```

Members

EndOfJobTimeAction

Specifies the action that the system will perform when the end-of-job time limit has been exceeded. This member can be one of the following values:

Value	Meaning
JOB_OBJECT_TERMINATE_AT_END_OF_JOB	Terminates all processes and sets the exit status to ERROR_NOT_ENOUGH_QUOTA. The processes cannot prevent or delay their own termination. The job object is set to the signaled state and remains signaled until this limit is reset. No additional processes can be assigned to the job until the limit is reset. This is the default termination action.
JOB_OBJECT_POST_AT_END_OF_JOB	Posts a completion packet to the completion port using the **PostQueuedCompletionStatus** function. After the completion packet is posted, the system clears the end-of-job time limit, and processes in the job can continue their execution. If no completion port is associated with the job when the time limit has been exceeded, the action taken is the same as for JOB_OBJECT_TERMINATE_AT_END_OF_JOB.

Remarks

The end-of-job time limit is specified in the **PerJobUserTimeLimit** member of the **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure.

To associate a completion port with a job, use the **JOBOBJECT_ASSOCIATE_COMPLETION_PORT** structure.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **JOBOBJECT_ASSOCIATE_COMPLETION_PORT**, **JOBOBJECT_BASIC_LIMIT_INFORMATION**, **PostQueuedCompletionStatus**, **QueryInformationJobObject**, **SetInformationJobObject**

JOBOBJECT_EXTENDED_LIMIT_INFORMATION

The **JOBOBJECT_EXTENDED_LIMIT_INFORMATION** structure contains basic and extended limit information for a job object.

```
typedef struct _JOBOBJECT_EXTENDED_LIMIT_INFORMATION {
    JOBOBJECT_BASIC_LIMIT_INFORMATION BasicLimitInformation;
    IO_COUNTERS IoInfo;
    SIZE_T ProcessMemoryLimit;
    SIZE_T JobMemoryLimit;
    SIZE_T PeakProcessMemoryUsed;
    SIZE_T PeakJobMemoryUsed;
} JOBOBJECT_EXTENDED_LIMIT_INFORMATION, *PJOBOBJECT_EXTENDED_LIMIT_INFORMATION;
```

Members

BasicLimitInformation

Pointer to a **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure that contains basic limit information.

IoInfo

Reserved.

ProcessMemoryLimit

Ignored unless the **LimitFlags** member of the **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure specifies the JOB_OBJECT_LIMIT_PROCESS_MEMORY value. Specifies the per-process memory limit.

JobMemoryLimit

Ignored unless the **LimitFlags** member of the **JOBOBJECT_BASIC_LIMIT_INFORMATION** structure specifies the JOB_OBJECT_LIMIT_JOB_MEMORY value. Specifies the per-job memory limit.

PeakProcessMemoryUsed

Specifies the most process memory used by any process ever associated with the job.

PeakJobMemoryUsed

Specifies the peak memory usage of all processes associated with the job.

Remarks

The system tracks the value of **PeakProcessMemoryUsed** and **PeakJobMemoryUsed** constantly. This allows you know the peak memory usage of each job. You can use this information to establish a memory limit using the JOB_OBJECT_LIMIT_PROCESS_MEMORY or JOB_OBJECT_LIMIT_JOB_MEMORY value.

Note that the job memory and process memory limits are very similar in operation, but they are independent. You could set a job-wide limit of 100 MB with a per-process limit

of 10 MB. In this scenario, no single process could commit more than 10 MB, and the set of processes associated with a job could never exceed 100 MB.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **JOBOBJECT_BASIC_LIMIT_INFORMATION**, **QueryInformationJobObject**, **SetInformationJobObject**

JOBOBJECT_SECURITY_LIMIT_INFORMATION

The **JOBOBJECT_SECURITY_LIMIT_INFORMATION** structure contains the security limitations for a job object.

```
typedef struct _JOBOBJECT_SECURITY_LIMIT_INFORMATION {
    DWORD SecurityLimitFlags ;
    HANDLE JobToken ;
    PTOKEN_GROUPS SidsToDisable ;
    PTOKEN_PRIVILEGES PrivilegesToDelete ;
    PTOKEN_GROUPS RestrictedSids ;
} JOBOBJECT_SECURITY_LIMIT_INFORMATION, *PJOBOBJECT_SECURITY_LIMIT_INFORMATION ;
```

Members

SecurityLimitFlags

Specifies the security limitations for the job. This member can be one or more of the following values.

Value	Meaning
JOB_OBJECT_SECURITY_NO_ADMIN	Prevents any process in the job from using a token that specifies the local administrators group.
JOB_OBJECT_SECURITY_RESTRICTED_TOKEN	Prevents any process in the job from using a token that was not created with the **CreateRestrictedToken** function.
JOB_OBJECT_SECURITY_ONLY_TOKEN	Forces processes in the job to run under a specific token. Requires a token handle in the **JobToken** member.

JOB_OBJECT_SECURITY_FILTER_TOKENS	Applies a filter to the token when a process impersonates a client. Requires at least one of the following members to be set: **SidsToDisable**, **PrivilegesToDelete**, or **RestrictedSids**.

JobToken

Handle to a primary token that represents a user. The handle must have TOKEN_ASSIGN_PRIMARY access.

If the token was created with **CreateRestrictedToken**, all processes in the job are limited to that token or a further restricted token. Otherwise, the caller must have the SE_ASSIGNPRIMARYTOKEN_NAME privilege.

SidsToDisable

Pointer to a **TOKEN_GROUPS** structure that specifies the SIDs to disable for access checking, if **SecurityLimitFlags** is JOB_OBJECT_SECURITY_FILTER_TOKENS.

This member can be NULL if you do not want to disable any SIDs.

PrivilegesToDelete

Pointer to a **TOKEN_PRIVILEGES** structure that specifies the privileges to delete from the token, if **SecurityLimitFlags** is JOB_OBJECT_SECURITY_FILTER_TOKENS.

This member can be NULL if you do not want to delete any privileges.

RestrictedSids

Pointer to a **TOKEN_GROUPS** structure that specifies the deny-only SIDs that will be added to the access token, if **SecurityLimitFlags** is JOB_OBJECT_SECURITY_FILTER_TOKENS.

This member can be NULL if you do not want to specify any deny-only SIDs.

Remarks

After security limitations are placed on processes in a job, they cannot be revoked.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

+ See Also

Processes and Threads Overview, *Process and Thread Structures*, **CreateRestrictedToken**, **QueryInformationJobObject**, **SetInformationJobObject**, **TOKEN_GROUPS**, **TOKEN_PRIVILEGES**

PROCESS_INFORMATION

The **PROCESS_INFORMATION** structure is filled in by the **CreateProcess** function with information about a newly created process and its primary thread.

```
typedef struct _PROCESS_INFORMATION {
    HANDLE hProcess;
    HANDLE hThread;
    DWORD dwProcessId;
    DWORD dwThreadId;
} PROCESS_INFORMATION;
```

Members

hProcess
Returns a handle to the newly created process. The handle is used to specify the process in all functions that perform operations on the process object.

hThread
Returns a handle to the primary thread of the newly created process. The handle is used to specify the thread in all functions that perform operations on the thread object.

dwProcessId
Returns a global process identifier that can be used to identify a process. The value is valid from the time the process is created until the time the process is terminated.

dwThreadId
Returns a global thread identifiers that can be used to identify a thread. The value is valid from the time the thread is created until the time the thread is terminated.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.

Processes and Threads Overview, *Process and Thread Structures*, **CreateProcess**

STARTUPINFO

The **STARTUPINFO** structure is used with the **CreateProcess** function to specify main window properties if a new window is created for the new process. For graphical user interface (GUI) processes, this information affects the first window created by the **CreateWindow** function and shown by the **ShowWindow** function. For console processes, this information affects the console window if a new console is created for the

process. A process can use the **GetStartupInfo** function to retrieve the **STARTUPINFO** structure specified when the process was created.

```
typedef struct _STARTUPINFO {
    DWORD   cb;
    LPTSTR  lpReserved;
    LPTSTR  lpDesktop;
    LPTSTR  lpTitle;
    DWORD   dwX;
    DWORD   dwY;
    DWORD   dwXSize;
    DWORD   dwYSize;
    DWORD   dwXCountChars;
    DWORD   dwYCountChars;
    DWORD   dwFillAttribute;
    DWORD   dwFlags;
    WORD    wShowWindow;
    WORD    cbReserved2;
    LPBYTE  lpReserved2;
    HANDLE  hStdInput;
    HANDLE  hStdOutput;
    HANDLE  hStdError;
} STARTUPINFO, *LPSTARTUPINFO;
```

Members

cb

Specifies the size, in bytes, of the structure.

lpReserved

Reserved. Set this member to NULL before passing the structure to **CreateProcess**.

lpDesktop

Windows NT/2000: Pointer to a null-terminated string that specifies either the name of the desktop only or the name of both the desktop and window station for this process. A backslash in the string pointed to by **lpDesktop** indicates that the string includes both desktop and window station names. If **lpDesktop** is NULL, the new process inherits the desktop and window station of its parent process. If **lpDesktop** is an empty string, the process does not inherit the desktop and window station of its parent process; instead, the system determines if a new desktop and window station need to be created. If the impersonated user already has a desktop, the system will use the existing desktop.

lpTitle

For console processes, this is the title displayed in the title bar if a new console window is created. If NULL, the name of the executable file is used as the window title instead. This parameter must be NULL for GUI or console processes that do not create a new console window.

dwX
Ignored unless **dwFlags** specifies STARTF_USEPOSITION. Specifies the x offset, in pixels, of the upper left corner of a window if a new window is created. The offset is from the upper left corner of the screen. For GUI processes, the specified position is used the first time the new process calls **CreateWindow** to create an overlapped window if the *x* parameter of **CreateWindow** is CW_USEDEFAULT.

dwY
Ignored unless **dwFlags** specifies STARTF_USEPOSITION. Specifies the y offset, in pixels, of the upper left corner of a window if a new window is created. The offset is from the upper left corner of the screen. For GUI processes, the specified position is used the first time the new process calls **CreateWindow** to create an overlapped window if the *y* parameter of **CreateWindow** is CW_USEDEFAULT.

dwXSize
Ignored unless **dwFlags** specifies STARTF_USESIZE. Specifies the width, in pixels, of the window if a new window is created. For GUI processes, this is used only the first time the new process calls **CreateWindow** to create an overlapped window if the *nWidth* parameter of **CreateWindow** is CW_USEDEFAULT.

dwYSize
Ignored unless **dwFlags** specifies STARTF_USESIZE. Specifies the height, in pixels, of the window if a new window is created. For GUI processes, this is used only the first time the new process calls **CreateWindow** to create an overlapped window if the *nHeight* parameter of **CreateWindow** is CW_USEDEFAULT.

dwXCountChars
Ignored unless **dwFlags** specifies STARTF_USECOUNTCHARS. For console processes, if a new console window is created, **dwXCountChars** specifies the screen buffer width in character columns. This value is ignored in a GUI process.

dwYCountChars
Ignored unless **dwFlags** specifies STARTF_USECOUNTCHARS. For console processes, if a new console window is created, **dwYCountChars** specifies the screen buffer height in character rows. This value is ignored in a GUI process.

dwFillAttribute
Ignored unless **dwFlags** specifies STARTF_USEFILLATTRIBUTE. Specifies the initial text and background colors if a new console window is created in a console application. These values are ignored in GUI applications. This value can be any combination of the following values: FOREGROUND_BLUE, FOREGROUND_GREEN, FOREGROUND_RED, FOREGROUND_INTENSITY, BACKGROUND_BLUE, BACKGROUND_GREEN, BACKGROUND_RED, and BACKGROUND_INTENSITY. For example, the following combination of values produces red text on a white background:

```
FOREGROUND_RED| BACKGROUND_RED| BACKGROUND_GREEN| BACKGROUND_BLUE
```

dwFlags

This is a bit field that determines whether certain **STARTUPINFO** members are used when the process creates a window. Any combination of the following values can be specified:

Value	Meaning
STARTF_FORCEONFEEDBACK	Indicates that the cursor is in feedback mode for two seconds after **CreateProcess** is called. If during those two seconds the process makes the first GUI call, the system gives five more seconds to the process. If during those five seconds the process shows a window, the system gives five more seconds to the process to finish drawing the window. The system turns the feedback cursor off after the first call to **GetMessage**, regardless of whether the process is drawing.
STARTF_FORCEOFFFEEDBACK	Indicates that the feedback cursor is forced off while the process is starting. The normal cursor is displayed.
STARTF_RUNFULLSCREEN	Indicates that the process should be run in full-screen mode, rather than in windowed mode. This flag is only valid for console applications running on an x86 computer.
STARTF_USECOUNTCHARS	If this value is not specified, the **dwXCountChars** and **dwYCountChars** members are ignored.
STARTF_USEFILLATTRIBUTE	If this value is not specified, the **dwFillAttribute** member is ignored.
STARTF_USEPOSITION	If this value is not specified, the **dwX** and **dwY** members are ignored.
STARTF_USESHOWWINDOW	If this value is not specified, the **wShowWindow** member is ignored.
STARTF_USESIZE	If this value is not specified, the **dwXSize** and **dwYSize** members are ignored.
STARTF_USESTDHANDLES	Sets the standard input, standard output, and standard error handles for the process to the handles specified in the **hStdInput**, **hStdOutput**, and **hStdError** members of the **STARTUPINFO** structure. The **CreateProcess** function's *fInheritHandles* parameter must be set to TRUE for this to work properly. If this value is not specified, the **hStdInput**, **hStdOutput**, and **hStdError** members of the **STARTUPINFO** structure are ignored.

wShowWindow

Ignored unless **dwFlags** specifies STARTF_USESHOWWINDOW. The **wShowWindow** member can be any of the SW_ constants defined in WINUSER.H. For GUI processes, **wShowWindow** specifies the default value the first time **ShowWindow** is called. The *nCmdShow* parameter of **ShowWindow** is ignored. In subsequent calls to **ShowWindow**, the **wShowWindow** member is used if the *nCmdShow* parameter of **ShowWindow** is set to SW_SHOWDEFAULT.

cbReserved2

Reserved; must be zero.

lpReserved2

Reserved; must be NULL.

hStdInput

Ignored unless **dwFlags** specifies STARTF_USESTDHANDLES. Specifies a handle that will be used as the standard input handle to the process if STARTF_USESTDHANDLES is specified.

hStdOutput

Ignored unless **dwFlags** specifies STARTF_USESTDHANDLES. Specifies a handle that will be used as the standard output handle to the process if STARTF_USESTDHANDLES is specified.

hStdError

Ignored unless **dwFlags** specifies STARTF_USESTDHANDLES. Specifies a handle that will be used as the standard error handle to the process if STARTF_USESTDHANDLES is specified.

Remarks

If a GUI process is being started and neither STARTF_FORCEONFEEDBACK or STARTF_FORCEOFFFEEDBACK is specified, the process feedback cursor is used. A GUI process is one whose subsystem is specified as "windows."

Requirements

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.
Unicode: Declared as Unicode and ANSI structures.

See Also

Processes and Threads Overview, *Process and Thread Structures*, **CreateProcess**, **CreateProcessAsUser**, **CreateWindow**, **GetMessage**, **GetStartupInfo**, **PeekMessage**, **ShowWindow**, **WinMain**

Process and Thread Macros

GetCurrentFiber

The **GetCurrentFiber** macro returns the address of the current fiber.

```
PVOID GetCurrentFiber(VOID);
```

Parameters

This macro has no parameters.

Return Values

The return value is the address of the currently running fiber.

Remarks

The **CreateFiber** and **ConvertThreadToFiber** functions return the fiber address when the fiber is created. The **GetCurrentFiber** macro allows you to retrieve the address at any other time.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

Processes and Threads Overview, *Process and Thread Macros*, **CreateFiber**, **ConvertThreadToFiber**

GetFiberData

The **GetFiberData** macro returns the fiber data associated with the current fiber.

```
PVOID GetFiberData(VOID);
```

Parameters

This macro has no parameters.

Return Values

The return value is the fiber data for the currently running fiber.

Remarks

The fiber data is the value passed to the **CreateFiber** or **ConvertThreadToFiber** functions in the *lpParameter* parameter. This value is also received as the parameter to the fiber function. It is stored as part of the fiber state information.

! Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

+ See Also

Processes and Threads Overview, *Process and Thread Macros*, **CreateFiber**, **ConvertThreadToFiber**

Dynamic Link Libraries

Dynamic link libraries (DLL) are modules that contain functions and data. A DLL is loaded at run time by its calling modules (.exe or .dll). When a DLL is loaded, it is mapped into the address space of the calling process.

DLLs can define two kinds of functions: exported and internal. The exported functions can be called by other modules. Internal functions can only be called from within the DLL where they are defined. Although DLLs can export data, its data is usually only used by its functions.

DLLs provide a way to modularize applications so that functionality can be updated and reused more easily. They also help reduce memory overhead when several applications use the same functionality at the same time, because although each application gets its own copy of the data, they can share the code.

The Microsoft Win32 application programming interface (API) is implemented as a set of dynamic link libraries, so any process that uses the Win32 API uses dynamic linking.

About Dynamic Link Libraries

Dynamic linking allows a module to include only the information the system needs at load time or run time to locate the code for an exported DLL function. Dynamic linking differs from the more familiar static linking, in which the linker copies a library function's code into each module that calls it.

Types of Dynamic Linking

There are two methods for calling a function in a DLL:

- In *load-time dynamic linking*, a module makes explicit calls to exported DLL functions. This requires you to link the module with the import library for the DLL. An import

library supplies the system with the information needed to load the DLL and locate the exported DLL functions when the application is loaded. For more information, see *Load-Time Dynamic Linking*.

- In *run-time dynamic linking*, a module uses the **LoadLibrary** or **LoadLibraryEx** function to load the DLL at run time. After the DLL is loaded, the module calls the **GetProcAddress** function to get the addresses of the exported DLL functions. The module calls the exported DLL functions using the function pointers returned by **GetProcAddress**. This eliminates the need for an import library. For more information, see *Using Run-Time Dynamic Linking*.

DLLs and Memory Management

Every process that loads the DLL maps it into its virtual address space. After the process loads the DLL into its virtual address, it can call the exported DLL functions.

The system maintains a reference count for each DLL. When a thread loads the DLL, its reference count is incremented by one. When the process terminates, or when the reference count goes to 0 (run-time dynamic linking only), the DLL is unloaded from the virtual address space.

Like any other function, an exported DLL function runs in the context of the thread that calls it. Therefore, the following conditions apply:

- The threads of the process that called the DLL can use handles opened by a DLL function. Similarly, handles opened by any thread of the calling process can be used in the DLL function.
- The DLL uses the stack of the calling thread and the virtual address space of the calling process.
- The DLL allocates memory from the virtual address space of the calling process.

Advantages of Dynamic Linking

Dynamic linking has the following advantages over static linking:

- Processes that load a DLL at the same base address can use a single DLL simultaneously, sharing a single copy of the DLL code in physical memory. Doing this saves memory and reduces swapping.
- When the functions in a DLL change, the applications that use them do not need to be recompiled or relinked as long as the function arguments, calling conventions, and return values do not change. In contrast, statically linked object code requires that the application be relinked when the functions change.
- A DLL can provide after-market support. For example, a display driver DLL can be modified to support a display that was not available when the application was initially shipped.
- Programs written in different programming languages can call the same DLL function as long as the programs follow the same calling convention that the function uses. The calling convention (such as C, Pascal, or standard call) controls the order in

which the calling function must push the arguments onto the stack, whether the function or the calling function is responsible for cleaning up the stack, and whether any arguments are passed in registers. For more information, see the documentation included with your compiler.

A potential disadvantage to using DLLs is that the application is not self-contained; it depends on the existence of a separate DLL module. The system terminates processes using load-time dynamic linking if they require a DLL that is not found at process startup and gives an error message to the user. The system does not terminate a process using run-time dynamic linking in this situation, but functions exported by the DLL are not available to the program.

Dynamic Link Library Creation

To create a DLL, you must create one or more source code files, and possibly a linker file for exporting the functions. If you plan to allow applications that use your DLL to use load-time dynamic linking, you must also create an import library.

Creating Source Files

The source files contain exported functions, internal functions, and an optional entry-point function for the DLL. You may use any development tools that support the creation of Win32-based DLLs.

If your DLL may be used by a multithreaded application, you should make your DLL "thread-safe" by linking only with libraries that have support for multiple threads. Also, be sure to synchronize access to your global data.

Exporting Functions

How you specify exported functions depends on the tools that you are using for development. Some compilers allow you to export a function directly in the source code by using a modifier in the function declaration. Other times, you must specify exports in a file that you pass to the linker.

For example, using Microsoft Visual C++, there are two possible ways to export DLL functions: with _declspec modifier or with a .DEF file. If you use the _declspec modifier, it is not necessary to use a .DEF file.

For more information about exporting functions, see the documentation included with your development tools.

Creating an Import Library

The import library (.LIB) file contains information the linker needs to resolve external references to exported DLL functions, so the system can locate the specified DLL and exported DLL functions at run time. For example, to call the **CreateWindow** function, you must link your code with the import library USER32.LIB. The reason is that **CreateWindow** resides in a system DLL. The file USER32.LIB is the import library used to resolve the call to **CreateWindow** in your code.

For information about creating import libraries, see the documentation included with your development tools.

Dynamic Link Library Entry-Point Function

Every DLL must have an entry point, just as an application does. The system calls the entry-point function whenever processes and threads load or unload the DLL. If you are linking your DLL with a library, such as the C run-time library, it may provide an entry-point function for you, and allow you to provide a separate initialization function. Check the documentation for your run-time library for more information.

If you are providing your own entry-point, see the **DllMain** function. The name **DllMain** is a placeholder for a user-defined function. Earlier versions of the SDK documentation used **DllEntryPoint** as the entry-point function name. You must specify the actual name you use when you build your DLL. For more information, see the documentation included with your development tools.

Calling the Entry-Point Function

The system calls the entry-point function whenever any one of the following events occurs:

- A process loads the DLL. For processes using load-time dynamic linking, the DLL is loaded during process initialization. For processes using run-time linking, the DLL is loaded before **LoadLibrary** or **LoadLibraryEx** returns.
- A process unloads the DLL. The DLL is unloaded when the process terminates or calls the **FreeLibrary** function and the reference count becomes zero. If the process terminates as a result of the **TerminateProcess** or **TerminateThread** function, the system does not call the DLL entry-point function.
- A new thread is created in a process that has loaded the DLL. You can use the **DisableThreadLibraryCalls** function to disable notification when threads are created.
- A thread of a process that has loaded the DLL terminates normally, not using **TerminateThread** or **TerminateProcess**. When a process unloads the DLL, the entry-point function is called only once for the entire process, rather than once for each existing thread of the process. You can use **DisableThreadLibraryCalls** to disable notification when threads are terminated.

Only one thread at a time can call the entry-point function.

The system calls the entry-point function in the context of the process or thread that caused the function to be called. This allows a DLL to use its entry-point function for allocating memory in the virtual address space of the calling process or to open handles accessible to the process. The entry-point function can also allocate memory that is private to a new thread by using thread local storage (TLS). For more information about thread local storage, see *Thread Local Storage*.

Entry-Point Function Definition

The DLL entry-point function must be declared with the standard-call calling convention.

> **Windows NT/2000:** If the DLL entry point is not declared correctly, the DLL is not loaded, and the system displays a message indicating that the DLL entry point must be declared with WINAPI.
>
> **Windows 95:** If the DLL entry point is not declared correctly, the DLL is not loaded and the system displays a message titled "Error starting program," which instructs the user to check the file to determine the problem.

In the body of the function, you may handle any combination of the following scenarios in which the DLL entry point has been called:

- A process loads the DLL (DLL_PROCESS_ATTACH).
- The current process creates a new thread (DLL_THREAD_ATTACH).
- A thread exits normally (DLL_THREAD_DETACH).
- A process unloads the DLL (DLL_PROCESS_DETACH).

Your function should perform only simple initialization tasks, such as setting up thread local storage (TLS), creating synchronization objects, and opening files. It must not call the **LoadLibrary** function, because this may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, you must not call the **FreeLibrary** function in the entry-point function, because this can result in a DLL being used after the system has executed its termination code.

Calling Win32 functions other than TLS, synchronization, and file functions may also result in problems that are difficult to diagnose. For example, calling User, Shell, and COM functions can cause access violation errors, because some functions in their DLLs call **LoadLibrary** to load other system components.

The following example demonstrates how to structure the DLL entry-point function.

```
BOOL WINAPI DllMain(
    HINSTANCE hinstDLL,  // handle to DLL module
    DWORD fdwReason,     // reason for calling function
    LPVOID lpReserved )  // reserved
{
    // Perform actions based on the reason for calling.
    switch( fdwReason )
    {
        case DLL_PROCESS_ATTACH:
         // Initialize once for each new process.
         // Return FALSE to fail DLL load.
            break;

        case DLL_THREAD_ATTACH:
```

```
        // Do thread-specific initialization.
           break;

        case DLL_THREAD_DETACH:
        // Do thread-specific cleanup.
           break;

        case DLL_PROCESS_DETACH:
        // Perform any necessary cleanup.
           break;
    }
    return TRUE;  // Successful DLL_PROCESS_ATTACH.
}
```

Entry-Point Function Return Value

When a DLL entry-point function is called because a process is loading, the function returns TRUE to indicate success. For processes using load-time linking, a return value of FALSE causes the process initialization to fail and the process terminates. For processes using run-time linking, a return value of FALSE causes the **LoadLibrary** or **LoadLibraryEx** function to return NULL, indicating failure. (The system immediately calls your entry-point function with DLL_PROCESS_DETACH and unloads the DLL.) The return value of the entry-point function is disregarded when the function is called for any other reason.

Load-Time Dynamic Linking

When the system starts a program that uses load-time dynamic linking, it uses the information in the file to locate the names of the required DLL(s). The system then searches for the DLLs in the following locations, in sequence:

1. The directory from which the application loaded.
2. The current directory.
3. **Windows 95/98:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to obtain the path of this directory.
4. **Windows NT/2000:** The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

If the system cannot locate a specified DLL, it terminates the process and displays a dialog box that reports the error. Otherwise, the system maps the DLL modules into the virtual address space of the process and increments the DLL reference count.

The system calls the entry-point function. The function receives a code indicating that the process is loading the DLL. If the entry-point function does not return TRUE, the system terminates the process and reports the error. For more information about the entry-point function, see *Dynamic Link Library Entry-Point Function*.

Finally, the system modifies the code of the process to provide starting addresses for the referenced DLL functions.

The DLL is mapped into the virtual address space of the process during its initialization and is loaded into physical memory only when needed.

Run-Time Dynamic Linking

When the application calls the **LoadLibrary** or **LoadLibraryEx** functions, the system attempts to locate the DLL using the same search sequence used in load-time dynamic linking (see *Load-Time Dynamic Linking*). If the search succeeds, the system maps the DLL module into the virtual address space of the process and increments the reference count. If the call to **LoadLibrary** or **LoadLibraryEx** specifies a DLL whose code is already mapped into the virtual address space of the calling process, the function simply returns a handle to the DLL and increments the DLL reference count. Note that two DLLs that have the same base file name and extension but are found in different directories are not considered to be the same DLL.

The system calls the entry-point function in the context of the thread that called **LoadLibrary** or **LoadLibraryEx**. The entry-point function is not called if the DLL was already loaded by the process through a call to **LoadLibrary** or **LoadLibraryEx** with no corresponding call to the **FreeLibrary** function.

If the system cannot find the DLL or if the entry-point function returns FALSE, **LoadLibrary** or **LoadLibraryEx** returns NULL. If **LoadLibrary** or **LoadLibraryEx** succeeds, it returns a handle to the DLL module. The process can use this handle to identify the DLL in a call to the **GetProcAddress**, **FreeLibrary**, or **FreeLibraryAndExitThread** function.

The **GetModuleHandle** function returns a handle used in **GetProcAddress**, **FreeLibrary**, or **FreeLibraryAndExitThread**. The **GetModuleHandle** function succeeds only if the DLL module is already mapped into the address space of the process by load-time linking or by a previous call to **LoadLibrary** or **LoadLibraryEx**. Unlike **LoadLibrary** or **LoadLibraryEx**, **GetModuleHandle** does not increment the module reference count. The **GetModuleFileName** function retrieves the full path of the module associated with a handle returned by **GetModuleHandle**, **LoadLibrary**, or **LoadLibraryEx**.

The process can use **GetProcAddress** to get the address of an exported function in the DLL using a DLL module handle returned by either **LoadLibrary, LoadLibraryEx,** or **GetModuleHandle**.

When the DLL module is no longer needed, the process can call **FreeLibrary** or **FreeLibraryAndExitThread**. These functions decrement the module reference count and unmap the DLL code from the virtual address space of the process if the reference count is zero.

Run-time dynamic linking enables the process to continue running even if a DLL is not available. The process can then use an alternate method to accomplish its objective. For example, if a process is unable to locate one DLL, it can try to use another, or it can notify the user of an error. If the user can provide the full path of the missing DLL, the process can use this information to load the DLL even though it is not in the normal search path. This situation contrasts with load-time linking, in which the system simply terminates the process if it cannot find the DLL.

Run-time dynamic linking can cause problems if the DLL uses the **DllMain** function to perform initialization for each thread of a process, because the entry-point is not called for threads that existed before **LoadLibrary** or **LoadLibraryEx** is called. For an example showing how to deal with this problem, see *Using Thread Local Storage in a Dynamic Link Library*.

Dynamic Link Library Data

Win32-based DLLs can contain global data or local data.

Variable Scope

The default scope of DLL variables is the same as that of variables declared in the application. Global variables in a DLL source code file are global to each process using the DLL. Static variables have scope limited to the block in which they are declared. As a result, each process has its own instance of the DLL global and static variables by default.

Your development tools may allow you to override the default scope of global and static variables. For more information, see the documentation included with your development tools.

Dynamic Memory Allocation

When a DLL allocates memory using any of the memory allocation functions (**GlobalAlloc**, **LocalAlloc**, **HeapAlloc**, and **VirtualAlloc**), the memory is allocated in the virtual address space of the calling process and is accessible only to the threads of that process.

A DLL can use file mapping to allocate memory that can be shared among processes. For a general discussion of how to use file mapping to create named shared memory, see *File Mapping*. For an example that uses the **DllMain** function to set up shared memory using file mapping, see *Using Shared Memory in a Dynamic Link Library*.

Thread Local Storage

The thread local storage (TLS) functions enable a DLL to allocate an index for storing and retrieving a different value for each thread of a multithreaded process. For example, a spreadsheet application can create a new instance of the same thread each time the user opens a new spreadsheet. A DLL providing the functions for various spreadsheet operations can use TLS to save information about the current state of each spreadsheet (row, column, and so on). For a general discussion of thread local storage, see *Thread Local Storage*. For an example that uses the **DllMain** function to set up thread local storage, see *Using Thread Local Storage in a Dynamic Link Library*.

Warning The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: **_declspec(thread)**. If you use this syntax in a DLL, you will not be able to load the DLL explicitly using **LoadLibrary** or **LoadLibraryEx**. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of **_declspec(thread)**.

Dynamic Link Library Redirection

Problems can occur when an application loads a version of a DLL other than the one with which it shipped. Starting with Windows 2000, you can ensure that your application uses the correct version of a DLL by creating a *redirection file*. The contents of a redirection file are ignored, but its presence forces all DLLs in the application's directory to be loaded from that directory.

The redirection file must be named as follows:

```
appname.local
```

For example, if the application's name is editor.exe, the redirection file is named editor.exe.local. You must install editor.exe.local in the same directory that contains editor.exe. You must also install the DLLs in the same directory.

The **LoadLibrary** and **LoadLibraryEx** functions change their search sequence if a redirection file is present. If a path is specified and there is a redirection file for the application, these functions search for the DLL in the application's directory. If the DLL exists in the application's directory, these functions ignore the specified path and load the DLL from the application's directory. If the module is not in the application's directory, these functions load the DLL from the specified directory.

For example, an application `c:\myapp\myapp.exe` calls **LoadLibrary** using the following path:

```
c:\program files\common files\system\mydll.dll.
```

If `c:\myapp\myapp.exe.local` and `c:\myapp\mydll.dll` exist, **LoadLibrary** will load `c:\myapp\mydll.dll`. Otherwise, **LoadLibrary** will load `c:\program files\common files\system\mydll.dll`.

Note It is good practice to install your application's DLLs in the same directory that contains the application, even if you are not using redirection. It ensures that installing your application will not overwrite other copies of the DLL and cause other applications to fail. In addition, other applications will not overwrite your copy of the DLL and cause your application to fail.

Dynamic Link Library Updates

It is sometimes necessary to replace a DLL with a newer version. Before replacing a DLL, perform a version check to ensure that you are replacing an older version with a

newer version. It is possible to replace a DLL that is in use. The method you use to replace DLLs that are in use depends on the operating system you are using.

On Windows NT/Windows 2000, it is not necessary to restart the computer if you perform the following steps:

1. Use the **MoveFileEx** function to rename the DLL being replaced. Do not specify MOVEFILE_COPY_ALLOWED, and make sure the renamed file is on the same volume that contains the original file. You could also simply rename the file in the same directory by giving it a different extension.
2. Copy the new DLL to the directory that contains the renamed DLL. All applications will now use the new DLL.
3. Use **MoveFileEx** with MOVEFILE_DELAY_UNTIL_REBOOT to delete the renamed DLL.

Before you make this replacement, applications will use the original DLL until it is unloaded. After you make the replacement, applications will use the new DLL. When you write a DLL, you must be careful to ensure that it is prepared for this situation, especially if the DLL maintains global state information or communicates with other services. If the DLL is not prepared for a change in global state information or communication protocols, updating the DLL will require you to restart the computer to ensure that all applications are using the same version of the DLL.

On Windows 95/98, it is necessary to restart the computer. For more information, see the Remarks section of **MoveFileEx**.

Dynamic Link Library Reference

Dynamic Link Library Functions

DisableThreadLibraryCalls

The **DisableThreadLibraryCalls** function disables the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications for the dynamic-link library (DLL) specified by *hLibModule*. This can reduce the size of the working code set for some applications.

```
BOOL DisableThreadLibraryCalls(
  HMODULE hLibModule  // handle to DLL module
);
```

Parameters

hLibModule

[in] Handle to the DLL module for which the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications are to be disabled. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. The **DisableThreadLibraryCalls** function fails if the DLL specified by *hLibModule* has active static thread local storage, or if *hLibModule* is an invalid module handle. To get extended error information, call **GetLastError**.

Remarks

The **DisableThreadLibraryCalls** function lets a DLL disable the DLL_THREAD_ATTACH and DLL_THREAD_DETACH notification calls. This can be a useful optimization for multithreaded applications that have many DLLs, frequently create and delete threads, and whose DLLs do not need these thread-level notifications of attachment/detachment. A remote procedure call (RPC) server application is an example of such an application. In these sorts of applications, DLL initialization routines often remain in memory to service DLL_THREAD_ATTACH and DLL_THREAD_DETACH notifications. By disabling the notifications, the DLL initialization code is not paged in because a thread is created or deleted, thus reducing the size of the application's working code set. To implement the optimization, modify a DLL's DLL_PROCESS_ATTACH code to call **DisableThreadLibraryCalls**.

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.

See Also

Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **FreeLibraryAndExitThread**

DllMain

The **DllMain** function is an optional method of entry into a dynamic-link library (DLL). If the function is used, it is called by the system when processes and threads are initialized and terminated, or upon calls to the **LoadLibrary** and **FreeLibrary** functions.

DllMain is a placeholder for the library-defined function name. Earlier versions of the SDK documentation used **DllEntryPoint** as the entry-point function name. You must specify the actual name you use when you build your DLL. For more information, see the documentation included with your development tools.

```
BOOL WINAPI DllMain(
  HINSTANCE hinstDLL,  // handle to the DLL module
  DWORD fdwReason,     // reason for calling function
  LPVOID lpvReserved   // reserved
);
```

Parameters

hinstDLL

[in] Handle to the DLL module. The value is the base address of the DLL. The **HINSTANCE** of a DLL is the same as the **HMODULE** of the DLL, so *hinstDLL* can be used in calls to functions that require a module handle.

fdwReason

[in] Specifies a flag indicating why the DLL entry-point function is being called. This parameter can be one of the following values.

Value	Meaning
DLL_PROCESS_ATTACH	Indicates that the DLL is being loaded into the virtual address space of the current process as a result of the process starting up or as a result of a call to **LoadLibrary**. DLLs can use this opportunity to initialize any instance data or to use the **TlsAlloc** function to allocate a thread local storage (TLS) index.
DLL_THREAD_ATTACH	Indicates that the current process is creating a new thread. When this occurs, the system calls the entry-point function of all DLLs currently attached to the process. The call is made in the context of the new thread. DLLs can use this opportunity to initialize a TLS slot for the thread. A thread calling the DLL entry-point function with DLL_PROCESS_ATTACH does not call the DLL entry-point function with DLL_THREAD_ATTACH. Note that a DLL's entry-point function is called with this value only by threads created after the DLL is loaded by the process. When a DLL is loaded using **LoadLibrary**, existing threads do not call the entry-point function of the newly loaded DLL.

(continued)

(continued)

Value	Meaning
DLL_THREAD_DETACH	Indicates that a thread is exiting cleanly. If the DLL has stored a pointer to allocated memory in a TLS slot, it uses this opportunity to free the memory. The system calls the entry-point function of all currently loaded DLLs with this value. The call is made in the context of the exiting thread.
DLL_PROCESS_DETACH	Indicates that the DLL is being unloaded from the virtual address space of the calling process as a result of unsuccessfully loading the DLL, termination of the process, or a call to **FreeLibrary**. The DLL can use this opportunity to call the **TlsFree** function to free any TLS indices allocated by using **TlsAlloc** and to free any thread local data.

lpvReserved

[in] Specifies further aspects of DLL initialization and cleanup.

If *fdwReason* is DLL_PROCESS_ATTACH, *lpvReserved* is NULL for dynamic loads and non-NULL for static loads.

If *fdwReason* is DLL_PROCESS_DETACH, *lpvReserved* is NULL if **DllMain** has been called by using **FreeLibrary** and non-NULL if **DllMain** has been called during process termination.

Return Values

When the system calls the **DllMain** function with the DLL_PROCESS_ATTACH value, the function returns TRUE if it succeeds or FALSE if initialization fails. If the return value is FALSE when **DllMain** is called because the process uses the **LoadLibrary** function, **LoadLibrary** returns NULL. (The system immediately calls your entry-point function with DLL_PROCESS_DETACH and unloads the DLL.) If the return value is FALSE when **DllMain** is called during process initialization, the process terminates with an error. To get extended error information, call **GetLastError**.

When the system calls the **DllMain** function with any value other than DLL_PROCESS_ATTACH, the return value is ignored.

Remarks

During initial process startup or after a call to **LoadLibrary**, the system scans the list of loaded DLLs for the process. For each DLL that has not already been called with the DLL_PROCESS_ATTACH value, the system calls the DLL's entry-point function. This call is made in the context of the thread that caused the process address space to change, such as the primary thread of the process or the thread that called **LoadLibrary**. Access to the entry point is serialized by the system on a process-wide basis.

There are cases in which the entry-point function is called for a terminating thread even if the DLL never attached to the thread—for example, the entry-point function was never

called with the DLL_THREAD_ATTACH value in the context of the thread in either of these two situations:

- The thread was the initial thread in the process, so the system called the entry-point function with the DLL_PROCESS_ATTACH value.
- The thread was already running when a call to the **LoadLibrary** function was made, so the system never called the entry-point function for it.

When a DLL is unloaded from a process as a result of an unsuccessful load of the DLL, termination of the process, or a call to **FreeLibrary**, the system does not call the DLL's entry-point function with the DLL_THREAD_DETACH value for the individual threads of the process. The DLL is only sent a DLL_PROCESS_DETACH notification. DLLs can take this opportunity to clean up all resources for all threads known to the DLL.

Warning On attach, the body of your DLL entry-point function should perform only simple initialization tasks, such as setting up thread local storage (TLS), creating objects, and opening files. You must not call **LoadLibrary** in the entry-point function, because you may create dependency loops in the DLL load order. This can result in a DLL being used before the system has executed its initialization code. Similarly, you must not call the **FreeLibrary** function in the entry-point function on detach, because this can result in a DLL being used after the system has executed its termination code.

Calling Win32 functions other than TLS, object-creation, and file functions may result in problems that are difficult to diagnose. For example, calling User, Shell, COM, RPC, and Windows Sockets functions (or any functions that call these functions) can cause access violation errors, because their DLLs call **LoadLibrary** to load other system components. While it is acceptable to create synchronization objects in **DllMain**, you should not perform synchronization in **DllMain** (or a function called by **DllMain**) because all calls to **DllMain** are serialized. Waiting on synchronization objects in **DllMain** can cause a deadlock.

To provide more complex initialization, create an initialization routine for the DLL. You can require applications to call the initialization routine before calling any other routines in the DLL. Otherwise, you can have the initialization routine create a named mutex, and have each routine in the DLL call the initialization routine if the mutex does not exist.

! Requirements

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

+ See Also

Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, *Dynamic Link Library Entry-Point Function*, **FreeLibrary**, **GetModuleFileName**, **LoadLibrary**, **TlsAlloc**, **TlsFree**

FreeLibrary

The **FreeLibrary** function decrements the reference count of the loaded dynamic-link library (DLL) module. When the reference count reaches zero, the module is unmapped from the address space of the calling process and the handle is no longer valid.

```
BOOL FreeLibrary(
  HMODULE hLibModule   // handle to DLL module
);
```

Parameters

hLibModule

[in/out] Handle to the loaded DLL module. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

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

Each process maintains a reference count for each loaded library module. This reference count is incremented each time **LoadLibrary** is called and is decremented each time **FreeLibrary** is called. A DLL module loaded at process initialization due to load-time dynamic linking has a reference count of one. This count is incremented if the same module is loaded by a call to **LoadLibrary**.

Before unmapping a library module, the system enables the DLL to detach from the process by calling the DLL's **DllMain** function, if it has one, with the DLL_PROCESS_DETACH value. Doing so gives the DLL an opportunity to clean up resources allocated on behalf of the current process. After the entry-point function returns, the library module is removed from the address space of the current process.

It is not safe to call **FreeLibrary** from **DllMain**. For more information, see the Remarks section in **DllMain**.

Calling **FreeLibrary** does not affect other processes using the same library module.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Dynamic Link Libraries Overview, Dynamic Link Library Functions, **DllMain**, **FreeLibraryAndExitThread**, **GetModuleHandle**, **LoadLibrary**

FreeLibraryAndExitThread

The **FreeLibraryAndExitThread** function decrements the reference count of a loaded dynamic link library (DLL) by one, and then calls **ExitThread** to terminate the calling thread. The function does not return.

The **FreeLibraryAndExitThread** function gives threads that are created and executed within a dynamic link library an opportunity to safely unload the DLL and terminate themselves.

```
VOID FreeLibraryAndExitThread(
  HMODULE hLibModule,  // handle to the DLL module
  DWORD dwExitCode     // exit code for thread
);
```

Parameters

hLibModule
[in] Handle to the DLL module whose reference count the function decrements. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

dwExitCode
[in] Specifies the exit code for the calling thread.

Return Values

The function has no return value. The function does not return. Invalid *hLibModule* handles are ignored.

Remarks

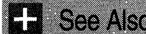
The **FreeLibraryAndExitThread** function is implemented as:

```
FreeLibrary(hLibModule);
ExitThread(dwExitCode);
```

Refer to the reference pages for **FreeLibrary** and **ExitThread** for further information on those functions.

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.



Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **FreeLibrary**, **ExitThread**, **DisableThreadLibraryCalls**

GetModuleFileName

The **GetModuleFileName** function retrieves the full path and file name for the file containing the specified module.

Windows 95/98: The **GetModuleFilename** function will return long file names when an application's version number is greater than or equal to 4.00 and the long file name is available. Otherwise, it returns only 8.3 format file names.

```
DWORD GetModuleFileName(
  HMODULE hModule,    // handle to module
  LPTSTR lpFilename,  // file name of module
  DWORD nSize         // size of buffer
);
```

Parameters

hModule

[in] Handle to the module whose file name is being requested. If this parameter is NULL, **GetModuleFileName** returns the path for the file containing the current process.

lpFilename

[out] Pointer to a buffer that receives the path and file name of the specified module.

nSize

[in] Specifies the length, in characters, of the *lpFilename* buffer. If the length of the path and file name exceeds this limit, the string is truncated.

Return Values

If the function succeeds, the return value is the length, in characters, of the string copied to the buffer.

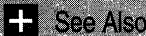
If the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If a DLL is loaded in two processes, its file name in one process may differ in case from its file name in the other process.

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.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.



Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **GetModuleHandle**, **LoadLibrary**

GetModuleHandle

The **GetModuleHandle** function returns a module handle for the specified module if the file has been mapped into the address space of the calling process.

```
HMODULE GetModuleHandle(
  LPCTSTR lpModuleName   // module name
);
```

Parameters

lpModuleName

[in] Pointer to a null-terminated string that contains the name of the module (either a .dll or .exe file). If the file name extension is omitted, the default library extension .dll is appended. The file name string can include a trailing point character (.) to indicate that the module name has no extension. The string does not have to specify a path. When specifying a path, be sure to use backslashes (\), not forward slashes (/). The name is compared (case independently) to the names of modules currently mapped into the address space of the calling process.

If this parameter is NULL, **GetModuleHandle** returns a handle to the file used to create the calling process.

Return Values

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

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

Remarks

The returned handle is not global, inheritable, or duplicative, and it cannot be used by another process.

The handles returned by **GetModuleHandle** and **LoadLibrary** can be used in the same functions—for example, **GetProcAddress**, **FreeLibrary**, or **LoadResource**. The difference between the two functions involves the reference count. **LoadLibrary** maps the module into the address space of the calling process, if necessary, and increments the module's reference count, if it is already mapped. **GetModuleHandle**, however, returns the handle to a mapped module without incrementing its reference count.

Note that the reference count is used in **FreeLibrary** to determine whether to unmap the function from the address space of the process. For this reason, use care when using a handle returned by **GetModuleHandle** in a call to **FreeLibrary** because doing so can cause a dynamic-link library (DLL) module to be unmapped prematurely.

This function must also be used carefully in a multithreaded application. There is no guarantee that the module handle remains valid between the time this function returns the handle and the time it is used by another function. For example, a thread might retrieve a module handle by calling **GetModuleHandle.** Before the thread uses the handle in another function, a second thread could free the module and the system could load another module, giving it the same handle as the module that was recently freed. The first thread would then be left with a module handle that refers to a module different than the one intended.

Requirements

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

Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **FreeLibrary**, **GetModuleFileName**, **GetProcAddress**, **LoadLibrary**, **LoadResource**

GetProcAddress

The **GetProcAddress** function returns the address of the specified exported dynamic-link library (DLL) function.

```
FARPROC GetProcAddress(
  HMODULE hModule,    // handle to DLL module
  LPCSTR lpProcName   // function name
);
```

Parameters

hModule

[in] Handle to the DLL module that contains the function. The **LoadLibrary** or **GetModuleHandle** function returns this handle.

lpProcName

[in] Pointer to a null-terminated string containing the function name, or specifies the function's ordinal value. If this parameter is an ordinal value, it must be in the low-order word; the high-order word must be zero.

Return Values

If the function succeeds, the return value is the address of the DLL's exported function.

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

Remarks

The **GetProcAddress** function is used to retrieve addresses of exported functions in DLLs.

The spelling and case of the function name pointed to by *lpProcName* must be identical to that in the **EXPORTS** statement of the source DLL's module-definition (.DEF) file. The exported names of Win32 API functions may differ from the names you use when calling these functions in your code. This difference is hidden by macros used in the SDK header files. For more information, see *Win32 Function Prototypes*.

The *lpProcName* parameter can identify the DLL function by specifying an ordinal value associated with the function in the **EXPORTS** statement. **GetProcAddress** verifies that the specified ordinal is in the range 1 through the highest ordinal value exported in the .DEF file. The function then uses the ordinal as an index to read the function's address from a function table. If the .DEF file does not number the functions consecutively from 1 to *N* (where *N* is the number of exported functions), an error can occur where **GetProcAddress** returns an invalid, non-NULL address, even though there is no function with the specified ordinal.

In cases where the function may not exist, the function should be specified by name rather than by ordinal 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **FreeLibrary**, **GetModuleHandle**, **LoadLibrary**

LoadLibrary

The **LoadLibrary** function maps the specified executable module into the address space of the calling process.

For additional load options, use the **LoadLibraryEx** function.

```
HMODULE LoadLibrary(
  LPCTSTR lpLibFileName   // file name of module
);
```

Parameters

lpLibFileName

[in] Pointer to a null-terminated string that names the executable module (either a .dll or .exe file). The name specified is the file name of the module and is not related to the name stored in the library module itself, as specified by the **LIBRARY** keyword in the module-definition (.def) file.

If the string specifies a path but the file does not exist in the specified directory, the function fails. When specifying a path, be sure to use backslashes (\), not forward slashes (/).

If the string does not specify a path, the function uses a standard search strategy to find the file. See the **Remarks** for more information.

Return Values

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

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

Windows 95: If you are using **LoadLibrary** to load a module that contains a resource whose numeric identifier is greater than 0x7FFF, **LoadLibrary** fails. If you are attempting to load a 16-bit DLL directly from 32-bit code, **LoadLibrary** fails. If you are attempting to load a DLL whose subsystem version is greater than 4.0, **LoadLibrary** fails. If your **DllMain** function tries to call the Unicode version of a Win32 function, **LoadLibrary** fails.

Remarks

LoadLibrary can be used to map a DLL module and return a handle that can be used in **GetProcAddress** to get the address of a DLL function. **LoadLibrary** can also be used to map other executable modules. For example, the function can specify an .exe file to get

a handle that can be used in **FindResource** or **LoadResource**. However, do not use **LoadLibrary** to run an .exe file, use the **CreateProcess** function.

If the module is a DLL not already mapped for the calling process, the system calls the DLL's **DllMain** function with the DLL_PROCESS_ATTACH value. If the DLL's entry-point function does not return TRUE, **LoadLibrary** fails and returns NULL. (The system immediately calls your entry-point function with DLL_PROCESS_DETACH and unloads the DLL.)

It is not safe to call **LoadLibrary** from **DllMain**. For more information, see the Remarks section in **DllMain**.

Module handles are not global or inheritable. A call to **LoadLibrary** by one process does not produce a handle that another process can use—for example, in calling **GetProcAddress**. The other process must make its own call to **LoadLibrary** for the module before calling **GetProcAddress**.

If no file name extension is specified in the *lpLibFileName* parameter, the default library extension .dll is appended. However, the file name string can include a trailing point character (.) to indicate that the module name has no extension. When no path is specified, the function searches for loaded modules whose base name matches the base name of the module to be loaded. If the name matches, the load succeeds. Otherwise, the function searches for the file in the following sequence:

1. The directory from which the application loaded.
2. The current directory.
3. **Windows 95/98:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT/ 2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.
4. **Windows NT/ 2000:** The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

The first directory searched is the one directory containing the image file used to create the calling process (for more information, see the **CreateProcess** function). Doing this allows private dynamic link library (DLL) files associated with a process to be found without adding the process's installed directory to the PATH environment variable.

Windows 2000: If a path is specified and there is a redirection file for the application, the function searches for the module in the application's directory. If the module exists in the application's directory, the **LoadLibrary** function ignores the specified path and loads

the module from the application's directory. If the module does not exist in the application's directory, **LoadLibrary** loads the module from the specified directory.

The Visual C++ compiler supports a syntax that enables you to declare thread-local variables: **_declspec(thread)**. If you use this syntax in a DLL, you will not be able to load the DLL explicitly using **LoadLibrary** or **LoadLibraryEx**. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of **_declspec(thread)**.

Requirements

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

Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **DllMain**, **FindResource**, **FreeLibrary**, **GetProcAddress**, **GetSystemDirectory**, **GetWindowsDirectory**, **LoadLibraryEx**, **LoadResource**

LoadLibraryEx

The **LoadLibraryEx** function maps a specified executable module into the address space of the calling process. The executable module can be a .dll or an .exe file. The specified module may cause other modules to be mapped into the address space.

```
HMODULE LoadLibraryEx(
  LPCTSTR lpLibFileName,  // file name of module
  HANDLE hFile,           // reserved, must be NULL
  DWORD dwFlags           // entry-point execution option
);
```

Parameters

lpLibFileName

[in] Pointer to a null-terminated string that names the executable module (either a .dll or an .exe file). The name specified is the file name of the executable module. This name is not related to the name stored in a library module itself, as specified by the **LIBRARY** keyword in the module-definition (.DEF) file.

If the string specifies a path, but the file does not exist in the specified directory, the function fails. When specifying a path, be sure to use backslashes (\), not forward slashes (/).

If the string does not specify a path, and the file name extension is omitted, the function appends the default library extension .dll to the file name. However, the file name string can include a trailing point character (.) to indicate that the module name has no extension.

If the string does not specify a path, the function uses a standard search strategy to find the file. See the Remarks for more information.

If mapping the specified module into the address space causes the system to map in other, associated executable modules, the function can use either the standard search strategy or an alternate search strategy to find those modules. See the Remarks for more information.

hFile

This parameter is reserved for future use. It must be NULL.

dwFlags

[in] Specifies the action to take when loading the module. If no flags are specified, the behavior of this function is identical to that of the **LoadLibrary** function. This parameter can be one of the following values.

Flag	Meaning
DONT_RESOLVE_DLL_REFERENCES	**Windows NT/2000:** If this value is used, and the executable module is a DLL, the system does not call **DllMain** for process and thread initialization and termination. Also, the system does not load additional executable modules that are referenced by the specified module. If this value is not used, and the executable module is a DLL, the system calls **DllMain** for process and thread initialization and termination. The system loads additional executable modules that are referenced by the specified module.
LOAD_LIBRARY_AS_DATAFILE	If this value is used, the system maps the file into the calling process's virtual address space as if it were a data file. Nothing is done to execute or prepare to execute the mapped file. Use this flag when you want to load a DLL only to extract messages or resources from it. **Windows NT/2000:** You can use the resulting module handle with any Win32 functions that operate on resources. **Windows 95/98:** You can use the resulting module handle only with resource management functions such as **EnumResourceLanguages**, **EnumResourceNames**, **EnumResourceTypes**, **FindResource**, **FindResourceEx**,

(continued)

(continued)

Flag	Meaning
	LoadResource, and **SizeofResource**. You cannot use this handle with specialized resource management functions such as **LoadBitmap**, **LoadCursor**, **LoadIcon**, **LoadImage**, and **LoadMenu**.
LOAD_WITH_ALTERED_SEARCH_PATH	If this value is used, and *lpLibFileName* specifies a path, the system uses the alternate file search strategy discussed in the Remarks section to find associated executable modules that the specified module causes to be loaded. If this value is not used, or if *lpLibFileName* does not specify a path, the system uses the standard search strategy discussed in the Remarks section to find associated executable modules that the specified module causes to be loaded.

Return Values

If the function succeeds, the return value is a handle to the mapped executable module.

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

Windows 95: If you are using **LoadLibraryEx** to load a module that contains a resource whose numeric identifier is greater than 0x7FFF, **LoadLibraryEx** fails. If you are attempting to load a 16-bit DLL directly from 32-bit code, **LoadLibraryEx** fails. If you are attempting to load a DLL whose subsystem version is greater than 4.0, **LoadLibraryEx** fails. If your **DllMain** function tries to call the Unicode version of a Win32 function, **LoadLibraryEx** fails.

Remarks

The calling process can use the handle returned by this function to identify the module in calls to the **GetProcAddress**, **FindResource**, and **LoadResource** functions.

The **LoadLibraryEx** function is very similar to the **LoadLibrary** function. The differences consist of a set of optional behaviors that **LoadLibraryEx** provides. First, **LoadLibraryEx** can map a DLL module without calling the **DllMain** function of the DLL. Second, **LoadLibraryEx** can use either of two file search strategies to find executable modules that are associated with the specified module. Third, **LoadLibraryEx** can load a module in a way that is optimized for the case where the module will never be executed, loading the module as if it were a data file. You select these optional behaviors by setting the *dwFlags* parameter; if *dwFlags* is zero, **LoadLibraryEx** behaves identically to **LoadLibrary.**

It is not safe to call **LoadLibraryEx** from **DllMain**. For more information, see the Remarks section in **DllMain**.

If no path is specified in the *lpLibFileName* parameter, and the base file name does not match the base file name of a loaded module, the **LoadLibraryEx** function uses the same standard file search strategy that **LoadLibrary**, **SearchPath**, and **OpenFile** use to find the executable module and any associated executable modules that it causes to be loaded. This standard strategy searches for a file in the following sequence:

1. The directory from which the application loaded.
2. The current directory.
3. **Windows 95/98:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.
4. **Windows NT/2000:** The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

If a path is specified, and the *dwFlags* parameter is set to LOAD_WITH_ALTERED_SEARCH_PATH, the **LoadLibraryEx** function uses an alternate file search strategy to find any executable modules that the specified module causes to be loaded. This alternate strategy searches for a file in the following sequence:

1. The directory specified by the *lpLibFileName* path. In other words, the directory that the specified executable module is in.
2. The current directory.
3. **Windows 95/98:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.
4. **Windows NT/2000:** The 16-bit Windows system directory. There is no function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

Note The standard file search strategy and the alternate search strategy differ in just one way: the standard strategy starts its search in the calling application's directory, and the alternate strategy starts its search in the directory of the executable module that **LoadLibraryEx** is loading.

If you specify the alternate search strategy, its behavior continues until all associated executable modules have been located. After the system starts processing DLL initialization routines, the system reverts to the standard search strategy.

Windows 2000: If a path is specified and there is a redirection file associated with the application, the **LoadLibraryEx** function searches for the module in the application directory. If the module exists in the application directory, **LoadLibraryEx** ignores the path specification and loads the module from the application directory. If the module does not exist in the application directory, the function loads the module from the specified directory.

Visual C++: The **Visual C++** compiler supports a syntax that enables you to declare thread-local variables: **_declspec(thread)**. If you use this syntax in a DLL, you will not be able to load the DLL explicitly using **LoadLibrary** or **LoadLibraryEx**. If your DLL will be loaded explicitly, you must use the thread local storage functions instead of **_declspec(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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Dynamic Link Libraries Overview, *Dynamic Link Library Functions*, **DllMain**, **FindResource**, **FreeLibrary**, **GetProcAddress**, **GetSystemDirectory**, **GetWindowsDirectory**, **LoadLibrary**, **LoadResource**, **OpenFile**, **SearchPath**

Synchronization

The Microsoft Win32 API provides a variety of ways to coordinate multiple threads of execution. Such coordination is between multiple threads is considered *synchronization*.

The rest of this chapter provides overview information, and is geared toward getting you familiar with the issues surrounding synchronization. While this overview is complete, and should get you far in preparing to deal with synchronization issues in your Windows application, the actual programmatic elements were too numerous (page consuming,

that is) to fit within the 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. Rather than choosing selected functions (and thereby, almost certainly leaving out the one *you* really needed), I've done something better: provided them for you on DVD.

Getting More Information About Synchronization

The companion DVD that's bundled inside this volume of the *Microsoft Win32 Developer's Reference Library* has the complete set of reference information for Synchronization, with complete programming element reference and overview information. If you haven't already, install the companion DVD, and all window programming information (plus a bunch more) will be at your fingertips.

About Synchronization

To synchronize access to a resource, use one of the synchronization objects in one of the wait functions. The state of a synchronization object is either *signaled* or *nonsignaled*. The wait functions allow a thread to block its own execution until a specified nonsignaled object is set to the signaled state.

The following are other synchronization mechanisms:

- overlapped input and output
- asynchronous procedure calls
- critical section objects
- interlocked variable access

Wait Functions

The Win32 API provides a set of *wait functions* to allow a thread to block its own execution. The wait functions do not return until the specified criteria have been met. The type of wait function determines the set of criteria used. When a wait function is called, it checks whether the wait criteria have been met. If the criteria have not been met, the calling thread enters an efficient wait state, consuming very little processor time while waiting for the criteria to be met.

There are four types of wait functions:

- single-object
- multiple-object
- alertable
- registered

Single-Object Wait Functions

The **SignalObjectAndWait**, **WaitForSingleObject**, and **WaitForSingleObjectEx** functions require a handle to one synchronization object. These functions return when one of the following occurs:

- The specified object is in the signaled state.
- The time-out interval elapses. The time-out interval can be set to INFINITE to specify that the wait will not time out.

The **SignalObjectAndWait** function enables the calling thread to atomically set the state of an object to signaled and wait for the state of another object to be set to signaled.

Multiple-Object Wait Functions

The **WaitForMultipleObjects**, **WaitForMultipleObjectsEx**, **MsgWaitForMultipleObjects**, and **MsgWaitForMultipleObjectsEx** functions enable the calling thread to specify an array containing one or more synchronization object handles. These functions return when one of the following occurs:

- The state of any one of the specified objects is set to signaled or the states of all objects have been set to signaled. You control whether one or all of the states will be used in the function call.
- The time-out interval elapses. The time-out interval can be set to INFINITE to specify that the wait will not time out.

The **MsgWaitForMultipleObjects** and **MsgWaitForMultipleObjectsEx** function allow you to specify input event objects in the object handle array. This is done when you specify the type of input to wait for in the thread's input queue.

For example, a thread could use **MsgWaitForMultipleObjects** to block its execution until the state of a specified object has been set to signaled and there is mouse input available in the thread's input queue. The thread can use the **GetMessage** or **PeekMessage** function to retrieve the input.

When waiting for the states of all objects to be set to signaled, these multiple-object functions do not modify the states of the specified objects until the states of all objects have been set signaled. For example, the state of a mutex object can be signaled, but the calling thread does not get ownership until the states of the other objects specified in the array have also been set to signaled. In the meantime, some other thread may get ownership of the mutex object, thereby setting its state to nonsignaled.

Alertable Wait Functions

The **MsgWaitForMultipleObjectsEx**, **SignalObjectAndWait**, **WaitForMultipleObjectsEx,** and **WaitForSingleObjectEx** functions differ from the other wait functions in that they can optionally perform an *alertable wait operation*. In an alertable wait operation, the function can return when the specified conditions are met, but it can also return if the system queues an I/O completion routine or an APC for execution by the waiting thread. For more information about alertable wait operations

and I/O completion routines, see *Synchronization and Overlapped Input and Output*. For more information about APCs, see *Asynchronous Procedure Calls*.

Registered Wait Functions

The **RegisterWaitForSingleObject** function differs from the other wait functions in that the wait operation is performed by a thread from the thread pool. When the specified conditions are met, the callback function is executed by a worker thread from the thread pool.

By default, a registered wait operation is a multiple-wait operation. The system resets the timer every time the event is signaled (or the time-out interval elapses) until you call the **UnregisterWaitEx** function to cancel the operation. To specify that a wait operation should be executed only once, set the *dwFlags* parameter of **RegisterWaitForSingleObject** to WT_EXECUTEONLYONCE.

Wait Functions and Synchronization Objects

Before returning, a wait function can modify the states of some types of synchronization objects. Modification occurs only for the object or objects whose signaled state caused the function to return. A wait function can modify the states of synchronization objects as follows:

- The count of a semaphore object decreases by one, and the state of the semaphore is set to nonsignaled if its count is zero.
- The states of mutex, auto-reset event, and change-notification objects are set to nonsignaled.
- The state of a synchronization timer is set to nonsignaled.
- The states of manual-reset event, manual-reset timer, process, thread, and console input objects are not affected by a wait function.

Wait Functions and Creating Windows

You have to be careful when using the wait functions and code that directly or indirectly creates windows. If a thread creates any windows, it must process messages. Message broadcasts are sent to all windows in the system. If you have a thread that uses a wait function with no time-out interval, the system will deadlock. Two examples of code that indirectly creates windows are DDE and COM **CoInitialize**. Therefore, if you have a thread that creates windows, use **MsgWaitForMultipleObjects** or **MsgWaitForMultipleObjectsEx**, rather than the other wait functions.

Synchronization Objects

A *synchronization object* is an object whose handle can be specified in one of the wait functions to coordinate the execution of multiple threads. More than one process can have a handle to the same synchronization object, making interprocess synchronization possible.

The following object types are provided exclusively for synchronization:

Type	Description
Event	Notifies one or more waiting threads that an event has occurred. For more information, see *Event Objects*.
Mutex	Can be owned by only one thread at a time, enabling threads to coordinate mutually exclusive access to a shared resource. For more information, see *Mutex Objects.*
Semaphore	Maintains a count between zero and some maximum value, limiting the number of threads that are simultaneously accessing a shared resource. For more information, see *Semaphore Objects*.
Waitable timer	Notifies one or more waiting threads that a specified time has arrived. For more information, see *Waitable Timer Objects*.

Though available for other uses, the following objects can also be used for synchronization.

Object	Description
Change notification	Created by the **FindFirstChangeNotification** function, its state is set to signaled when a specified type of change occurs within a specified directory or directory tree. For more information, see *File I/O*.
Console input	Created when a console is created. The handle to console input is returned by the **CreateFile** function when CONIN$ is specified, or by the **GetStdHandle** function. Its state is set to signaled when there is unread input in the console's input buffer, and set to nonsignaled when the input buffer is empty. For more information about consoles, see *Consoles and Character-Mode Support*.
Job	Created by calling the **CreateJobObject** function. The state of a job object is set to signaled when all its processes are terminated because the specified end-of-job time limit has been exceeded. For more information about job objects, see *Job Objects*.
Process	Created by calling the **CreateProcess** function. Its state is set to nonsignaled while the process is running, and set to signaled when the process terminates. For more information about processes, see *Processes and Threads*.
Thread	Created when a new thread is created by calling the **CreateProcess**, **CreateThread**, or **CreateRemoteThread** function. Its state is set to nonsignaled while the thread is running, and set to signaled when the thread terminates. For more information about threads, see *Processes and Threads*.

In some circumstances, you can also use a file, named pipe, or communications device as a synchronization object; however, their use for this purpose is discouraged. Instead, use asynchronous I/O and wait on the event object set in the **OVERLAPPED** structure. It

is safer to use the event object because of the confusion that can occur when multiple simultaneous overlapped operations are performed on the same file, named pipe, or communications device. In this situation, there is no way to know which operation caused the object's state to be signaled.

For additional information about I/O operations on files, named pipes, or communications, see *Synchronization and Overlapped Input and Output*.

Event Objects

An *event object* is a synchronization object whose state can be explicitly set to signaled by use of the **SetEvent** or **PulseEvent** function. Following are the two types of event object:

Object	Description
Manual-reset event	An event object whose state remains signaled until it is explicitly reset to nonsignaled by the **ResetEvent** function. While it is signaled, any number of waiting threads, or threads that subsequently specify the same event object in one of the wait functions, can be released.
Auto-reset event	An event object whose state remains signaled until a single waiting thread is released, at which time the system automatically sets the state to nonsignaled. If no threads are waiting, the event object's state remains signaled.

The event object is useful in sending a signal to a thread indicating that a particular event has occurred. For example, in overlapped input and output, the system sets a specified event object to the signaled state when the overlapped operation has been completed. A single thread can specify different event objects in several simultaneous overlapped operations, then use one of the multiple-object wait functions to wait for the state of any one of the event objects to be signaled.

A thread uses the **CreateEvent** function to create an event object. The creating thread specifies the initial state of the object and whether it is a manual-reset or auto-reset event object. The creating thread can also specify a name for the event object. Threads in other processes can open a handle to an existing event object by specifying its name in a call to the **OpenEvent** function. For additional information about names for mutex, event, semaphore, and timer objects, see *Interprocess Synchronization*.

A thread can use the **PulseEvent** function to set the state of an event object to signaled and then reset it to nonsignaled after releasing the appropriate number of waiting threads. For a manual-reset event object, all waiting threads are released. For an auto-reset event object, the function releases only a single waiting thread, even if multiple threads are waiting. If no threads are waiting, **PulseEvent** simply sets the state of the event object to nonsignaled and returns.

Mutex Objects

A mutex object is a synchronization object whose state is set to signaled when it is not owned by any thread, and nonsignaled when it is owned. Only one thread at a time can own a mutex object, whose name comes from the fact that it is useful in coordinating mutually exclusive access to a shared resource. For example, to prevent two threads from writing to shared memory at the same time, each thread waits for ownership of a mutex object before executing the code that accesses the memory. After writing to the shared memory, the thread releases the mutex object.

A thread uses the **CreateMutex** function to create a mutex object. The creating thread can request immediate ownership of the mutex object and can also specify a name for the mutex object. Threads in other processes can open a handle to an existing mutex object by specifying its name in a call to the **OpenMutex** function. For additional information about names for mutex, event, semaphore, and timer objects, see *Interprocess Synchronization*.

Any thread with a handle to a mutex object can use one of the wait functions to request ownership of the mutex object. If the mutex object is owned by another thread, the wait function blocks the requesting thread until the owning thread releases the mutex object using the **ReleaseMutex** function. The return value of the wait function indicates whether the function returned for some reason other than the state of the mutex being set to signaled.

Threads that are waiting for ownership of a mutex are placed in a first in, first out (FIFO) queue. Therefore, the first thread to wait on the mutex will be the first to receive ownership of the mutex, regardless of thread priority. However, kernel-mode APCs and events that suspend a thread will cause the system to remove the thread from the queue. When the thread resumes its wait for the mutex, it is placed at the end of the queue.

After a thread obtains ownership of a mutex, it can specify the same mutex in repeated calls to the wait functions without blocking its execution. This prevents a thread from deadlocking itself while waiting for a mutex that it already owns. To release its ownership under such circumstances, the thread must call **ReleaseMutex** once for each time that the mutex satisfied the conditions of a wait function.

If a thread terminates without releasing its ownership of a mutex object, the mutex object is considered to be abandoned. A waiting thread can acquire ownership of an abandoned mutex object, but the wait function's return value indicates that the mutex object is abandoned. It is best to assume that an abandoned mutex object indicates that an error has occurred and that any shared resource being protected by the mutex object is in an undefined state. If the thread proceeds as though the mutex object had not been abandoned, its "abandoned" flag is cleared when the thread releases its ownership. This restores normal behavior if a handle to the mutex object is subsequently specified in a wait function.

Semaphore Objects

A *semaphore object* is a synchronization object that maintains a count between zero and a specified maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, no more threads can successfully wait for the semaphore object state to become signaled. The state of a semaphore is set to signaled when its count is greater than zero, and nonsignaled when its count is zero.

The semaphore object is useful in controlling a shared resource that can support a limited number of users. It acts as a gate that limits the number of threads sharing the resource to a specified maximum number. For example, an application might place a limit on the number of windows that it creates. It uses a semaphore with a maximum count equal to the window limit, decrementing the count whenever a window is created and incrementing it whenever a window is closed. The application specifies the semaphore object in call to one of the wait functions before each window is created. When the count is zero—indicating that the window limit has been reached—the wait function blocks execution of the window-creation code.

A thread uses the **CreateSemaphore** function to create a semaphore object. The creating thread specifies the initial count and the maximum value of the count for the object. The initial count must be neither less than zero nor greater than the maximum value. The creating thread can also specify a name for the semaphore object. Threads in other processes can open a handle to an existing semaphore object by specifying its name in a call to the **OpenSemaphore** function. For additional information about names for mutex, event, semaphore, and timer objects, see *Interprocess Synchronization*.

Threads that are waiting for a semaphore are placed in a first in, first out (FIFO) queue. Therefore, the first thread to wait on the semaphore will be the first to successfully complete the wait, regardless of thread priority. However, kernel-mode APCs and events that suspend a thread will cause the system to remove the thread from the queue. When the thread resumes its wait for the semaphore, it is placed at the end of the queue.

Each time one of the wait functions returns because the state of a semaphore was set to signaled, the count of the semaphore is decreased by one. The **ReleaseSemaphore** function increases a semaphore's count by a specified amount. The count can never be less than zero or greater than the maximum value.

The initial count of a semaphore is typically set to the maximum value. The count is then decremented from that level as the protected resource is consumed. Alternatively, you can create a semaphore with an initial count of zero to block access to the protected resource while the application is being initialized. After initialization, you can use **ReleaseSemaphore** to increment the count to the maximum value.

A thread that owns a mutex object can wait repeatedly for the same mutex object to become signaled without its execution becoming blocked. A thread that waits repeatedly for the same semaphore object, however, decrements the semaphore's count each time a wait operation is completed; the thread is blocked when the count gets to zero. Similarly, only the thread that owns a mutex can successfully call the **ReleaseMutex**

function, though any thread can use **ReleaseSemaphore** to increase the count of a semaphore object.

A thread can decrement a semaphore's count more than once by repeatedly specifying the same semaphore object in calls to any of the wait functions. However, calling one of the multiple-object wait functions with an array that contains multiple handles of the same semaphore does not result in multiple decrements.

Timer Queues

The **CreateTimerQueue** function creates a queue for timers. Timers in this queue, known as *timer-queue timers*, are lightweight objects that enable you to specify a callback function to be called when the specified due time arrives. The wait operation is performed by a thread in the thread pool.

To add a timer to the queue, call the **CreateTimerQueueTimer** function. To update a timer-queue timer, call the **ChangeTimerQueueTimer** function. You can specify a callback function to be executed by a worker thread from the thread pool when the timer expires.

A timer-queue timer is set to the signaled state when its specified due time arrives. Any thread with a handle to the timer can use one of the wait functions to wait for the timer state to be set to signaled.

To cancel a pending timer, call the **DeleteTimerQueueTimer** function. When you are finished with the queue of timers, call the **DeleteTimerQueueEx** function to delete the timer queue. Any pending timers in the queue are canceled and deleted.

Waitable Timer Objects

A "waitable" timer object is a synchronization object whose state is set to signaled when the specified due time arrives. There are two types of waitable timers that can be created: manual-reset and synchronization. A timer of either type can also be a periodic timer:

Object	Description
manual-reset timer	A timer whose state remains signaled until **SetWaitableTimer** is called to establish a new due time.
synchronization timer	A timer whose state remains signaled until a thread completes a wait operation on the timer object.
periodic timer	A timer that is reactivated each time the specified period expires, until the timer is reset or canceled. A periodic timer is either a periodic manual-reset timer or a periodic synchronization timer.

A thread uses the **CreateWaitableTimer** function to create a timer object. Specify TRUE for the *bManualReset* parameter to create a manual-reset timer and FALSE to create a synchronization timer. The creating thread can specify a name for the timer object in the *lpTimerName* parameter. Threads in other processes can open a handle to an existing

timer by specifying its name in a call to the **OpenWaitableTimer** function. Any thread with a handle to a timer object can use one of the wait functions to wait for the timer state to be set to signaled.

- The thread calls the **SetWaitableTimer** function to activate the timer. Note the use of the following parameters for **SetWaitableTimer**:
- Use the *lpDueTime* parameter to specify the time at which the timer is to be set to the signaled state. When a manual-reset timer is set to the signaled state, it remains in this state until **SetWaitableTimer** establishes a new due time. When a synchronization timer is set to the signaled state, it remains in this state until a thread completes a wait operation on the timer object.
- Use the *lPeriod* parameter of the **SetWaitableTimer** function to specify the timer period. If the period is not zero, the timer is a periodic timer; it is reactivated each time the period expires, until the timer is reset or canceled. If the period is zero, the timer is not a periodic timer; it is signaled once and then deactivated.

A thread can use the **CancelWaitableTimer** function to set the timer to the inactive state. To reset the timer, call **SetWaitableTimer**. When you are finished with the timer object, call **CloseHandle** to close the handle to the timer object.

Interprocess Synchronization

Multiple processes can have handles to the same event, mutex, semaphore, or timer object, so these objects can be used to accomplish interprocess synchronization. The process that creates an object can use the handle returned by the creation function (**CreateEvent**, **CreateMutex**, **CreateSemaphore**, or **CreateWaitableTimer**). Other processes can open a handle to the object by using its name, or through inheritance or duplication.

Object Names

Named objects provide an easy way for processes to share object handles. Once a process has created a named event, mutex, semaphore, or timer object, other processes can use the name to call the appropriate function (**OpenEvent**, **OpenMutex**, **OpenSemaphore**, or **OpenWaitableTimer**) to open a handle to the object. Name comparison is case sensitive.

The names of event, semaphore, mutex, waitable timer, file-mapping, and job objects share the same name space. If you try to create an object using a name that is in use by an object of another type, the function fails and **GetLastError** returns ERROR_INVALID_HANDLE. Therefore, when creating named objects, use unique names and be sure to check function return values for duplicate-name errors.

If you try to create an object using a name that is in use by an object of same type, the function succeeds, returning a handle to the existing object, and **GetLastError** returns ERROR_ALREADY_EXISTS. For example, if the name specified in a call to the **CreateMutex** function matches the name of an existing mutex object, the function returns a handle to the existing object. In this case, the call to **CreateMutex** is equivalent

to a call to the **OpenMutex** function. Having multiple processes use **CreateMutex** for the same mutex is therefore equivalent to having one process that calls **CreateMutex** while the other processes call **OpenMutex**, except that it eliminates the need to ensure that the creating process is started first. When using this technique for mutex objects, however, none of the calling processes should request immediate ownership of the mutex. If multiple processes do request immediate ownership, it can be difficult to predict which process actually gets the initial ownership.

Terminal Services: A Terminal Services environment has a global name space for events, semaphores, mutexes, waitable timers, file-mapping objects, and job objects. In addition, each Terminal Services client session has its own separate name space for these objects. Terminal Services client processes can use object names with a "Global\" or "Local\" prefix to explicitly create an object in the global or session name space. For more information, see *Kernel Object Name Spaces*.

Windows 2000: On Windows 2000 systems without Terminal Services running, the "Global\" and "Local\" prefixes are ignored. The names of events, semaphores, mutexes, waitable timers, file-mapping objects, and job objects share the same name space.

Windows NT 4.0 and earlier, Windows 95/98: The names of events, semaphores, mutexes, waitable timers, and file-mapping objects share the same name space. The functions for creating or opening these objects fail if you specify a name containing the backslash character (\).

Object Inheritance

When you create a process with the **CreateProcess** function, you can specify that the process inherit handles to mutex, event, semaphore, or timer objects using the **SECURITY_ATTRIBUTES** structure. The handle inherited by the process has the same access to the object as the original handle. The inherited handle appears in the handle table of the created process, but you must communicate the handle value to the created process. You can do this by specifying the value as a command-line argument when you call **CreateProcess**. The created process then uses the **GetCommandLine** function to retrieve the command-line string and convert the handle argument into a usable handle. For more information, see *Inheritance*.

Object Duplication

The **DuplicateHandle** function creates a duplicate handle that can be used by another specified process. This method of sharing object handles is more complex than using named objects or inheritance. It requires communication between the creating process and the process into which the handle is duplicated. The necessary information (the handle value and process identifier) can be communicated by any of the interprocess communication methods, such as named pipes or named shared memory.

Synchronization Object Security and Access Rights

Windows NT/Windows 2000 security enables you to control access to event, mutex, semaphore, and waitable timer objects. Timer queues, interlocked variables, and critical

section objects are not securable. For more information about security, see *Access-Control Model.*

You can specify a security descriptor for an interprocess synchronization object when you call the **CreateEvent**, **CreateMutex**, **CreateSemaphore**, or **CreateWaitableTimer** function. To get or set the security descriptor of an event, mutex, semaphore, or waitable timer object, call the **GetNamedSecurityInfo**, **SetNamedSecurityInfo**, **GetSecurityInfo**, or **SetSecurityInfo** functions.

The handles returned by **CreateEvent**, **CreateMutex**, **CreateSemaphore**, and **CreateWaitableTimer** have full access to the new object. When you call the **OpenEvent**, **OpenMutex**, **OpenSemaphore**, and **OpenWaitableTimer** functions, the system checks the requested access rights against the object's security descriptor.

The valid access rights for all interprocess synchronization objects include the DELETE, READ_CONTROL, SYNCHRONIZE, WRITE_DAC, and WRITE_OWNER standard access rights.

The following table lists the specific access rights for event objects:

Value	Meaning
EVENT_ALL_ACCESS	Specifies all possible access rights for an event object.
EVENT_MODIFY_STATE	Specifies modify state access, which is required for the **ResetEvent** and **PulseEvent** functions.

The following table lists the specific access rights for mutex objects:

Value	Meaning
MUTEX_ALL_ACCESS	Specifies all possible access rights for a mutex object.
MUTEX_MODIFY_STATE	Specifies modify state access, which is required for the **ReleaseMutex** function.

The following table lists the specific access rights for semaphore objects:

Value	Meaning
SEMAPHORE_ALL_ACCESS	Specifies all possible access rights for a semaphore object.
SEMAPHORE_MODIFY_STATE	Specifies modify state access, which is required for the **ReleaseSemaphore** function.

The following table lists the specific access rights for waitable timer objects:

Value	Meaning
TIMER_ALL_ACCESS	Specifies all possible access rights for a waitable timer object.
TIMER_MODIFY_STATE	Specifies modify state access, which is required for the **SetWaitableTimer** and **CancelWaitableTimer** functions.
TIMER_QUERY_STATE	Reserved for future use.

To read or write the SACL of an interprocess synchronization object, you must request the ACCESS_SYSTEM_SECURITY access right. For more information, see *Access-Control Lists (ACLs)* and *SACL Access Right*.

Synchronization and Overlapped Input and Output

The Win32 API supports both synchronous and asynchronous (or overlapped) I/O operations on files, named pipes, and serial communications devices. The **WriteFile**, **ReadFile**, **DeviceIoControl**, **WaitCommEvent**, **ConnectNamedPipe**, and **TransactNamedPipe** functions can be performed either synchronously or asynchronously. The **ReadFileEx** and **WriteFileEx** functions can be performed asynchronously only.

When a function is executed synchronously, it does not return until the operation has been completed. This means that the execution of the calling thread can be blocked for an indefinite period while it waits for a time-consuming operation to finish. Functions called for overlapped operation can return immediately, even though the operation has not been completed. This enables a time-consuming I/O operation to be executed in the background while the calling thread is free to perform other tasks. For example, a single thread can perform simultaneous I/O operations on different handles, or even simultaneous read and write operations on the same handle.

To synchronize its execution with the completion of the overlapped operation, the calling thread uses the **GetOverlappedResult** function or one of the wait functions to determine when the overlapped operation has been completed. You can also use the **HasOverlappedIoCompleted** macro to poll for completion.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle.

Overlapped operations require a file, named pipe, or communications device that was created with the FILE_FLAG_OVERLAPPED flag. To call a function to perform an overlapped operation, the calling thread must specify a pointer to an **OVERLAPPED** structure. If this pointer is NULL, the function return value may incorrectly indicate that the operation completed. The **OVERLAPPED** structure must contain a handle to a manual-reset—not an auto-reset—event object. The system sets the state of the event object to nonsignaled when a call to the I/O function returns before the operation has been completed. The system sets the state of the event object to signaled when the operation has been completed.

When a function is called to perform an overlapped operation, it is possible that the operation will be completed before the function returns. When this happens, the results are handled as if the operation had been performed synchronously. If the operation was not completed, however, the function's return value is FALSE, and the **GetLastError** function returns ERROR_IO_PENDING.

A thread can manage overlapped operations by either of two methods:

- Use the **GetOverlappedResult** function to wait for the overlapped operation to be completed.
- Specify a handle to the **OVERLAPPED** structure's manual-reset event object in one of the wait functions and then call **GetOverlappedResult** after the wait function returns. The **GetOverlappedResult** function returns the results of the completed overlapped operation, and for functions in which such information is appropriate, it reports the actual number of bytes that were transferred.

When performing multiple simultaneous overlapped operations, the calling thread must specify an **OVERLAPPED** structure with a different manual-reset event object for each operation. To wait for any one of the overlapped operations to be completed, the thread specifies all the manual-reset event handles as wait criteria in one of the multiple-object wait functions. The return value of the multiple-object wait function indicates which manual-reset event object was signaled, so the thread can determine which overlapped operation caused the wait operation to be completed.

If no event object is specified in the **OVERLAPPED** structure, the system signals the state of the file, named pipe, or communications device when the overlapped operation has been completed. Thus, you can specify these handles as synchronization objects in a wait function, though their use for this purpose can be difficult to manage. When performing simultaneous overlapped operations on the same file, named pipe, or communications device, there is no way to know which operation caused the object's state to be signaled. It is safer to use a separate event object for each overlapped operation.

For examples that illustrate the use of overlapped operations, completion routines, and the **GetOverlappedResult** function, see *Using Pipes*.

Asynchronous Procedure Calls

An asynchronous procedure call (APC) is a function that executes asynchronously in the context of a particular thread. When an APC is queued to a thread, the system issues a software interrupt. The next time the thread is scheduled, it will run the APC function. APCs made by the system are called "kernel-mode APCs." APCs made by an application are called "user-mode APCs." A thread must be in an alertable state to run a user-mode APC.

Each thread has its own APC queue. An application queues an APC to a thread by calling the **QueueUserAPC** function. The calling thread specifies the address of an APC function in the call to **QueueUserAPC**. The queuing of an APC is a request for the thread to call the APC function.

When a user-mode APC is queued, the thread to which it is queued is not directed to call the APC function unless it is in an alertable state. A thread enters an alertable state when it calls the **SleepEx**, **SignalObjectAndWait**, **MsgWaitForMultipleObjectsEx**, **WaitForMultipleObjectsEx**, or **WaitForSingleObjectEx** function. Note that you can not use **WaitForSingleObjectEx** to wait on the handle to the object for which the APC is queued. Otherwise, when the asynchronous operation is completed, the handle is set to the signaled state and the thread is no longer in an alertable wait state, so the APC function will not be executed. However, the APC is still queued, so the APC function will be executed if you call another alertable wait function.

Note that the **ReadFileEx**, **SetWaitableTimer**, and **WriteFileEx** functions are implemented using an APC as the completion notification callback mechanism.

Critical Section Objects

Critical section objects provide synchronization similar to that provided by mutex objects, except that critical section objects can be used only by the threads of a single process. Event, mutex, and semaphore objects can also be used in a single-process application, but critical section objects provide a slightly faster, more efficient mechanism for mutual-exclusion synchronization. Like a mutex object, a critical section object can be owned by only one thread at a time, which makes it useful for protecting a shared resource from simultaneous access. There is no guarantee about the order in which threads will obtain ownership of the critical section, however, the system will be fair to all threads.

The process is responsible for allocating the memory used by a critical section. Typically, this is done by simply declaring a variable of type **CRITICAL_SECTION**. Before the threads of the process can use it, initialize the critical section by using the **InitializeCriticalSection** or **InitializeCriticalSectionAndSpinCount** function.

A thread uses the **EnterCriticalSection** or **TryEnterCriticalSection** function to request ownership of a critical section. It uses the **LeaveCriticalSection** function to release ownership of a critical section. If the critical section object is currently owned by another thread, **EnterCriticalSection** waits indefinitely for ownership. In contrast, when a mutex object is used for mutual exclusion, the wait functions accept a specified time-out interval. The **TryEnterCriticalSection** function attempts to enter a critical section without blocking the calling thread.

Once a thread owns a critical section, it can make additional calls to **EnterCriticalSection** or **TryEnterCriticalSection** without blocking its execution. This prevents a thread from deadlocking itself while waiting for a critical section that it already owns. To release its ownership, the thread must call **LeaveCriticalSection** once for each time that it entered the critical section.

A thread uses the **InitializeCriticalSectionAndSpinCount** or **SetCriticalSectionSpinCount** function to specify a spin count for the critical section object. On single-processor systems, the spin count is ignored and the critical section spin count is set to 0. On multiprocessor systems, if the critical section is unavailable, the calling thread will spin *dwSpinCount* times before performing a wait operation on a

semaphore associated with the critical section. If the critical section becomes free during the spin operation, the calling thread avoids the wait operation.

Any thread of the process can use the **DeleteCriticalSection** function to release the system resources that were allocated when the critical section object was initialized. After this function has been called, the critical section object can no longer be used for synchronization.

When a critical section object is owned, the only other threads affected are those waiting for ownership in a call to **EnterCriticalSection**. Threads that are not waiting are free to continue running.

Interlocked Variable Access

The interlocked functions provide a simple mechanism for synchronizing access to a variable that is shared by multiple threads. The threads of different processes can use this mechanism if the variable is in shared memory. Note that simple reads and writes to properly-aligned 32-bit variables are atomic. The interlocked functions should be used to perform complex operations in an atomic manner.

The **InterlockedIncrement** and **InterlockedDecrement** functions combine the operations of incrementing or decrementing the variable and checking the resulting value. This atomic operation is useful in a multitasking operating system, in which the system can interrupt one thread's execution to grant a slice of processor time to another thread. Without such synchronization, one thread could increment a variable but be interrupted by the system before it can check the resulting value of the variable. A second thread could then increment the same variable. When the first thread receives its next time slice, it will check the value of the variable, which has now been incremented not once but twice. The interlocked variable-access functions protect against this kind of error.

The **InterlockedExchangePointer** function atomically exchanges the values of the specified variables. The **InterlockedExchangeAdd** function combines two operations: adding two variables together and storing the result in one of the variables.

The **InterlockedCompareExchangePointer** function combines two operations: comparing two values and storing a third value in one of the variables, based on the outcome of the comparison.

CHAPTER 7

Memory Management

About Memory Management

Each process has its own 32-bit virtual address space that enables addressing up to 4 gigabytes (GB) of memory.

Virtual Address Space

The virtual addresses used by a process do not represent the actual physical location of an object in memory. Instead, the system maintains a *page map* for each process, which is an internal data structure used to translate virtual addresses into corresponding physical addresses.

The virtual address space is divided into partitions, as follows:

Windows NT Server Enterprise Edition/Windows 2000 Advanced Server: The 3-GB partition in low memory (0x00000000 through 0xBFFFFFFF) is available to the process, and the 1-GB partition in high memory (0xC0000000 through 0xFFFFFFFF) is reserved for the system.

Windows NT/2000: The 2-GB partition in low memory (0x00000000 through 0x7FFFFFFF) is available to the process, and the 2-GB partition in high memory (0x80000000 through 0xFFFFFFFF) is reserved for the system.

Windows 95/98: The following are the partitions on Windows 95/98:

Range	Usage
0K–~64K (0xFFFF)	Not writable. This boundary is approximate due to the way the Windows 95/98 loads some features of Microsoft® MS-DOS®. This memory is private to the process.
~64K (0x10000)– 4 MB (0x3FFFFF)	Reserved for MS-DOS compatibility. This memory is fully readable and writable by the process. However, this range of memory may have some MS-DOS–related structures or code in it, so processes should not arbitrarily read from or write to it. This memory is private to the process.
4MB (0x400000)– 2GB (0x7FFFFFFF)	Available for code and user data. User data is readable and writable by the process. Code is execute-only. This memory is private to the process.

(continued)

(continued)

Range	Usage
2GB (0x80000000)– 3GB (0xBFFFFFFF)	Shared area, readable and writable by all processes. A number of system DLLs and other data are loaded into this space.
3GB (0xC0000000)– 4GB (0xFFFFFFFF)	System memory, readable or writable by any process. However, this is where low-level system code resides, so writing to this region can corrupt the system, with potentially catastrophic consequences.

Virtual Address Space and Physical Storage

The virtual address space of each process is much larger than the total physical memory available to all processes. To increase the size of physical storage, the system uses the disk for additional storage. The total amount of storage available to all executing processes is the sum of the physical memory and the free space on disk available to the *paging file*, a disk file used to increase the amount of physical storage. Physical storage and the virtual address space of each process are organized into *pages*, units of memory for which size depends on the host computer. For example, on *x*86 computers the host page size is 4 kilobytes (KB).

To maximize its flexibility in managing memory, the system can move pages of physical memory to and from a paging file on disk. When a page is moved in physical memory, the system updates the page maps of the affected processes. When the system needs space in physical memory, it moves the least recently used pages of physical memory to the paging file. Manipulation of physical memory by the system is completely transparent to applications, which operate only in their virtual address spaces.

Page State

The pages of a process's virtual address space can be in one of the following states:

State	Description
Free	A free page is not currently accessible, but it is available to be committed or reserved.
Reserved	A reserved page is a block of the process's virtual address space that has been set aside for future use. The process cannot access a reserved page, and there is no physical storage associated with it. A reserved page reserves a range of virtual addresses that cannot be used subsequently by other allocation functions. A process can use the **VirtualAlloc** function to reserve pages of its address space and later to commit the reserved pages. It can use the **VirtualFree** function to release them.

Committed — A committed page is one for which physical storage (in memory or on disk) has been allocated. It can be protected to allow either no access or read-only access, or it can have read and write access. A process can use the **VirtualAlloc** function to allocate committed pages. The **GlobalAlloc** and **LocalAlloc** functions allocate committed pages with read/write access. A committed page allocated by **VirtualAlloc** can be decommitted by the **VirtualFree** function, which releases the page's storage and changes the state of the page to reserved.

Scope of Allocated Memory

All memory a process allocates by using the Win32 memory allocation functions (**HeapAlloc**, **VirtualAlloc**, **GlobalAlloc**, **LocalAlloc**) is accessible only to the process. However, memory allocated by a DLL is allocated in the address space of the process that called the DLL and is not accessible to other processes using the same DLL. To create shared memory, you must use file mapping.

Named file mapping provides an easy way to create a block of shared memory. A process can specify a name when it uses the **CreateFileMapping** function to create a file-mapping object. Other processes can specify the same name to either the **CreateFileMapping** or **OpenFileMapping** function to obtain a handle to the mapping object.

Each process specifies its handle to the file-mapping object in the **MapViewOfFile** function to map a view of the file into its own address space. The views of all processes for a single file-mapping object are mapped into the same sharable pages of physical storage. However, the virtual addresses of the mapped views can vary from one process to another, unless the **MapViewOfFileEx** function is used to map the view at a specified address. Although sharable, the pages of physical storage used for a mapped file view are not global; they are not accessible to processes that have not mapped a view of the file.

Any pages committed by mapping a view of a file are released when the last process with a view of the mapping object either terminates or unmaps its view by calling the **UnmapViewOfFile** function. At this time, the specified file (if any) associated with the mapping object is updated. A specified file also can be forced to update by calling the **FlushViewOfFile** function.

For more information, see *File Mapping*. For an example of shared memory in a DLL, see *Using Shared Memory in a Dynamic Link Library*.

If multiple processes have write access to shared memory, you must synchronize access to the memory. For more information, see *Synchronization*.

Virtual Memory Functions

The Microsoft Win32 API provides a set of virtual memory functions that enable a process to manipulate or determine the status of pages in its virtual address space. They can perform the following operations:

- Reserve a range of a process's virtual address space. Reserving address space does not allocate any physical storage, but it prevents other allocation operations from using the specified range. It does not affect the virtual address spaces of other processes. Reserving pages prevents needless consumption of physical storage, while enabling a process to reserve a range of its address space into which a dynamic data structure can grow. The process can allocate physical storage for this space, as needed.
- Commit a range of reserved pages in a process's virtual address space, so that physical storage (either in RAM or on disk) is accessible only to the allocating process.
- Specify read/write, read-only, or no access for a range of committed pages. This differs from the standard allocation functions that always allocate pages with read/write access.
- Free a range of reserved pages, making the range of virtual addresses available for subsequent allocation operations by the calling process.
- Decommit a range of committed pages, releasing their physical storage and making it available for subsequent allocation by any process.
- Lock one or more pages of committed memory into physical memory (RAM), so that the system cannot swap the pages out to the paging file.
- Obtain information about a range of pages in the virtual address space of the calling process or a specified process.
- Change the access protection for a specified range of committed pages in the virtual address space of the calling process or a specified process.

Allocating Virtual Memory

The virtual memory functions manipulate pages of memory. The functions use the size of a page on the current computer to round off specified sizes and addresses.

The **VirtualAlloc** function performs one of the following operations:

- Reserves one or more free pages
- Commits one or more reserved pages
- Reserves and commits one or more free pages

You can specify the starting address of the pages to be reserved or committed, or you can allow the system to determine the address. The function rounds the specified address to the appropriate page boundary. Reserved pages are not accessible, but committed pages can be allocated with the PAGE_READWRITE, PAGE_READONLY, or PAGE_NOACCESS flag. When pages are committed, storage is allocated in the paging file, but each page is initialized and loaded into physical memory only at the first attempt to read from or write to that page. You can use normal pointer references to access memory committed by the **VirtualAlloc** function.

Freeing Virtual Memory

The **VirtualFree** function performs one of the following operations:

- Decommits one or more committed pages, changing the state of the pages to reserved. Decommitting pages releases the physical storage associated with the pages, making it available to be allocated by any process. Any block of committed pages can be decommitted.
- Releases a block of one or more reserved pages, changing the state of the pages to free. Releasing a block of pages makes the range of reserved addresses available to be allocated by the process. Reserved pages can be released only by freeing the entire block that was initially reserved by **VirtualAlloc**.
- Decommits and releases a block of one or more committed pages simultaneously, changing the state of the pages to free. The specified block must include the entire block initially reserved by **VirtualAlloc**, and all of the pages must be currently committed.

Once memory is released or decommitted, you can never refer to it again.
Any information that may have been in that memory is gone forever. Attempting to read from or write to a free page results in an access violation exception. If you require information, do not decommit or free memory containing that information.

Working with Pages

To determine the size of a page on the current computer, use the **GetSystemInfo** function.

The **VirtualQuery** and **VirtualQueryEx** functions return information about a region of consecutive pages beginning at a specified address in the address space of a process. **VirtualQuery** returns information about memory in the calling process. **VirtualQueryEx** returns information about memory in a specified process and is used to support debuggers that need information about a process being debugged. The region of pages is bounded by the specified address rounded down to the nearest page boundary. It extends through all subsequent pages with the following attributes in common:

- The state of all pages is the same: either committed, reserved, or free.
- If the initial page is not free, all pages in the region are part of the same initial allocation of pages that were reserved by a call to **VirtualAlloc**.
- The access protection of all pages is the same (that is, the PAGE_READONLY, PAGE_READWRITE, or PAGE_NOACCESS flag).

The **VirtualLock** function enables a process to lock one or more pages of committed memory into physical memory (RAM), preventing the system from swapping the pages out to the paging file. It can be used to ensure that critical data is accessible without disk access. Locking pages into memory is dangerous because it restricts the system's ability to manage memory. Excessive use of **VirtualLock** can degrade system performance by

causing executable code to be swapped out to the paging file. The **VirtualUnlock** function unlocks memory locked by **VirtualLock**.

The **VirtualProtect** function enables a process to modify the access protection of any committed page in the address space of a process. For example, a process can allocate read/write pages to store sensitive data, and then it can change the access to read only or no access to protect against accidental overwriting. **VirtualProtect** is typically used with pages allocated by **VirtualAlloc**, but it also works with pages committed by any of the other allocation functions. However, **VirtualProtect** changes the protection of entire pages, and pointers returned by the other functions are not necessarily aligned on page boundaries. The **VirtualProtectEx** function is similar to **VirtualProtect**, except it changes the protection of memory in a specified process. Changing the protection is useful to debuggers in accessing the memory of a process being debugged.

Heap Functions

The heap functions enable a process to create a private heap, a block of one or more pages in the address space of the calling process. The process can then use a separate set of functions to manage the memory in that heap. There is no difference between memory allocated from a private heap and allocated by using the other memory allocation functions.

The **HeapCreate** function creates a private heap object from which the calling process can allocate memory blocks by using the **HeapAlloc** function. **HeapCreate** specifies both an initial size and a maximum size for the heap. The initial size determines the number of committed, read/write pages initially allocated for the heap. The maximum size determines the total number of reserved pages. These pages create a contiguous block in the virtual address space of a process into which the heap can grow. Additional pages are automatically committed from this reserved space if requests by **HeapAlloc** exceed the current size of committed pages, assuming that the physical storage for it is available. Once the pages are committed, they are not decommitted until the process is terminated or until the heap is destroyed by calling the **HeapDestroy** function.

The memory of a private heap object is accessible only to the process that created it. If a DLL creates a private heap, it does so in the address space of the process that called the DLL. It is accessible only to that process.

The **HeapAlloc** function allocates a specified number of bytes from a private heap and returns a pointer to the allocated block. The pointer identifies the block for the **HeapFree** function to release or for the **HeapSize** function to determine the size.

Memory allocated by **HeapAlloc** is not movable. Because the system cannot compact a private heap, the heap can become fragmented.

A possible use for the heap functions is to create a private heap when a process starts up, specifying an initial size sufficient to satisfy the memory requirements of the process. If the call to the **HeapCreate** function fails, the process can terminate or notify the user of the memory shortage; if it succeeds, however, the process is assured of having the memory it needs.

Memory requested by **HeapCreate** may or may not be contiguous. Memory allocated within a heap by **HeapAlloc** is contiguous. You should not write to or read from memory in a heap except that which is allocated by **HeapAlloc**; neither should you assume any relationship between two areas of memory allocated by **HeapAlloc**.

You should not refer in any way to memory that has been freed by **HeapFree**. Once that memory is freed, any information that may have been in it is gone forever. If you require information, do not free memory containing the information. Function calls that return information about memory (such as **HeapSize**) may not be used with freed memory, as they might return bogus data.

External factors may cause accesses to heap memory to generate access violations. One possible cause of an access violation is very limited space in the paging file. Therefore, all accesses to heap memory should be protected with structured exception handlers. For more information, see *Structured Exception Handling*.

Windows 95/98: The heap managers are designed for memory blocks smaller than 4 megabytes (MB). If you expect your memory blocks to be larger than one or 2 MB, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Access Validation Functions

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

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

The **IsBadHugeReadPtr** and **IsBadHugeWritePtr** functions are also available for compatibility with 16-bit versions of Windows that distinguished between normal memory allocations and huge allocations occupying multiple segments. In 32-bit versions of Windows, these functions are equivalent to **IsBadReadPtr** and **IsBadWritePtr**.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when an access validation function indicates that the process has the desired access to the specified memory, you

should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception. For more information, see *Structured Exception Handling*.

Address Windowing Extensions

This topic describes the Address Windowing Extensions (AWE). These Windows 2000 extensions provide user applications with 32-bit virtual addressing to greater than 32-bit regions of physical memory.

Windows NT and Windows 2000 have always provided applications with a flat 32-bit virtual address space, which describes 4 GB of virtual memory. The address space is usually split so that 2 GB of address space is directly accessible to the application. The other 2 GB are accessible only to the executive software. Additionally, with Windows NT 4.0 SP3, an option is provided on Windows NT Server Enterprise Edition/Windows 2000 Advanced Server *x*86 systems for applications to have a 3-GB flat virtual address space, with the executive software using only 1 GB.

AWE is a set of extensions that allows an application with the Lock Pages in Memory user right to use physical nonpaged memory and window views to various portions of this physical memory within a 32-bit virtual address space. In this way, applications are able to quickly manipulate physical memory greater than 4 GB. Certain data-intensive applications, such as database management systems and scientific and engineering software, need access to very large caches of data. In the case of very large data sets, restricting the cache to fit within an application's 2 GB of user address space is a severe restriction. In these situations, the cache is too small to properly support the application.

AWE solves this problem by allowing applications to directly address huge amounts of memory while continuing to use 32-bit pointers. AWE allows applications to have data caches larger than 4 GB (where sufficient physical memory is present).

AWE places a few restrictions on how this memory may be used, primarily because these restrictions allow extremely fast mapping, remapping, and freeing. Fast memory management is important for these potentially enormous address spaces.

- Virtual address ranges allocated for the AWE are not sharable with other processes (and therefore not inheritable). In fact, two different AWE virtual addresses within the same process are not allowed to map the same physical page. These restrictions provide fast remapping and cleanup when memory is freed.
- The physical pages that can be allocated for an AWE region are limited by the number of physical pages present in the machine, since this memory is never paged—it is locked down until the application explicitly frees it or exits. The physical pages allocated for a given process can be mapped into any AWE virtual region within the same process. Applications that use AWE must be careful not to take so much physical memory that they cause other applications to page excessively, or prevent creation of new processes or threads due to lack of resources. Use the **GlobalMemoryStatusEx** function to monitor physical memory use.

- AWE virtual addresses are always read/write and cannot be protected via calls to **VirtualProtect** (that is, no read-only memory, no access memory, guard pages, or the like can be specified).
- AWE address ranges cannot be used to buffer data for graphics or video calls.
- An AWE memory range cannot be split, and pieces of it cannot be deleted. Instead, the entire virtual address range must be deleted as a unit when deletion is required. This means that the MEM_RELEASE flag (and not the MEM_DECOMMIT flag) must be specified to **VirtualFree**.
- Applications that use AWE are not supported in emulation mode. That is, an *x*86 application that uses AWE functions must be recompiled to run on Windows 2000 on the Alpha processor, whereas most applications can run without recompiling under an emulator on other platforms.

This solution addresses the physical memory issues in a very general, widely applicable manner. Some of the benefits of AWE are:

- A small group of new functions is defined to manipulate AWE memory.
- AWE is supported on all platforms supported by Windows 2000, including Alpha (32-bit) and the 64-bit version of Windows 2000.
- AWE provides a very fast remapping capability. Remapping is done by manipulating virtual memory tables in the kernel, not by moving data in physical memory.
- AWE provides page size granularity appropriate to the processor (for example, 4 KB on *x*86 and 8 KB on Alpha), which is more useful to applications than large pages (for example, 2 MB or 4 MB on *x*86).

To obtain the Lock Pages in Memory privilege, an administrator must add the attribute "Lock Pages in Memory" to the user's User Rights Assignments. For more information on how to do this, see "User Rights" in the Windows 2000 help system.

AWE Functions

Three new functions have been added to allow applications to explicitly control their virtual address space. The following functions make up the AWE API:

Function	Description
VirtualAlloc	Reserve a portion of virtual address space to use for AWE, using the AWE flag MEM_PHYSICAL
AllocateUserPhysicalPages	Allocate physical memory for use with AWE
MapUserPhysicalPages	Map (or invalidate) AWE virtual addresses onto any set of physical pages obtained with **AllocateUserPhysicalPages**
MapUserPhysicalPagesScatter	Map (or invalidate) AWE virtual addresses onto any set of physical pages obtained with **AllocateUserPhysicalPages**, but with finer control than that provided by **MapUserPhysicalPages**
FreeUserPhysicalPages	Free physical memory that was used for AWE

Global and Local Functions

The global and local functions are the 16-bit Windows heap functions. Win32 memory management supports these functions for porting from 16-bit Windows, or maintaining source code compatibility with 16-bit Windows. The global and local functions are slower than the new memory management functions and do not provide as many features. Therefore, new applications should not use these functions.

A process can use the **GlobalAlloc** and **LocalAlloc** functions to allocate memory. Win32 memory management does not provide a separate local heap and global heap, as 16-bit Windows does. As a result, there is no difference between the memory objects allocated by these functions. In addition, the change from a 16-bit segmented memory model to a 32-bit virtual memory model has made some of the related global and local functions and their options unnecessary or meaningless. For example, there are no longer near and far pointers, because both local and global allocations return 32-bit virtual addresses.

Memory objects allocated by **GlobalAlloc** and **LocalAlloc** are in private, committed pages with read/write access that cannot be accessed by other processes. Memory allocated by using **GlobalAlloc** with the GMEM_DDESHARE flag is not actually shared globally as it is in 16-bit Windows. However, this flag is available for compatibility purposes and can be used by some applications to enhance the performance of dynamic data exchange (DDE) operations. Applications requiring shared memory for other purposes must use file-mapping objects. Multiple processes can map a view of the same file-mapping object to provide named shared memory. For more information, see *File Mapping*.

By using **GlobalAlloc** and **LocalAlloc**, you can allocate a block of memory of any size that can be represented by 32 bits. You are limited only by the available physical memory, including storage in the paging file on disk. In 16-bit Windows, when you allocate a fixed memory object, **GlobalAlloc** and **LocalAlloc** return a 32-bit pointer that the calling process can immediately use to access the memory. When you allocate memory using GMEM_MOVEABLE, the return value is a handle. To get a pointer to a movable memory object, use the **GlobalLock** and **LocalLock** functions.

The actual size of the memory allocated by **GlobalAlloc** or **LocalAlloc** can be larger than the requested size. To determine the actual number of bytes allocated, use the **GlobalSize** or **LocalSize** function. If the amount allocated is greater than the amount requested, the process can use the entire amount.

The **GlobalReAlloc** and **LocalReAlloc** functions change the size, in bytes, or the attributes of a memory object allocated by **GlobalAlloc** and **LocalAlloc**. The size can increase or decrease.

The **GlobalFree** and **LocalFree** functions release memory allocated by **GlobalAlloc**, **LocalAlloc**, **GlobalReAlloc**, or **LocalReAlloc**.

Other global and local functions include the **GlobalDiscard**, **LocalDiscard**, **GlobalFlags**, **LocalFlags**, **GlobalHandle**, and **LocalHandle** functions. To discard the specified memory object without invalidating the handle, use **GlobalDiscard** or **LocalDiscard**. The handle can be used later by **GlobalReAlloc** or **LocalReAlloc** to allocate a new block of memory associated with the same handle. To return information about a specified memory object, use **GlobalFlags** or **LocalFlags**. The information includes the object's lock count and indicates whether the object is discardable or has already been discarded. To return a handle to the memory object associated with a specified pointer, use **GlobalHandle** or **LocalHandle**.

Windows 95/98: The heap managers are designed for memory blocks smaller than 4 MB. If you expect your memory blocks to be larger than one or 2 MB, you can avoid significant performance degradation by using the **VirtualAlloc** or **VirtualAllocEx** function instead.

Standard C Library Functions

Win32-based applications can safely use the memory management features of the C run-time library (**malloc**, **free**, and so on) and C++ (**new**, **delete**, and so on). The C run-time library functions do not have the potential problems they have under 16-bit Windows. Memory management is no longer a problem because the system is free to manage memory by moving pages of physical memory without affecting the virtual addresses. Similarly, the distinction between near and far pointers is no longer relevant. Therefore, you can use the standard C library functions for memory management. However, the Win32 memory management functions do provide functionality that is unavailable in the C run-time library.

Memory Management Reference

AllocateUserPhysicalPages

The **AllocateUserPhysicalPages** function allocates physical memory pages to be mapped and unmapped within any AWE virtual address space region of the specified process.

The caller must have Lock Pages in Memory privilege for this call to succeed.

```
BOOL AllocateUserPhysicalPages(
  HANDLE hProcess,           // handle to process
  PULONG_PTR NumberOfPages,  // number of pages
  PULONG_PTR UserPfnArray    // address of storage
);
```

Parameters

hProcess

[in] Handle to a process. The function allocates memory within the virtual address space of this process. The user executing the calling application must have Lock Pages in Memory privileged access to this process. If the user does not, the function fails.

NumberOfPages

[in/out] Specifies the size, in pages, of the physical memory to allocate. Use the **GetSystemInfo** function to determine the page size of the computer. This parameter also returns the number of pages actually allocated, which may be less than the number requested.

UserPfnArray

[out] Specifies the virtual address in which to store the page frame numbers of the allocated memory. The size of the memory allocated should be at least *NumberOfPages* times the size of the data type ULONG_PTR.

Do not attempt to modify this buffer. It contains operating system data, corruption of which could be catastrophic. There is no information in it that is useful to your application.

Return Values

If the function succeeds, the return value is TRUE. Fewer pages than requested may actually be allocated. The caller must check the value of the *NumberOfPages* parameter on return to see how many pages were allocated. All allocated page frame numbers are sequentially placed in the memory pointed to by the *UserPfnArray* parameter.

If the function fails, the return value is FALSE and no frames are allocated. To get extended error information, call **GetLastError**.

Remarks

The **AllocateUserPhysicalPages** function is used to allocate physical memory. Memory allocated by this function must be physically present in the system. After it is allocated, it is locked down and unavailable to the rest of the virtual memory management system of Windows 2000.

Note that a given physical page cannot be simultaneously mapped at more than one virtual address.

Physical pages can reside at any physical address. You should make no assumptions about the contiguity of the physical pages.

Use the **GetSystemInfo** function to determine the page size of the computer.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **Address Windowing Extensions**, **MapUserPhysicalPages**, **MapUserPhysicalPagesScatter**, **FreeUserPhysicalPages**, **GetSystemInfo**

CopyMemory

The **CopyMemory** function copies a block of memory from one location to another.

```
VOID CopyMemory(
  PVOID Destination,      // copy destination
  CONST VOID *Source,     // memory block
  SIZE_T Length           // size of memory block
);
```

Parameters

Destination
[in] Pointer to the starting address of the copied block's destination.

Source
[in] Pointer to the starting address of the block of memory to copy.

Length
[in] Specifies the size, in bytes, of the block of memory to copy.

Return Values

This function has no return value.

Remarks

If the source and destination blocks overlap, the results are undefined. For overlapped blocks, use the **MoveMemory** function.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

! Requirements

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.

See Also

Memory Management Overview, *Memory Management Functions*, **FillMemory**, **MoveMemory**, **ZeroMemory**

FillMemory

The **FillMemory** function fills a block of memory with a specified value.

```
VOID FillMemory (
  PVOID Destination,   // memory block
  SIZE_T Length,       // size of memory block
  BYTE Fill            // fill value
);
```

Parameters

Destination
[out] Pointer to the starting address of the block of memory to fill.

Length
[in] Specifies the size, in bytes, of the block of memory to fill.

Fill
[in] Specifies the byte value with which to fill the memory block.

Return Values

This function has no return value.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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.

See Also

Memory Management Overview, *Memory Management Functions*, **CopyMemory**, **MoveMemory**, **ZeroMemory**

FreeUserPhysicalPages

The **FreeUserPhysicalPages** function frees physical memory pages previously allocated with **AllocateUserPhysicalPages**. If any of these pages is currently mapped in the AWE address space, it is automatically unmapped by this call. Note that this does not affect the virtual address space occupied by the specified AWE region.

```
BOOL FreeUserPhysicalPages(
  HANDLE hProcess,              // handle to process
  PULONG_PTR NumberOfPages,     // pages to free
  PULONG_PTR UserPfnArray       // virtual address
);
```

Parameters

hProcess

[in] Handle to a process. The function frees memory within the virtual address space of this process.

NumberOfPages

[in/out] Specifies the size, in pages, of the physical memory to free. On return, if the function failed, this parameter indicates the number of pages freed.

UserPfnArray

[in] Specifies the virtual address to obtain the page frame numbers to free.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE. In this case, the *NumberOfPages* parameter will reflect how many pages have actually been released. To get extended error information, call **GetLastError**.

Remarks

In a multiprocessor environment, this function maintains coherence of the hardware translation buffer. Upon return from this function, all threads on all processors are guaranteed to see the correct mapping.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **Address Windowing Extensions**, **AllocateUserPhysicalPages**, **MapUserPhysicalPages**, **MapUserPhysicalPagesScatter**

GetProcessHeap

The **GetProcessHeap** function obtains a handle to the heap of the calling process. This handle then can be used in subsequent calls to the **HeapAlloc**, **HeapReAlloc**, **HeapFree**, and **HeapSize** functions.

```
HANDLE GetProcessHeap(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a handle to the calling process's heap.

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

Remarks

The **GetProcessHeap** function allows you to allocate memory from the process heap without having to first create a heap with the **HeapCreate** function, as shown in this example:

```
HeapAlloc(GetProcessHeap(), 0, dwBytes);
```

Note The handle obtained by calling this function should not be used in calls to the **HeapDestroy** function.

To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **HeapAlloc**, **HeapCreate**, **HeapDestroy**, **HeapFree**, **HeapReAlloc**, **HeapSize**

GetProcessHeaps

The **GetProcessHeaps** function obtains handles to all of the heaps that are valid for the calling process.

```
DWORD GetProcessHeaps(
  DWORD NumberOfHeaps,  // maximum number of heap handles
  PHANDLE ProcessHeaps  // buffer for heap handles
);
```

Parameters

NumberOfHeaps
[in] Specifies the maximum number of heap handles that can be stored into the buffer pointed to by *ProcessHeaps*.

ProcessHeaps
[out] Pointer to a buffer to receive an array of heap handles.

Return Values

The return value is the number of heap handles that are valid for the calling process.

If the return value is less than or equal to *NumberOfHeaps*, it is also the number of heap handles stored into the buffer pointed to by *ProcessHeaps*.

If the return value is greater than *NumberOfHeaps*, the buffer pointed to by *ProcessHeaps* is too small to hold all the valid heap handles of the calling process.The function will have stored no handles into that buffer. In this situation, use the return value to allocate a buffer that is large enough to receive the handles, and call the function again.

If the return value is zero, the function has failed, because every process has at least one valid heap, the process heap. To get extended error information, call **GetLastError**.

Remarks

Use the **GetProcessHeaps** function to obtain a handle to the process heap of the calling process. The **GetProcessHeaps** function obtains a handle to that heap, plus handles to any additional private heaps created by calling the **HeapCreate** function.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **HeapCreate**

GetWriteWatch

The **GetWriteWatch** function retrieves the addresses of the pages that have been written to in a region of virtual memory.

```
UINT GetWriteWatch(
  DWORD dwFlags,             // write-tracking state
  PVOID lpBaseAddress,       // base address of region
  SIZE_T dwRegionSize,       // size of region
  PVOID *lpAddresses,        // array of page addresses
  PULONG_PTR lpdwCount,      // number of addresses returned
  PULONG lpdwGranularity     // page size
);
```

Parameters

dwFlags

[in] Indicates whether the function resets the write-tracking state. To reset the write-tracking state, set this parameter to WRITE_WATCH_FLAG_RESET. If this parameter is zero, **GetWriteWatch** does not reset the write-tracking state. For more information, see the following Remarks section.

lpBaseAddress

[in] Specifies the base address of the memory region for which to retrieve write-tracking information. This address must be in a memory region that was allocated by the **VirtualAlloc** function with the MEM_WRITE_WATCH flag.

dwRegionSize

[in] Specifies the size, in bytes, of the memory region for which to retrieve write-tracking information.

lpAddresses

[out] Pointer to a buffer that receives an array of page addresses in the memory region. The addresses indicate the pages that have been written to since the region was allocated or the write-tracking state was reset.

lpdwCount

[in/out] On input, this variable indicates the size, in array elements, of the *lpAddresses* array. On output, the variable receives the number of page addresses returned in the array.

lpdwGranularity
[out] Pointer to a variable that receives the page size, in bytes.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is a nonzero value.

Remarks

When you call the **VirtualAlloc** function to reserve or commit memory, you can specify the MEM_WRITE_WATCH flag. This flag causes the system to keep track of the pages in the committed memory region that have been written to. You then can call the **GetWriteWatch** function to retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset.

To reset the write-tracking state, set the WRITE_WATCH_FLAG_RESET flag in the *dwFlags* parameter. Alternatively, you can call the **ResetWriteWatch** function to reset the write-tracking state. If you use **ResetWriteWatch**, however, you must ensure that no threads write to the region during the interval between the **GetWriteWatch** and **ResetWriteWatch** calls. Otherwise, there may be written pages that you fail to detect.

The **GetWriteWatch** function can be useful to profilers, debugging tools, or garbage collectors.

Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **ResetWriteWatch**, **VirtualAlloc**

GlobalMemoryStatus

The **GlobalMemoryStatus** function obtains information about the computer system's current usage of both physical memory and virtual memory.

```
VOID GlobalMemoryStatus(
  LPMEMORYSTATUS lpBuffer   // memory status structure
);
```

Parameters

lpBuffer

[out] Pointer to a **MEMORYSTATUS** structure. The **GlobalMemoryStatus** function stores information about current memory availability into this structure.

Return Values

This function does not return a value.

Remarks

You can use the **GlobalMemoryStatus** function to determine how much memory your application can allocate without severely impacting other applications.

The information returned by the **GlobalMemoryStatus** function is volatile. There is no guarantee that two sequential calls to this function will return the same information.

On computers with more than 4 GB of memory, the **GlobalMemoryStatus** function can return incorrect information. Windows 2000 reports a value of –1 to indicate an overflow. Earlier versions of Windows NT report a value that is the real amount of memory, modulo 4 GB. For this reason, on Windows 2000, use instead the **GlobalMemoryStatusEx** function.

On Intel *x*86 computers with more than 2 GB and less than 4 GB of memory, the **GlobalMemoryStatus** function will always return 2 GB in the **dwTotalPhys** member of the **MEMORYSTATUS** structure. Similarly, if the total available memory is between 2 GBand 4 GB, the **dwAvailPhys** member of the **MEMORYSTATUS** structure will be rounded down to 2 GB. If the executable is linked using the /LARGEADDRESSWARE linker option, then the **GlobalMemoryStatus** function will return the correct amount of physical memory in both members. The /LARGEADDRESSWARE linker option is not available (or necessary) on the Alpha processor.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **GlobalMemoryStatusEx**, **MEMORYSTATUS**

HeapAlloc

The **HeapAlloc** function allocates a block of memory from a heap. The allocated memory is not movable.

```
LPVOID HeapAlloc(
  HANDLE hHeap,    // handle to private heap block
  DWORD dwFlags,   // heap allocation control
  SIZE_T dwBytes   // number of bytes to allocate
);
```

Parameters

hHeap

[in] Specifies the heap from which the memory will be allocated. This parameter is a handle returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of heap allocation. Specifying any of these flags will override the corresponding flag specified when the heap was created with **HeapCreate**. You can specify one or more of the following flags:

Flag	Meaning
HEAP_GENERATE_EXCEPTIONS	Specifies that the system will raise an exception to indicate a function failure, such as an out-of-memory condition, instead of returning NULL.
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while the **HeapAlloc** function is accessing the heap. This flag should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.
HEAP_ZERO_MEMORY	Specifies that the allocated memory will be initialized to zero. Otherwise, the memory is not initialized to zero.

dwBytes

[in] Specifies the number of bytes to be allocated.

If the heap specified by the *hHeap* parameter is a "non-growable" heap, dwBytes must be less than 0x7FFF8. You create a non-growable heap by calling the **HeapCreate** function with a nonzero value.

Return Values

If the function succeeds, the return value is a pointer to the allocated memory block.

If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.

If the function fails and you have specified HEAP_GENERATE_EXCEPTIONS, the function may generate the following exceptions:

Value	Meaning
STATUS_NO_MEMORY	The allocation attempt failed because of a lack of available memory or heap corruption.
STATUS_ACCESS_VIOLATION	The allocation attempt failed because of heap corruption or improper function parameters.

Note Heap corruption can lead to either exception; depends upon the nature of the heap corruption.

If the function fails, it does not call **SetLastError**. An application cannot call **GetLastError** for extended error information.

Remarks

If **HeapAlloc** succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual size of the allocated block, use the **HeapSize** function.

To free a block of memory allocated by **HeapAlloc**, use the **HeapFree** function.

Memory allocated by **HeapAlloc** is not movable. Since the memory is not movable, it is possible for the heap to become fragmented.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be used safely only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using instead the **VirtualAlloc** or **VirtualAllocEx** 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **HeapCreate**, **HeapDestroy**, **HeapFree**, **HeapReAlloc**, **HeapSize**, **SetLastError**

HeapCompact

The **HeapCompact** function attempts to compact a specified heap. It compacts the heap by coalescing adjacent free blocks of memory and decommitting large free blocks of memory.

```
UINT HeapCompact(
  HANDLE hHeap,   // handle to heap
  DWORD dwFlags   // heap access options
);
```

Parameters

hHeap
[in] Handle to the heap that the function will attempt to compact.

dwFlags
[in] A set of bit flags that control heap access during function operation. The following bit flag has meaning:

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while the **HeapCompact** function accesses the heap.

Return Values

If the function succeeds, the return value is the size, in bytes, of the largest committed free block in the heap. This is an unsigned integer value.

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

In the unlikely case that there is absolutely no space available in the heap, the function return value is zero, and **GetLastError** returns the value NO_ERROR.

Remarks

There is no guarantee that an application can successfully allocate a memory block of the size returned by **HeapCompact**. Other threads or the commit threshold might prevent such an allocation.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **HeapCreate**, **HeapValidate**

HeapCreate

The **HeapCreate** function creates a heap object that can be used by the calling process. The function reserves space in the virtual address space of the process and allocates physical storage for a specified initial portion of this block.

```
HANDLE HeapCreate(
  DWORD flOptions,        // heap allocation attributes
  SIZE_T dwInitialSize,   // initial heap size
  SIZE_T dwMaximumSize    // maximum heap size
);
```

Parameters

flOptions

[in] Specifies optional attributes for the new heap. These options affect subsequent access to the new heap through calls to the heap functions (**HeapAlloc**, **HeapFree**, **HeapReAlloc**, and **HeapSize**). You can specify one or more of the following values:

Value	Meaning
HEAP_GENERATE_EXCEPTIONS	Specifies that the system will raise an exception to indicate a function failure, such as an out-of-memory condition, instead of returning NULL.
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used when the heap functions allocate and free memory from this heap. The default, when the HEAP_NO_SERIALIZE flag is not specified, is to serialize access to the heap. Serialization of heap access allows two or more threads to simultaneously allocate and free memory from the same heap.

dwInitialSize

[in] Specifies the initial size, in bytes, of the heap. This value determines the initial amount of physical storage that is allocated for the heap. The value is rounded up to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

dwMaximumSize

[in] If *dwMaximumSize* is a nonzero value, it specifies the maximum size, in bytes, of the heap. The **HeapCreate** function rounds *dwMaximumSize* up to the next page

boundary, and then reserves a block of that size in the process's virtual address space for the heap. If allocation requests made by the **HeapAlloc** or **HeapReAlloc** functions exceed the initial amount of physical storage specified by *dwInitialSize*, the system allocates additional pages of physical storage for the heap, up to the heap's maximum size.

In addition, if *dwMaximumSize* is nonzero, the heap cannot grow, and an absolute limitation arises: the maximum size of a memory block in the heap is a bit less than 0x7FFF8 bytes. Requests to allocate larger blocks will fail, even if the maximum size of the heap is large enough to contain the block.

If *dwMaximumSize* is zero, it specifies that the heap is growable. The heap's size is limited only by available memory. Requests to allocate blocks larger than 0x7FFF8 bytes do not automatically fail; the system calls **VirtualAlloc** to obtain the memory needed for such large blocks. Applications that need to allocate large memory blocks should set *dwMaximumSize* to zero.

Return Values

If the function succeeds, the return value is a handle to the newly created heap.

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

Remarks

The **HeapCreate** function creates a private heap object from which the calling process can allocate memory blocks by using the **HeapAlloc** function. The initial size determines the number of committed pages that are initially allocated for the heap. The maximum size determines the total number of reserved pages. These pages create a block in the process's virtual address space into which the heap can grow. If requests by **HeapAlloc** exceed the current size of committed pages, additional pages are automatically committed from this reserved space, assuming that the physical storage is available.

The memory of a private heap object is accessible only to the process that created it. If a DLL creates a private heap, the heap is created in the address space of the process that called the DLL, and it is accessible only to that process.

The system uses memory from the private heap to store heap support structures, so not all of the specified heap size is available to the process. For example, if the **HeapAlloc** function requests 64 KB from a heap with a maximum size of 64 KB, the request may fail because of system overhead.

If the HEAP_NO_SERIALIZE flag is not specified (the simple default), the heap will serialize access within the calling process. Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap.

Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt

to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **GetProcessHeaps**, **GetSystemInfo**, **HeapAlloc**, **HeapDestroy**, **HeapFree**, **HeapReAlloc**, **HeapSize**, **HeapValidate**, **VirtualAlloc**

HeapDestroy

The **HeapDestroy** function destroys the specified heap object. **HeapDestroy** decommits and releases all the pages of a private heap object, and invalidates the handle to the heap.

```
BOOL HeapDestroy(
  HANDLE hHeap   // handle to heap
);
```

Parameters

hHeap
[in] Specifies the heap to be destroyed. This parameter should be a heap handle returned by the **HeapCreate** function. Do not use the handle to the process heap returned by the **GetProcessHeap** 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

Processes can call **HeapDestroy** without first calling the **HeapFree** function to free memory allocated from the heap.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **HeapAlloc**, **HeapCreate**, **HeapFree**, **HeapReAlloc**, **HeapSize**

HeapFree

The **HeapFree** function frees a memory block allocated from a heap by the **HeapAlloc** or **HeapReAlloc** function.

```
BOOL HeapFree(
  HANDLE hHeap,  // handle to heap
  DWORD dwFlags, // heap free options
  LPVOID lpMem   // pointer to memory
);
```

Parameters

hHeap

[in] Specifies the heap whose memory block the function frees. This parameter is a handle returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of freeing a memory block. Only one flag is currently defined; however, all other flag values are reserved for future use. Specifying this value overrides the corresponding value specified in the *flOptions* parameter when the heap was created by using the **HeapCreate** function:

Flag	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while **HeapFree** is accessing the heap. This flag should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.

lpMem

[in] Pointer to the memory block to free. This pointer is returned by the **HeapAlloc** or **HeapReAlloc** function.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero. An application can call **GetLastError** for extended error information.

Remarks

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

You should not refer in any way to memory that has been freed by **HeapFree**. After that memory is freed, any information that might have been in it is gone forever. If you require information, do not free memory containing the information. Function calls that return information about memory (such as **HeapSize**) may not be used with freed memory, as they can return bogus data.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **HeapAlloc**, **HeapCreate**, **HeapDestroy**, **HeapReAlloc**, **HeapSize**, **SetLastError**

HeapLock

The **HeapLock** function attempts to acquire the critical section object, or lock, that is associated with a specified heap.

If the function succeeds, the calling thread owns the heap lock. Only the calling thread will be able to allocate or release memory from the heap. The execution of any other thread of the calling process will be blocked if that thread attempts to allocate or release memory from the heap. Such threads will remain blocked until the thread that owns the heap lock calls the **HeapUnlock** function.

```
BOOL HeapLock(
  HANDLE hHeap   // handle to heap
);
```

Parameters

hHeap
[in] Handle to the heap to lock for exclusive access by the calling thread.

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 **HeapLock** function is primarily useful for preventing the allocation and release of heap memory by other threads while the calling thread uses the **HeapWalk** function.

Each call to **HeapLock** must be matched by a corresponding call to the **HeapUnlock** function. Failure to call **HeapUnlock** will block the execution of any other threads of the calling process that attempt to access the heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **HeapUnlock**, **HeapWalk**

HeapReAlloc

The **HeapReAlloc** function reallocates a block of memory from a heap. This function enables you to resize a memory block and change other memory block properties. The allocated memory is not movable.

```
LPVOID HeapReAlloc(
  HANDLE hHeap,    // handle to heap block
  DWORD dwFlags,   // heap reallocation options
  LPVOID lpMem,    // pointer to memory to reallocate
  SIZE_T dwBytes   // number of bytes to reallocate
);
```

Parameters

hHeap

[in] Heap from which the memory will be reallocated. This is a handle returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of heap reallocation. Specifying any of these flags overrides the corresponding flag specified in the *flOptions* parameter when the heap was created by using the **HeapCreate** function. You can specify one or more of the following flags:

Flag	Meaning
HEAP_GENERATE_EXCEPTIONS	Specifies that the operating system raises an exception to indic a function failure, such as an out-of-memory condition, instead returning NULL.
HEAP_NO_SERIALIZE	Specifies that mutual exclusion is not used while **HeapReAlloc** accessing the heap. This flag should not be specified when accessing the process h The system may create additional threads within the applicatio process, such as a CTRL+C handler, that simultaneously acce the process heap.
HEAP_REALLOC_IN_PLACE_ONLY	Specifies that there can be no movement when reallocating a memory block to a larger size. If this flag is not specified and th reallocation request is for a larger size, the function may move block to a new location. If this flag is specified and the block ca be enlarged without moving, the function fails, leaving the origi memory block unchanged.
HEAP_ZERO_MEMORY	If the reallocation request is for a larger size, this flag specifies the additional region of memory beyond the original size be initialized to zero. The contents of the memory block up to its original size are unaffected.

lpMem

[in] Pointer to the block of memory that the function reallocates. This pointer is returned by an earlier call to the **HeapAlloc** or **HeapReAlloc** function.

dwBytes

[in] New size of the memory block, in bytes. A memory block's size can be increased or decreased by using this function.

If the heap specified by the *hHeap* parameter is a "non-growable" heap, *dwBytes* must be less than 0x7FFF8. You create a non-growable heap by calling the **HeapCreate** function with a nonzero value.

Return Values

If the function succeeds, the return value is a pointer to the reallocated memory block.

If the function fails and you have not specified HEAP_GENERATE_EXCEPTIONS, the return value is NULL.

If the function fails and you have specified HEAP_GENERATE_EXCEPTIONS, the function may generate the following exceptions:

Value	Meaning
STATUS_NO_MEMORY	The reallocation attempt failed for lack of available memory.
STATUS_ACCESS_VIOLATION	The reallocation attempt failed because of heap corruption or improper function parameters.

If the function fails, it calls **SetLastError**. An application can call **GetLastError** for extended error information.

Remarks

If **HeapReAlloc** succeeds, it allocates at least the amount of memory requested. If the actual amount allocated is greater than the amount requested, the process can use the entire amount. To determine the actual size of the reallocated block, use the **HeapSize** function.

If **HeapReAlloc** fails, the original memory is not freed, and the original handle and pointer are still valid.

To free a block of memory allocated by **HeapReAlloc**, use the **HeapFree** function.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Windows 95/98: The heap managers are designed for memory blocks smaller than four megabytes. If you expect your memory blocks to be larger than one or two megabytes, you can avoid significant performance degradation by using instead the **VirtualAlloc** or **VirtualAllocEx** function.

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **HeapAlloc**, **HeapCreate**, **HeapDestroy**, **HeapFree**, **HeapSize**, **SetLastError**

HeapSize

The **HeapSize** function returns the size, in bytes, of a memory block allocated from a heap by the **HeapAlloc** or **HeapReAlloc** function.

```
DWORD HeapSize(
  HANDLE hHeap,   // handle to heap
  DWORD dwFlags,  // heap size options
  LPCVOID lpMem   // pointer to memory
);
```

Parameters

hHeap

[in] Specifies the heap in which the memory block resides. This handle is returned by the **HeapCreate** or **GetProcessHeap** function.

dwFlags

[in] Specifies several controllable aspects of accessing the memory block. Only one flag is defined currently; however, all other flag values are reserved for future use. Specifying this flag will override the corresponding flag specified in the *flOptions* parameter when the heap was created by using the **HeapCreate** function:

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion will not be used while **HeapSize** is accessing the heap. This flag should not be specified when accessing the process heap. The system may create additional threads within the application's process, such as a CTRL+C handler, that simultaneously access the process heap.

lpMem
[in] Pointer to the memory block whose size the function will obtain. This is a pointer returned by the **HeapAlloc** or **HeapReAlloc** function.

Return Values

If the function succeeds, the return value is the size, in bytes, of the allocated memory block.

If the function fails, the return value is –1. The function does not call **SetLastError**. An application cannot call **GetLastError** for extended error information.

Remarks

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **GetProcessHeap**, **HeapAlloc**, **HeapCreate**, **HeapDestroy**, **HeapFree**, **HeapReAlloc**, **SetLastError**

HeapUnlock

The **HeapUnlock** function releases ownership of the critical section object, or lock, that is associated with a specified heap. The **HeapUnlock** function reverses the action of the **HeapLock** function.

```
BOOL HeapUnlock(
  HANDLE hHeap   // handle to heap
);
```

Parameters

hHeap
[in] Handle to the heap to unlock.

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 **HeapLock** function is primarily useful for preventing the allocation and release of heap memory by other threads while the calling thread uses the **HeapWalk** function. The **HeapUnlock** function is the inverse of **HeapLock**.

Each call to **HeapLock** must be matched by a corresponding call to the **HeapUnlock** function. Failure to call **HeapUnlock** will block the execution of any other threads of the calling process that attempt to access the heap.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **HeapLock**, **HeapWalk**

HeapValidate

The **HeapValidate** function attempts to validate a specified heap. The function scans all the memory blocks in the heap, and verifies that the heap control structures maintained by the heap manager are in a consistent state. You can also use the **HeapValidate** function to validate a single memory block within a specified heap, without checking the validity of the entire heap.

```
BOOL HeapValidate(
  HANDLE hHeap,   // handle to heap
  DWORD dwFlags,  // heap access options
  LPCVOID lpMem   // optional pointer to memory block
);
```

Parameters

hHeap

[in] Handle to the heap of interest. The **HeapValidate** function attempts to validate this heap, or a single memory block within this heap.

dwFlags

[in] A set of bit flags that control heap access during function operation. This parameter can be the following value:

Value	Meaning
HEAP_NO_SERIALIZE	Specifies that mutual exclusion is not used while the **HeapValidate** function accesses the heap.

lpMem

[in] Pointer to a memory block within the specified heap. This parameter may be NULL.

If this parameter is NULL, the function attempts to validate the entire heap specified by *hHeap*.

If this parameter is not NULL, the function attempts to validate the memory block pointed to by *lpMem*; it does not attempt to validate the rest of the heap.

Return Values

If the specified heap or memory block is valid, the return value is nonzero.

If the specified heap or memory block is invalid, the return value is zero. On a system set up for debugging, the **HeapValidate** function then displays debugging messages that describe the part of the heap or memory block that is invalid, and stops at a hard-coded breakpoint, so that you can examine the system to determine the source of the invalidity. The **HeapValidate** function does not set the thread's last error value. There is no extended error information for this function; do not call **GetLastError**.

Remarks

There are heap control structures for each memory block in a heap, and for the heap as a whole. When you use the **HeapValidate** function to validate a complete heap, it checks all of these control structures for consistency.

When you use **HeapValidate** to validate a single memory block within a heap, it checks only the control structures pertaining to that element. **HeapValidate** can only validate allocated memory blocks. Calling **HeapValidate** on a freed memory block will return FALSE because there are no control structures to validate.

If you want to validate the heap elements enumerated by the **HeapWalk** function, you should only call **HeapValidate** on the elements that have the PROCESS_HEAP_ENTRY_BUSY bit flag in the **wFlags** member of the **PROCESS_HEAP_ENTRY** structure. **HeapValidate** returns FALSE for all heap elements that do not have this bit set.

Serialization ensures mutual exclusion when two or more threads attempt to simultaneously allocate or free blocks from the same heap. There is a small performance cost to serialization, but it must be used whenever multiple threads allocate and free memory from the same heap. Setting the HEAP_NO_SERIALIZE flag eliminates mutual exclusion on the heap. Without serialization, two or more threads that use the same heap handle might attempt to allocate or free memory simultaneously, likely causing corruption in the heap. The HEAP_NO_SERIALIZE flag, therefore, can be safely used only in the following situations:

- The process has only one thread.
- The process has multiple threads, but only one thread calls the heap functions for a specific heap.
- The process has multiple threads, and the application provides its own mechanism for mutual exclusion to a specific heap.

Validating a heap can degrade performance, especially on symmetric multiprocessing (SMP) computers. The side effects can last until the process ends.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **HeapCreate**, **HeapWalk**, **PROCESS_HEAP_ENTRY**

HeapWalk

The **HeapWalk** function enumerates the memory blocks in a specified heap.

```
BOOL HeapWalk(
  HANDLE hHeap,                 // heap to enumerate
  LPPROCESS_HEAP_ENTRY lpEntry  // state information
);
```

Parameters

hHeap

[in] Handle to the heap whose memory blocks you wish to enumerate.

lpEntry

[in/out] Pointer to a **PROCESS_HEAP_ENTRY** structure that maintains state information for a particular heap enumeration.

If the **HeapWalk** function succeeds, returning the value TRUE, this structure's members contain information about the next memory block in the heap.

To initiate a heap enumeration, set the **lpData** field of the **PROCESS_HEAP_ENTRY** structure to NULL. To continue a particular heap enumeration, call the **HeapWalk** function repeatedly, with no changes to *hHeap*, *lpEntry*, or any of the members of the **PROCESS_HEAP_ENTRY** structure.

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**.

If the heap enumeration terminates successfully by reaching the end of the heap, the function returns FALSE, and **GetLastError** returns the error code ERROR_NO_MORE_ITEMS.

Remarks

To initiate a heap enumeration, call **HeapWalk** with the **lpData** field of the **PROCESS_HEAP_ENTRY** structure pointed to by *lpEntry* set to NULL.

To continue a heap enumeration, call **HeapWalk** with the same *hHeap* and *lpEntry* values, and with the **PROCESS_HEAP_ENTRY** structure unchanged from the preceding call to **HeapWalk**. Repeat this process until you have no need for further enumeration, or

until the function returns FALSE and **GetLastError** returns ERROR_NO_MORE_ITEMS, indicating that all of the heap's memory blocks have been enumerated.

No special call of **HeapWalk** is needed to terminate the heap enumeration, since no enumeration state data is maintained outside the contents of the **PROCESS_HEAP_ENTRY** structure.

HeapWalk can fail in a multithreaded application if the heap is not locked during the heap enumeration. Use the **HeapLock** and **HeapUnlock** functions to control heap locking during heap enumeration.

Walking a heap can degrade performance, especially on symmetric multiprocessing (SMP) computers. The side effects can last until the process ends.

Note To guard against an access violation, use structured exception handling to protect any code that writes to or reads from a heap. For more information on structured exception handling with memory accesses, see *Reading and Writing* and *Structured Exception Handling*.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **HeapLock**, **HeapUnlock**, **HeapValidate**, **PROCESS_HEAP_ENTRY**

IsBadCodePtr

The **IsBadCodePtr** function determines whether the calling process has read access to the memory at the specified address.

```
BOOL IsBadCodePtr(
  FARPROC lpfn   // memory address
);
```

Parameters

lpfn

[in] Pointer to an address in memory.

Return Values

If the calling process has read access to the specified memory, the return value is zero.

If the calling process does not have read access to the specified memory, the return value is nonzero. To get extended error information, call **GetLastError**.

If the application is compiled as a debugging version, and the process does not have read access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

IsBadCodePtr checks the read access only at the specified address and does not guarantee read access to a range of memory.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **IsBadReadPtr**, **IsBadStringPtr**, **IsBadWritePtr**

IsBadReadPtr

The **IsBadReadPtr** function verifies that the calling process has read access to the specified range of memory.

```
BOOL IsBadReadPtr(
  CONST VOID *lp,   // memory address
  UINT_PTR ucb      // size of block
);
```

Parameters

lp

[in] Pointer to the first byte of the memory block.

ucb

[in] Specifies the size, in bytes, of the memory block. If this parameter is zero, the return value is zero.

Return Values

If the calling process has read access to all bytes in the specified memory range, the return value is zero.

If the calling process does not have read access to all bytes in the specified memory range, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

If the calling process has read access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **IsBadCodePtr**, **IsBadStringPtr**, **IsBadWritePtr**

IsBadStringPtr

The **IsBadStringPtr** function verifies that the calling process has read access to a range of memory pointed to by a string pointer.

```
BOOL IsBadStringPtr(
  LPCTSTR lpsz,       // string
  UINT_PTR ucchMax    // maximum size of string
);
```

Parameters

lpsz
[in] Pointer to a null-terminated string, either Unicode or ASCII.

ucchMax
[in] Specifies the maximum size, in characters, of the string. The function checks for read access in all bytes up to the string's terminating null character or up to the number of bytes specified by this parameter, whichever is smaller. If this parameter is zero, the return value is zero.

Return Values

If the calling process has read access to all bytes up to the string's terminating null character or up to the number of bytes specified by *ucchMax*, the return value is zero.

If the calling process does not have read access to all bytes up to the string's terminating null character or up to the number of bytes specified by *ucchMax*, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have read access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

If the calling process has read access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has read access to the specified memory, you should use structured exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

Requirements

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.

Memory Management Overview, *Memory Management Functions*, **IsBadCodePtr**, **IsBadReadPtr**, **IsBadWritePtr**

IsBadWritePtr

The **IsBadWritePtr** function verifies that the calling process has write access to the specified range of memory.

```
BOOL IsBadWritePtr(
  LPVOID lp,      // memory address
  UINT_PTR ucb    // size of memory block
);
```

Parameters

lp
[in] Pointer to the first byte of the memory block.

ucb
[in] Specifies the size, in bytes, of the memory block. If this parameter is zero, the return value is zero.

Return Values

If the calling process has write access to all bytes in the specified memory range, the return value is zero.

If the calling process does not have write access to all bytes in the specified memory range, the return value is nonzero.

If the application is compiled as a debugging version, and the process does not have write access to all bytes in the specified memory range, the function causes an assertion and breaks into the debugger. Leaving the debugger, the function continues as usual, and returns a nonzero value. This behavior is by design, as a debugging aid.

Remarks

If the calling process has write access to some, but not all, of the bytes in the specified memory range, the return value is nonzero.

In a preemptive multitasking environment, it is possible for some other thread to change the process's access to the memory being tested. Even when the function indicates that the process has write access to the specified memory, you should use structured

exception handling when attempting to access the memory. Use of structured exception handling enables the system to notify the process if an access violation exception occurs, giving the process an opportunity to handle the exception.

IsBadWritePtr is not multithread safe. To use it properly on a pointer shared by multiple threads, call it inside a critical region of code that allows only one thread to access the memory being checked. Use operating system-level objects, such as critical sections or mutexes or the interlocked functions, to create the critical region of code.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **IsBadCodePtr**, **IsBadHugeReadPtr**, **IsBadHugeWritePtr**, **IsBadReadPtr**

MapUserPhysicalPages

The **MapUserPhysicalPages** function maps previously allocated physical memory pages at the specified address within an Address Windowing Extensions (AWE) virtual address space.

```
BOOL MapUserPhysicalPages(
  PVOID lpAddress,           // starting address of region
  ULONG_PTR NumberOfPages,   // size of physical memory
  PULONG_PTR UserPfnArray    // array of page frame numbers
);
```

Parameters

lpAddress
[in] Pointer to the starting address of the region of memory to remap. The value of *lpAddress* must be within the address range returned by the **VirtualAlloc** function when the AWE region was allocated.

NumberOfPages
[in] Specifies the size, in pages, of the physical memory (and virtual address space) for which to establish translations. The virtual address range is contiguous starting at *lpAddress*. The physical frames are specified by the *UserPfnArray*. The total number of pages cannot extend from the starting address beyond the end of the range specified in **AllocateUserPhysicalPages**.

UserPfnArray

[in] Specifies the address of an array of physical page frame numbers. These frames will be mapped by the argument *lpAddress* upon return from this function. The size of the memory allocated should be at least *NumberOfPages* times the size of the data type **ULONG_PTR**.

Do not attempt to modify this buffer. It contains operating system data, corruption of which could be catastrophic. There is no information in it that is useful to your application.

Specifying NULL for this parameter results in the specified address range being unmapped. (The specified physical pages are not freed. You must call **FreeUserPhysicalPages** to free them.)

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE and no mapping (partial or other) will have been done. To get extended error information, call **GetLastError**.

Remarks

Any number of physical memory pages can be specified, provided the memory will not extend outside the virtual address space allocated by **VirtualAlloc**. All existing address maps are automatically overwritten with the new translations, and the old translations are unmapped.

You cannot map physical memory pages outside the range specified in **AllocateUserPhysicalPages**.

Note The physical pages are unmapped but they are not freed. You must call **FreeUserPhysicalPages** to free the physical pages.

Physical pages can reside at any physical address. You should make no assumptions about the contiguity of the physical pages.

To simply unmap the current address range, specify NULL as the physical memory page array parameter. Any currently mapped pages are unmapped, but are not freed. You must call **FreeUserPhysicalPages** to free the physical pages.

In a multiprocessor environment, this function maintains hardware translation buffer coherence. Upon return from this function, all threads on all processors are guaranteed to see the correct mapping.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **Address Windowing Extensions**, **AllocateUserPhysicalPages**, **FreeUserPhysicalPages**, **MapUserPhysicalPagesScatter**

MapUserPhysicalPagesScatter

The **MapUserPhysicalPagesScatter** function maps previously allocated physical memory pages at the specified address within an Address Windowing Extensions (AWE) virtual address space. Unlike **MapUserPhysicalPages**, **MapUserPhysicalPagesScatter** allows "batch" mapping and unmapping of multiple regions.

```
BOOL MapUserPhysicalPagesScatter(
  PVOID *VirtualAddresses,   // starting address of region
  ULONG_PTR NumberOfPages,   // size of physical memory
  PULONG_PTR PageArray       // array of values
);
```

Parameters

VirtualAddresses
[in] Pointer to an array of starting addresses of the regions of memory to remap. Each entry in *VirtualAddresses* must be within the address range that the **VirtualAlloc** function returned when the AWE region was allocated. The value in *NumberOfPages* indicates the size of the array. Note that entries can be from multiple AWE regions.

NumberOfPages
[in] Specifies the size, in pages, of the physical memory (and virtual address space) to establish translations for. The array at *VirtualAddresses* specifies the virtual address range.

PageArray
[in] A non-NULL pointer to an array of values.

If this parameter is NULL, then every address in the *VirtualAddresses* array will be unmapped.

Otherwise, the array is used to indicate how each corresponding page in *VirtualAddresses* should be treated. A zero indicates that the corresponding entry in *VirtualAddresses* should be unmapped; any nonzero value that it has should be mapped.

The value in *NumberOfPages* indicates the size of the array.

Return Values

If the function succeeds, the return value is TRUE.

If the function fails, the return value is FALSE and the function does no mapping or unmapping (partial or other). To get extended error information, call **GetLastError**.

Remarks

You can specify any number of physical memory pages, provided the memory would not extend outside the virtual address space allocated by **VirtualAlloc**. All existing address maps are automatically overwritten with the new translations, and the old translations are unmapped.

You cannot map physical memory pages outside the range specified in **AllocateUserPhysicalPages**.

Note The physical pages may be unmapped but they are not freed. You must call **FreeUserPhysicalPages** to free the physical pages.

Physical pages can reside at any physical address. You should make no assumptions about the contiguity of the physical pages.

In a multiprocessor environment, this function maintains coherence of the hardware translation buffer. Upon return from this function, all threads on all processors are guaranteed to see the correct mapping.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

Memory Management Overview, *Memory Management Functions*, **Address Windowing Extensions**, **AllocateUserPhysicalPages**, **FreeUserPhysicalPages**, **MapUserPhysicalPages**

MoveMemory

The **MoveMemory** function moves a block of memory from one location to another.

```
VOID MoveMemory (
  PVOID Destination,      // move destination
  CONST VOID *Source,     // block to move
```

```
  SIZE_T Length         // size of block to move
);
```

Parameters

Destination
  [in] Pointer to the starting address of the destination of the move.

Source
  [in] Pointer to the starting address of the block of memory to move.

Length
  [in] Specifies the size, in bytes, of the block of memory to move.

Return Values

This function has no return value.

Remarks

The source and destination blocks may overlap.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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.

See Also

Memory Management Overview, *Memory Management Functions*, **CopyMemory**, **FillMemory**, **ZeroMemory**

ResetWriteWatch

The **ResetWriteWatch** function resets the write-tracking state for a region of virtual memory. Subsequent calls to the **GetWriteWatch** function will report only pages that have been written to since the reset operation.

```
UINT ResetWriteWatch(
  LPVOID lpBaseAddress,     // base address
  SIZE_T dwRegionSize       // size of memory region
);
```

Parameters

lpBaseAddress

[in] Pointer to the base address of the memory region for which to reset the write-tracking state. This address must be in a memory region that was allocated by the **VirtualAlloc** function with the MEM_WRITE_WATCH flag.

dwRegionSize

[in] Specifies the size, in bytes, of the memory region for which to reset the write-tracking information.

Return Values

If the function succeeds, the return value is zero.

If the function fails, the return value is a nonzero value.

Remarks

The **ResetWriteWatch** function can be useful to an application such as a garbage collector. The application calls the **GetWriteWatch** function to retrieve the list of written pages, and then writes to those pages as part of its cleanup operation. Then, the garbage collector calls **ResetWriteWatch** to remove the write-tracking records caused by its cleanup.

You can also reset the write-tracking state of a memory region by specifying the WRITE_WATCH_FLAG_RESET flag when you call **GetWriteWatch**.

If you use **ResetWriteWatch**, you must ensure that no threads write to the region during the interval between the **GetWriteWatch** and **ResetWriteWatch** calls. Otherwise, there might be written pages that you fail to detect.

Requirements

Windows NT/2000: Unsupported.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **GetWriteWatch**, **VirtualAlloc**

VirtualAlloc

The **VirtualAlloc** function reserves or commits a region of pages in the virtual address space of the calling process. Memory allocated by this function is automatically initialized to zero, unless the MEM_RESET flag is set.

To allocate memory in the address space of another process, use the **VirtualAllocEx** function.

```
LPVOID VirtualAlloc(
  LPVOID lpAddress,        // region to reserve or commit
  SIZE_T dwSize,           // size of region
  DWORD flAllocationType,  // type of allocation
  DWORD flProtect          // type of access protection
);
```

Parameters

lpAddress

[in] Specifies the desired starting address of the region to allocate. If the memory is being reserved, the specified address is rounded down to the next 64-kilobyte boundary. If the memory is already reserved and is being committed, the address is rounded down to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function. If this parameter is NULL, the system determines where to allocate the region.

dwSize

[in] Specifies the size, in bytes, of the region. If the *lpAddress* parameter is NULL, this value is rounded up to the next page boundary. Otherwise, the allocated pages include all pages containing one or more bytes in the range from *lpAddress* to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be included in the allocated region.

flAllocationType

[in] Specifies the type of allocation. This parameter can be any combination of the following values:

Value	Meaning
MEM_COMMIT	Allocates physical storage in memory or in the paging file on disk for the specified region of pages. An attempt to commit an already committed page will not cause the function to fail. This means that a range of committed or decommitted pages can be committed without having to worry about a failure.
MEM_PHYSICAL	Allocate physical memory. This flag is solely for use with Address Windowing Extensions (AWE) memory.

(continued)

(continued)

Value	Meaning
MEM_RESERVE	Reserves a range of the process's virtual address space without allocating any physical storage. The reserved range cannot be used by any other allocation operations (the **malloc** function, the **LocalAlloc** function, and so on) until it is released. Reserved pages can be committed in subsequent calls to the **VirtualAlloc** function.
MEM_RESET	**Windows NT/2000:** Specifies that memory pages within the range specified by *lpAddress* and *dwSize* will not be written to or read from the paging file. When you set the MEM_RESET flag, you are declaring that the contents of that range are no longer important. The range is going to be overwritten, and the application does not want the memory to migrate out to or in from the paging file. Setting this flag does not guarantee that the range operated on with MEM_RESET will contain zeros. If you want the range to contain zeros, decommit the memory and then recommit it. When you set the MEM_RESET flag, the **VirtualAlloc** function ignores the value of *fProtect*. However, you must still set *fProtect* to a valid protection value, such as PAGE_NOACCESS. **VirtualAlloc** returns an error if you set the MEM_RESET flag and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.
MEM_TOP_DOWN	**Windows NT/2000:** Allocates memory at the highest possible address.
MEM_WRITE_WATCH	**Windows 98:** Causes the system to keep track of the pages that are written to in the allocated region. If you specify this flag, you must also specify the MEM_RESERVE flag. The write-tracking feature remains enabled for the memory region until the region is freed. To retrieve the addresses of the pages that have been written to since the region was allocated or the write-tracking state was reset, call the **GetWriteWatch** function. To reset the write-tracking state, set a flag in the **GetWriteWatch** function or call the **ResetWriteWatch** function.

flProtect

[in] Specifies the type of access protection. If the pages are being committed, any one of the following flags can be specified, along with the PAGE_GUARD and PAGE_NOCACHE protection values, as needed.

Value	Meaning
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_GUARD	**Windows NT/2000:** Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages, thus, act as a one-shot access alarm. The PAGE_GUARD flag is a page protection modifier. An application uses it with one of the other page protection flags, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. **Windows 95/98:** To simulate this behavior, use the PAGE_NOACCESS flag.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This flag is a page protection modifier, only valid when used with one of the page protections other than PAGE_NOACCESS.

(continued)

(continued)

Value	Meaning
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_WRITECOMBINE	Enables write combining, that is, coalescing writes from cache to main memory, where the hardware supports it. This flag is primarily for frame buffer memory. Normally a frame buffer is not cached. However, with this bit, portions of the frame buffer can be cached. This means that writes to the same cache line are coalesced in cache and written out to the frame buffer once upon the first write to another cache line. This can reduce greatly writes across the bus to video memory. If the hardware does not support write combining, the flag is ignored. This flag is a page protection modifier, valid only when used with one of the page protections other than PAGE_NOACCESS.

Return Values

If the function succeeds, the return value is the base address of the allocated region of pages. If the function fails, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

VirtualAlloc can perform the following operations:

- Commit a region of pages reserved by a previous call to the **VirtualAlloc** function.
- Reserve a region of free pages.
- Reserve and commit a region of free pages.

You can use **VirtualAlloc** to reserve a block of pages and then make additional calls to **VirtualAlloc** to commit individual pages from the reserved block. This enables a process to reserve a range of its virtual address space without consuming physical storage until it is needed.

Each page in the process's virtual address space is in one of the following states:

State	Meaning
Free	The page is not committed or reserved and is not accessible to the process. **VirtualAlloc** can reserve, or simultaneously reserve and commit, a free page.

(continued)

(continued)

State	Meaning
Reserved	The range of addresses cannot be used by other allocation functions, but the page is not accessible and has no physical storage associated with it. **VirtualAlloc** can commit a reserved page, but it cannot reserve it a second time. The **VirtualFree** function can release a reserved page, making it a free page.
Committed	Physical storage is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. **VirtualAlloc** can commit an already committed page. This means that you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail. **VirtualFree** can decommit a committed page, releasing the page's storage, or it can simultaneously decommit and release a committed page.

If the *lpAddress* parameter is not NULL, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails and none of the pages are allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

Windows NT/2000: The PAGE_GUARD protection modifier flag establishes guard pages. Guard pages act as one-shot access alarms. For more information, see *Creating Guard Pages*.

Address Windowing Extensions (AWE): The **VirtualAlloc** function can be used to reserve an AWE region of memory within the virtual address space of a specified process. This region of memory can then be used to map physical pages into and out of virtual memory as required by the application.

The MEM_PHYSICAL and MEM_RESERVE flags must be set in the *AllocationType* parameter. The MEM_COMMIT flag must not be set.

The page protection must be set to PAGE_READWRITE.

The **VirtualFree** function can be used on an AWE region of memory—in this case, it will invalidate any physical page mappings in the region when freeing the address space. However, the physical pages themselves are not deleted, and the application subsequently can use them. The application must explicitly call **FreeUserPhysicalPages** to free the physical pages. On process termination, all resources are automatically cleaned up.

Requirements

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

Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **Address Windowing Extensions** (AWE), **AllocateUserPhysicalPages**, **FreeUserPhysicalPages**, **GetWriteWatch**, **HeapAlloc**, **MapUserPhysicalPages**, **MapUserPhysicalPagesScatter**, **ResetWriteWatch**, **VirtualAllocEx**, **VirtualFree**, **VirtualLock**, **VirtualProtect**, **VirtualQuery**

VirtualAllocEx

The **VirtualAllocEx** function reserves or commits (or both) a region of memory within the virtual address space of a specified process. The function initializes the memory it allocates to zero, unless the MEM_RESET flag is set.

```
LPVOID VirtualAllocEx(
  HANDLE hProcess,          // process to allocate memory
  LPVOID lpAddress,         // desired starting address
  SIZE_T dwSize,            // size of region to allocate
  DWORD flAllocationType,   // type of allocation
  DWORD flProtect,          // type of access protection
);
```

Parameters

hProcess

[in] Handle to a process. The function allocates memory within the virtual address space of this process.

You must have PROCESS_VM_OPERATION access to the process. If you do not, the function fails.

lpAddress

[in] Pointer that specifies a desired starting address for the region of pages that you want to allocate.

If you are reserving memory, the function rounds this address down to the nearest 64-KB boundary.

If you are committing memory that is already reserved, the function rounds this address down to the nearest page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

If *lpAddress* is NULL, the function determines where to allocate the region.

dwSize

[in] Specifies the size, in bytes, of the region of memory to allocate.

If *lpAddress* is NULL, the function rounds *dwSize* up to the next page boundary.

If *lpAddress* is not NULL, the function allocates all pages that contain one or more bytes in the range from *lpAddress* to (*lpAddress*+*dwSize*). This means, for example, that a 2-byte range that straddles a page boundary causes the function to allocate both pages.

flAllocationType

[in] Specifies the type of memory allocation. This parameter can be one or more of the following values:

Value	Meaning
MEM_COMMIT	The function allocates actual physical storage in memory or in the paging file on disk for the specified region of memory pages. The function initializes the memory to zero. An attempt to commit a memory page that is already committed does not cause the function to fail. This means that you can commit a range of pages without first determining the current commitment state of each page. If a memory page is not yet reserved, setting this flag causes the function to both reserve and commit the memory page.
MEM_RESERVE	The function reserves a range of the process's virtual address space without allocating any actual physical storage in memory or in the paging file on disk. Other memory allocation functions, such as **malloc** and **LocalAlloc**, cannot use a reserved range of memory until it is released. You can commit reserved memory pages in subsequent calls to the **VirtualAllocEx** function.

(continued)

(continued)

Value	Meaning
MEM_RESET	**Windows NT/2000:** Specifies that memory pages within the range specified by *lpAddress* and *dwSize* will not be written to or read from the paging file. When you set the MEM_RESET flag, you are declaring that the contents of that range are no longer important. The range is going to be overwritten, and the application does not want the memory to migrate out to or in from the paging file. Setting this flag does not guarantee that the range operated on with MEM_RESET will contain zeros. If you want the range to contain zeros, decommit the memory and, then, recommit it. When you set the MEM_RESET flag, the **VirtualAllocEx** function ignores the value of *fProtect*. However, you must still set *fProtect* to a valid protection value, such as PAGE_NOACCESS. **VirtualAllocEx** returns an error if you set the MEM_RESET flag and the range of memory is mapped to a file. A shared view is only acceptable if it is mapped to a paging file.
MEM_TOP_DOWN	The function allocates memory at the highest address possible.

flProtect

[in] Specifies access protection for the region of pages you are allocating. You can specify one of the following flags, along with the PAGE_GUARD and PAGE_NOCACHE protection values, as desired:

Value	Meaning
PAGE_EXECUTE	Enables execute permission to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read permission to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write permission to the committed region of pages.

Value	Meaning
PAGE_GUARD	Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages, thus, act as a one-shot access alarm. The PAGE_GUARD flag is a page protection modifier. An application uses it with one of the other page protection flags, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This flag is a page protection modifier, only valid when used with one of the page protections other than PAGE_NOACCESS.
PAGE_READONLY	Enables read permission to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only permission and execute permission, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write permission to the committed region of pages.
PAGE_WRITECOMBINE	Enables write combining, that is, coalescing writes from cache to main memory, where the hardware supports it. This flag is primarily for frame buffer memory. Normally, a frame buffer is not cached. However, with this bit, portions of the frame buffer can be cached. This means that writes to the same cache line are coalesced in cache and written out to the frame buffer once upon the first write to another cache line. This can reduce greatly writes across the bus to video memory. If the hardware does not support write combining, the flag is ignored. This flag is a page protection modifier, valid only when used with one of the page protections other than PAGE_NOACCESS.

Return Values

If the function succeeds, the return value is the base address of the allocated region of pages.

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

Remarks

The **VirtualAllocEx** function can perform the following operations:

- Commit a region of pages reserved by a previous call to the **VirtualAllocEx** function.
- Reserve a region of free pages.
- Reserve and commit a region of free pages.

You can use **VirtualAllocEx** to reserve a block of pages and then make additional calls to **VirtualAllocEx** to commit individual pages from the reserved block. This lets you reserve a range of a process's virtual address space without consuming physical storage until it is needed.

Each page of memory in a process's virtual address space is in one of the following states:

State	Meaning
Free	The page is not committed or reserved and is not accessible to the process. The **VirtualAllocEx** function can reserve, or simultaneously reserve and commit, a free page.
Reserved	The page is reserved. The range of addresses cannot be used by other allocation functions, but the page is not accessible and has no physical storage associated with it. The **VirtualAllocEx** function can commit a reserved page, but it cannot reserve it a second time. You can use the **VirtualFreeEx** function to release a reserved page in a specified process, making it a free page.
Committed	Physical storage is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read or write to that page. When the process terminates, the system releases the storage for committed pages. The **VirtualAllocEx** function can commit an already committed page. This means that you can commit a range of pages, regardless of whether they have already been committed, and the function will not fail. You can use the **VirtualFreeEx** function to decommit a committed page in a specified process, or to simultaneously decommit and free a committed page.

If the *lpAddress* parameter is not NULL, the function uses the *lpAddress* and *dwSize* parameters to compute the region of pages to be allocated. The current state of the

entire range of pages must be compatible with the type of allocation specified by the *flAllocationType* parameter. Otherwise, the function fails, and none of the pages is allocated. This compatibility requirement does not preclude committing an already committed page; see the preceding list.

The PAGE_GUARD protection modifier flag establishes guard pages. Guard pages act as one-shot access alarms. For more information, see *Creating Guard Pages*.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **HeapAlloc**, **VirtualFreeEx**, **VirtualLock**, **VirtualProtect**, **VirtualQuery**

VirtualFree

The **VirtualFree** function releases or decommits (or both) a region of pages within the virtual address space of the calling process.

To free memory allocated in another process by the **VirtualAllocEx** function, use the **VirtualFreeEx** function.

```
BOOL VirtualFree(
  LPVOID lpAddress,   // address of region
  SIZE_T dwSize,      // size of region
  DWORD dwFreeType    // operation type
);
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be freed. If the *dwFreeType* parameter includes the MEM_RELEASE flag, this parameter must be the base address returned by the **VirtualAlloc** function when the region of pages was reserved.

dwSize

[in] Specifies the size, in bytes, of the region to be freed. If the *dwFreeType* parameter includes the MEM_RELEASE flag, this parameter must be zero. Otherwise, the region of affected pages includes all pages containing one or more bytes in the range from

the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be freed.

dwFreeType

[in] Specifies the type of free operation. This parameter can be one, but not both, of the following values:

Value	Meaning
MEM_DECOMMIT	Decommits the specified region of committed pages. An attempt to decommit an uncommitted page will not cause the function to fail. This means that a range of committed or uncommitted pages can be decommitted without having to worry about a failure.
MEM_RELEASE	Releases the specified region of reserved pages. If this flag is specified, the *dwSize* parameter must be zero, or the function fails.

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

VirtualFree can perform one of the following operations:

- Decommit a region of committed or uncommitted pages.
- Release a region of reserved pages.
- Decommit and release a region of committed or uncommitted pages.

To release a region of pages, the entire range of pages must be in the same state (all reserved or all committed) and the entire region originally reserved by the **VirtualAlloc** function must be released at the same time. If only part of the pages in the original reserved region are committed, you must first call **VirtualFree** to decommit the committed pages and then call **VirtualFree** again to release the entire block.

Pages that have been released are free pages available for subsequent allocation operations. Attempting to read from or write to a free page results in an access violation exception.

VirtualFree can decommit an uncommitted page; this means that a range of committed or uncommitted pages can be decommitted without having to worry about a failure. Decommitting a page releases its physical storage, either in memory or in the paging file on disk. If a page is decommitted but not released, its state changes to reserved, and it can be committed again by a subsequent call to **VirtualAlloc**. Attempting to read from or write to a reserved page results in an access violation exception.

The current state of the entire range of pages must be compatible with the type of free operation specified by the *dwFreeType* parameter. Otherwise, the function fails, and no pages are released or decommitted.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **VirtualFreeEx**

VirtualFreeEx

The **VirtualFreeEx** function releases decommits (or both) a region of memory within the virtual address space of a specified process.

```
BOOL VirtualFreeEx(
  HANDLE hProcess,    // handle to process
  LPVOID lpAddress,   // starting address of memory region
  SIZE_T dwSize,      // size of memory region
  DWORD dwFreeType    // operation type
);
```

Parameters

hProcess

[in] Handle to a process. The function frees memory within the virtual address space of this process.

You must have PROCESS_VM_OPERATION access to this process. If you do not, the function fails.

lpAddress

[in] Pointer to the starting address of the region of memory to free.

If the MEM_RELEASE flag is set in the *dwFreeType* parameter, *lpAddress* must be the base address returned by the **VirtualAllocEx** function when the region was reserved.

dwSize

[in] Specifies the size, in bytes, of the region of memory to free.

If the MEM_RELEASE flag is set in the *dwFreeType* parameter, *dwSize* must be zero. The function frees the entire region that was reserved in the initial allocation call to **VirtualAllocEx**.

If the MEM_DECOMMIT flag is set, the function decommits all memory pages that contain one or more bytes in the range from the *lpAddress* parameter to (*lpAddress*+*dwSize*). This means, for example, that a 2-byte region of memory that straddles a page boundary causes both pages to be decommitted.

The function decommits the entire region that was reserved by **VirtualAllocEx**. If the following three conditions are met:

- the MEM_DECOMMIT flag is set
- *lpAddress* is the base address returned by the **VirtualAllocEx** function when the region was reserved
- *dwSize* is zero

The entire region then enters the reserved state.

dwFreeType

[in] Set of bit flags that specifies the type of free operation. This parameter can be one of the following values:

Value	Meaning
MEM_DECOMMIT	The function decommits the specified region of pages. The pages enter the reserved state. The function does not fail if you attempt to decommit an uncommitted page. This means that you can decommit a range of pages without first determining their current commitment state.
MEM_RELEASE	The function releases the specified region of pages. The pages enter the free state. If you specify this flag, *dwSize* must be zero, and *lpAddress* must point to the base address returned by the **VirtualAllocEx** function when the region was reserved. The function fails if either of these conditions is not met. If any pages in the region are currently committed, the function first decommits and then releases them. The function does not fail if you attempt to release pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining their current commitment state.

Return Values

If the function succeeds, the return value is a nonzero value.

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

Remarks

Each page of memory in a process's virtual address space is in one of the following states:

State	Meaning
Free	The page is neither committed nor reserved. The page is not accessible to the process. Attempting to read from or write to a free page results in an access violation exception. You can use the **VirtualFreeEx** function to put reserved or committed memory pages into the free state.
Reserved	The page is reserved. The range of addresses cannot be used by other allocation functions. The page is not accessible and has no physical storage associated with it. Attempting to read from or write to a free page results in an access violation exception. You can use the **VirtualFreeEx** function to put committed memory pages into the reserved state, and to put reserved memory pages into the free state.
Committed	The page is committed. Physical storage in memory or in the paging file on disk is allocated for the page, and access is controlled by a protection code. The system initializes and loads each committed page into physical memory only at the first attempt to read from or write to that page. When a process terminates, the system releases all storage for committed pages. You can use the **VirtualAllocEx** function to put committed memory pages into either the reserved or free state.

The **VirtualFreeEx** function can perform the following operations:

- Decommit a region of committed or uncommitted pages. After this operation, the pages are in the reserved state.
- Release a region of reserved pages. After this operation, the pages are in the free state.
- Decommit and release a region of committed or uncommitted pages. After this operation, the pages are in the free state.

The **VirtualFreeEx** function can decommit a range of pages that are in different states, some committed and some uncommitted. This means that you can decommit a range of pages without first determining the current commitment state of each page. Decommitting a page releases its physical storage, either in memory or in the paging file on disk.

If a page is decommitted but not released, its state changes to reserved. You can subsequently call **VirtualAllocEx** to commit it, or **VirtualFreeEx** to release it. Attempting to read from or write to a reserved page results in an access violation exception.

The **VirtualFreeEx** function can release a range of pages that are in different states, some reserved and some committed. This means that you can release a range of pages without first determining the current commitment state of each page. The entire range of pages originally reserved by the **VirtualAllocEx** function must be released at the same time.

If a page is released, its state changes to free, and it is available for subsequent allocation operations. Once memory is released or decommitted, you can never refer to the memory again. Any information that may have been in that memory is gone forever. Attempting to read from or write to a free page results in an access violation exception. If you require information, do not decommit or free memory containing that information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **VirtualAllocEx**

VirtualLock

The **VirtualLock** function locks the specified region of the process's virtual address space into memory, ensuring that subsequent access to the region will not incur a page fault.

```
BOOL VirtualLock(
  LPVOID lpAddress,   // first byte in range
  SIZE_T dwSize       // number of bytes in range
);
```

Parameters

lpAddress
[in] Pointer to the base address of the region of pages to be locked.

dwSize
[in] Specifies the size, in bytes, of the region to be locked. The region of affected pages includes all pages that contain one or more bytes in the range from the

lpAddress parameter to (*lpAddress*+*dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be locked.

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

All pages in the specified region must be committed. Memory protected with the PAGE_NOACCESS flag cannot be locked.

Locking pages into memory may degrade the performance of the system by reducing the available RAM and forcing the system to swap out other critical pages to the paging file. By default, a process can lock a maximum of 30 pages. The default limit is intentionally small to avoid severe performance degradation. Applications that need to lock larger numbers of pages must first call the **SetProcessWorkingSetSize** function to increase their minimum and maximum working set sizes. The maximum number of pages that a process can lock is equal to the number of pages in its minimum working set minus a small overhead.

Pages that a process has locked remain resident even when the process is idle for extended periods.

To unlock a region of locked pages, use the **VirtualUnlock** function. Locked pages are automatically unlocked when the process terminates.

This function is unlike the **GlobalLock** or **LocalLock** function in that it does not increment a lock count and translate a handle into a pointer. There is no lock count for virtual pages, so multiple calls to the **VirtualUnlock** function are never required to unlock a region of pages.

Windows 95/98: The **VirtualLock** function is implemented as a stub that has no effect and always returns a nonzero value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **SetProcessWorkingSetSize**, **VirtualUnlock**

VirtualProtect

The **VirtualProtect** function changes the access protection on a region of committed pages in the virtual address space of the calling process.

To change the access protection of any process, use the **VirtualProtectEx** function.

```
BOOL VirtualProtect(
  LPVOID lpAddress,        // region of committed pages
  SIZE_T dwSize,           // size of the region
  DWORD flNewProtect,      // desired access protection
  PDWORD lpflOldProtect    // old protection
);
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the **VirtualAlloc** or **VirtualAllocEx** function using the MEM_RESERVE flag. The pages cannot span adjacent reserved regions that were allocated by separate calls to **VirtualAlloc** or **VirtualAllocEx** using MEM_RESERVE.

dwSize

[in] Specifies the size, in bytes, of the region whose access protection attributes are to be changed. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

flNewProtect

[in] Specifies the new access protection. You can specify any one of the following values, along with the PAGE_GUARD and PAGE_NOCACHE values, as necessary:

Value	Meaning
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_EXECUTE_WRITECOPY	Enables execute, read, and write access to the committed region of pages. The pages are shared read-on-write and copy-on-write.

PAGE_GUARD	**Windows NT/2000:** Pages in the region become guard pages. Any attempt to access a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages, thus, act as a one-shot access alarm. The PAGE_GUARD flag is a page protection modifier. An application uses it with one of the other page protection flags, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. **Windows 95/98:** To simulate this behavior, use the PAGE_NOACCESS flag.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be specified as "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This flag is a page protection modifier, valid only when used with one of the page protections other than PAGE_NOACCESS.
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_WRITECOPY	**Windows NT/2000:** Gives copy-on-write access to the committed region of pages.

lpflOldProtect
[out] Pointer to a variable that receives the previous access protection value of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails.

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 set the access protection value on committed pages only. If the state of any page in the specified region is not committed, the function fails and returns without modifying the access protection of any pages in the specified region.

The **VirtualProtect** function changes the access protection of memory in the calling process, and the **VirtualProtectEx** function changes the access protection of memory in a specified process.

Windows NT/2000: The PAGE_GUARD protection modifier flag establishes guard pages. Guard pages act as one-shot access alarms. For more information, see *Creating Guard Pages*.

Windows 95/98: You cannot use **VirtualProtect** on any memory region located in the shared virtual address space (from 0x80000000 through 0xBFFFFFFF).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **VirtualAlloc**, **VirtualProtectEx**

VirtualProtectEx

The **VirtualProtectEx** function changes the access protection on a region of committed pages in the virtual address space of a specified process.

```
BOOL VirtualProtectEx(
  HANDLE hProcess,          // handle to process
```

```
  LPVOID lpAddress,       // region of committed pages
  SIZE_T dwSize,          // size of region
  DWORD flNewProtect,     // desired access protection
  PDWORD lpflOldProtect   // old protection
);
```

Parameters

hProcess

[in] Handle to the process whose memory protection is to be changed. The handle must have PROCESS_VM_OPERATION access. For more information on PROCESS_VM_OPERATION, see **OpenProcess**.

lpAddress

[in] Pointer to the base address of the region of pages whose access protection attributes are to be changed.

All pages in the specified region must be within the same reserved region allocated when calling the **VirtualAlloc** or **VirtualAllocEx** function using the MEM_RESERVE flag. The pages cannot span adjacent reserved regions that were allocated by separate calls to **VirtualAlloc** or **VirtualAllocEx** using MEM_RESERVE.

dwSize

[in] Specifies the size, in bytes, of the region whose access protection attributes are changed. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes the protection attributes of both pages to be changed.

flNewProtect

[in] Specifies the new access protection. You can specify any one of the following values, along with the PAGE_GUARD and PAGE_NOCACHE values, as desired:

Value	Meaning
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_EXECUTE_WRITECOPY	Enables execute, read, and write access to the committed region of pages. The pages are shared read-on-write and copy-on-write.

(continued)

(continued)

Value	Meaning
PAGE_GUARD	**Windows NT/2000:** Pages in the region become guard pages. Any attempt to read from or write to a guard page causes the system to raise a STATUS_GUARD_PAGE exception and turn off the guard page status. Guard pages, thus, act as a one-shot access alarm. The PAGE_GUARD flag is a page protection modifier. An application uses it with one of the other page protection flags, with one exception: it cannot be used with PAGE_NOACCESS. When an access attempt leads the system to turn off guard page status, the underlying page protection takes over. If a guard page exception occurs during a system service, the service typically returns a failure status indicator. **Windows 95/98:** To simulate this behavior, use the PAGE_NOACCESS flag.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be set to "no cache." This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This flag is a page protection modifier, only valid when used with one of the page protections other than PAGE_NOACCESS.
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_WRITECOPY	**Windows NT/2000:** Gives copy-on-write access to the committed region of pages.

lpflOldProtect
[out] Pointer to a variable that receives the previous access protection of the first page in the specified region of pages. If this parameter is NULL or does not point to a valid variable, the function fails.

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 access protection value can be set only on committed pages. If the state of any page in the specified region is not committed, the function fails and returns without modifying the access protection of any pages in the specified region.

VirtualProtectEx is identical to the **VirtualProtect** function except that it changes the access protection of memory in a specified process.

Windows NT/2000: The PAGE_GUARD protection modifier flag establishes guard pages. Guard pages act as one-shot access alarms. For more information, see *Creating Guard Pages*.

Windows 95/98: You cannot use **VirtualProtectEx** on any memory region located in the shared virtual address space (from 0x80000000 through 0xBFFFFFFF).

Requirements

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.

See Also

Memory Management Overview, *Memory Management Functions*, **VirtualAlloc**, **VirtualProtect**, **VirtualQueryEx**

VirtualQuery

The **VirtualQuery** function provides information about a range of pages in the virtual address space of the calling process.

To obtain information about a range of pages in the address space of another process, use the **VirtualQueryEx** function.

```
DWORD VirtualQuery(
  LPCVOID lpAddress,                    // address of region
  PMEMORY_BASIC_INFORMATION lpBuffer,   // information buffer
  DWORD dwLength                        // size of buffer
);
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be queried. This value is rounded down to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

lpBuffer

[out] Pointer to a **MEMORY_BASIC_INFORMATION** structure in which information about the specified page range is returned.

dwLength

[in] Specifies the size, in bytes, of the buffer pointed to by the *lpBuffer* parameter.

Return Values

The return value is the actual number of bytes returned in the information buffer.

Remarks

VirtualQuery provides information about a region of consecutive pages beginning at a specified address that share the following attributes:

- The state of all pages is the same with the MEM_COMMIT, MEM_RESERVE, MEM_FREE, MEM_PRIVATE, MEM_MAPPED, or MEM_IMAGE flag.
- If the initial page is not free, all pages in the region are part of the same initial allocation of pages reserved by a call to the **VirtualAlloc** function.
- The access of all pages is the same with the PAGE_READONLY, PAGE_READWRITE, PAGE_NOACCESS, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD, or PAGE_NOCACHE flag.

The function determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages or until it encounters a page with a nonmatching set of attributes. The function returns the attributes and the size, in bytes, of the region of pages with matching attributes. For example, if there is a 40-MB region of free memory, and **VirtualQuery** is called on a page that is 10 MB into the region, the function will obtain a state of MEM_FREE and a size of 30 MB.

This function reports on a region of pages in the memory of the calling process, and the **VirtualQueryEx** function reports on a region of pages in the memory of a specified process.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Memory Management Overview, *Memory Management Functions*, **GetSystemInfo**, **MEMORY_BASIC_INFORMATION**, **VirtualQueryEx**

VirtualQueryEx

The **VirtualQueryEx** function provides information about a range of pages within the virtual address space of a specified process.

```
DWORD VirtualQueryEx(
  HANDLE hProcess,                       // handle to process
  LPCVOID lpAddress,                     // address of region
  PMEMORY_BASIC_INFORMATION lpBuffer,    // information buffer
  DWORD dwLength                         // size of buffer
);
```

Parameters

hProcess
[in] Handle to the process whose memory information is queried. The handle must have been opened with the PROCESS_QUERY_INFORMATION flag, which enables using the handle to read information from the process object.

lpAddress
[in] Pointer to the base address of the region of pages to be queried. This value is rounded down to the next page boundary. To determine the size of a page on the host computer, use the **GetSystemInfo** function.

lpBuffer
[out] Pointer to a **MEMORY_BASIC_INFORMATION** structure in which information about the specified page range is returned.

dwLength
[in] Specifies the size, in bytes, of the buffer pointed to by the *lpBuffer* parameter.

Return Values

The return value is the actual number of bytes returned in the information buffer.

Remarks

VirtualQueryEx provides information about a region of consecutive pages beginning at a specified address that share the following attributes:

- The state of all pages is the same with the MEM_COMMIT, MEM_RESERVE, MEM_FREE, MEM_PRIVATE, MEM_MAPPED, or MEM_IMAGE flag.
- If the initial page is not free, all pages in the region are part of the same initial allocation of pages reserved by a call to the **VirtualAllocEx** function.
- The access of all pages is the same with the PAGE_READONLY, PAGE_READWRITE, PAGE_NOACCESS, PAGE_WRITECOPY, PAGE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_READWRITE, PAGE_EXECUTE_WRITECOPY, PAGE_GUARD, or PAGE_NOCACHE flag.

The **VirtualQueryEx** function determines the attributes of the first page in the region and then scans subsequent pages until it scans the entire range of pages, or until it encounters a page with a nonmatching set of attributes. The function returns the attributes and the size, in bytes, of the region of pages with matching attributes. For example, if there is a 40-MB region of free memory, and **VirtualQueryEx** is called on a page that is 10 MB into the region, the function will obtain a state of MEM_FREE and a size of 30 MB.

Requirements

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.

See Also

Memory Management Overview, *Memory Management Functions*, **GetSystemInfo**, **MEMORY_BASIC_INFORMATION**, **VirtualAllocEx**, **VirtualProtectEx**

VirtualUnlock

The **VirtualUnlock** function unlocks a specified range of pages in the virtual address space of a process, enabling the system to swap the pages out to the paging file if necessary.

```
BOOL VirtualUnlock(
  LPVOID lpAddress,   // first byte in range
  SIZE_T dwSize       // number of bytes in range
);
```

Parameters

lpAddress

[in] Pointer to the base address of the region of pages to be unlocked.

dwSize

[in] Specifies the size, in bytes, of the region being unlocked. The region of affected pages includes all pages containing one or more bytes in the range from the *lpAddress* parameter to (*lpAddress+dwSize*). This means that a 2-byte range straddling a page boundary causes both pages to be unlocked.

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

For the function to succeed, the range specified need not match a range passed to a previous call to the **VirtualLock** function, but all pages in the range must be locked.

Windows NT/2000: Calling **VirtualUnlock** on a range of memory that is not locked releases the pages from the process's working set.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Memory Management Overview, *Memory Management Functions*, **VirtualLock**

ZeroMemory

The **ZeroMemory** function fills a block of memory with zeros.

```
VOID ZeroMemory(
  PVOID Destination,  // memory block
  SIZE_T Length       // size of memory block
);
```

Parameters

Destination
[in] Pointer to the starting address of the block of memory to fill with zeros.

Length
[in] Size, in bytes, of the block of memory to fill with zeros.

Return Values

This function has no return value.

Remarks

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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.

See Also

Memory Management Overview, *Memory Management Functions*, **CopyMemory**, **FillMemory**, **MoveMemory**

Memory Management Structures

MEMORY_BASIC_INFORMATION

The **MEMORY_BASIC_INFORMATION** structure contains information about a range of pages in the virtual address space of a process. The **VirtualQuery** and **VirtualQueryEx** functions use this structure.

```
typedef struct _MEMORY_BASIC_INFORMATION {
    PVOID BaseAddress;
    PVOID AllocationBase;
    DWORD AllocationProtect;
    SIZE_T RegionSize;
    DWORD State;
    DWORD Protect;
    DWORD Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

Members

BaseAddress

Pointer to the base address of the region of pages.

AllocationBase

Pointer to the base address of a range of pages allocated by the **VirtualAlloc** function. The page pointed to by the **BaseAddress** member is contained within this allocation range.

AllocationProtect

Specifies the access protection given when the region was initially allocated. This member can be one of the following values, along with the PAGE_GUARD and PAGE_NOCACHE values.

Flag	Meaning
PAGE_EXECUTE	Enables execute access to the committed region of pages. An attempt to read or write to the committed region results in an access violation.
PAGE_EXECUTE_READ	Enables execute and read access to the committed region of pages. An attempt to write to the committed region results in an access violation.
PAGE_EXECUTE_READWRITE	Enables execute, read, and write access to the committed region of pages.
PAGE_EXECUTE_WRITECOPY	Enables execute, read, and write access to the committed region of pages. The pages are shared read-on-write and copy-on-write.
PAGE_GUARD	**Windows NT/2000:** Protects the page with the underlying page protection. However, access to the region causes a "guard page entered" condition to be raised in the subject process. This flag is a page protection modifier, valid only when used with one of the page protections other than PAGE_NOACCESS. **Windows 95/98:** To simulate this behavior, use the PAGE_NOACCESS flag.
PAGE_NOACCESS	Disables all access to the committed region of pages. An attempt to read from, write to, or execute in the committed region results in an access violation exception, called a general protection (GP) fault.

(continued)

(continued)

Flag	Meaning
PAGE_NOCACHE	Allows no caching of the committed regions of pages. The hardware attributes for the physical memory should be set to no cache. This is not recommended for general usage. It is useful for device drivers; for example, mapping a video frame buffer with no caching. This flag is a page protection modifier, valid only when used with one of the page protections other than PAGE_NOACCESS.
PAGE_READONLY	Enables read access to the committed region of pages. An attempt to write to the committed region results in an access violation. If the system differentiates between read-only access and execute access, an attempt to execute code in the committed region results in an access violation.
PAGE_READWRITE	Enables both read and write access to the committed region of pages.
PAGE_WRITECOPY	**Windows NT/2000:** Gives copy-on-write access to the committed region of pages.

RegionSize

Specifies the size, in bytes, of the region, beginning at the base address in which all pages have identical attributes.

State

Specifies the state of the pages in the region. One of the following states is indicated:

State	Meaning
MEM_COMMIT	Indicates committed pages for which physical storage has been allocated, either in memory or in the paging file on disk.
MEM_FREE	Indicates free pages not accessible to the calling process and available to be allocated. For free pages, the information in the **AllocationBase**, **AllocationProtect**, **Protect**, and **Type** members is undefined.
MEM_RESERVE	Indicates reserved pages where a range of the process's virtual address space is reserved without any physical storage being allocated. For reserved pages, the information in the **Protect** member is undefined.

Protect

Specifies the access protection of the pages in the region. One of the flags listed for the **AllocationProtect** member is specified.

Type

Specifies the type of pages in the region. The following types are defined:

Type	Meaning
MEM_IMAGE	Indicates that the memory pages within the region are mapped into the view of an image section.
MEM_MAPPED	Indicates that the memory pages within the region are mapped into the view of a section.
MEM_PRIVATE	Indicates that the memory pages within the region are private (that is, not shared by other processes).

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winnt.h; include windows.h.

See Also

Memory Management Overview, *Memory Management Structures*, **VirtualAlloc**, **VirtualQuery**, **VirtualQueryEx**

MEMORYSTATUS

The **MEMORYSTATUS** structure contains information about the current state of both physical memory and virtual memory. The **GlobalMemoryStatus** function stores information in a **MEMORYSTATUS** structure.

```
typedef struct _MEMORYSTATUS {
    DWORD dwLength;
    DWORD dwMemoryLoad;
    SIZE_T dwTotalPhys;
    SIZE_T dwAvailPhys;
    SIZE_T dwTotalPageFile;
    SIZE_T dwAvailPageFile;
    SIZE_T dwTotalVirtual;
    SIZE_T dwAvailVirtual;
} MEMORYSTATUS, *LPMEMORYSTATUS;
```

Members

dwLength

Size, in bytes, of the **MEMORYSTATUS** data structure. You do not need to set this member before calling the **GlobalMemoryStatus** function; the function sets it.

dwMemoryLoad

Windows NT 3.1–NT 4.0: The percentage of approximately the last 1000 pages of physical memory that is in use.

Windows 2000: The approximate percentage of total physical memory that is in use.

dwTotalPhys
Total size, in bytes, of physical memory.

dwAvailPhys
Size, in bytes, of physical memory available.

dwTotalPageFile
Total size possible, in bytes, of the paging file. Note that this number does not represent the actual physical size of the paging file on disk.

dwAvailPageFile
Size, in bytes, of space available in the paging file. The operating system can enlarge the paging file from time to time. The **dwAvailPageFile** member shows the difference between the size of current committed memory and the current size of the paging file —it does not show the largest size possible of the paging file.

dwTotalVirtual
Total size, in bytes, of the user mode portion of the virtual address space of the calling process.

dwAvailVirtual
Size, in bytes, of unreserved and uncommitted memory in the user mode portion of the virtual address space of the calling process.

Remarks

MEMORYSTATUS reflects the state of memory at the time of the call, as well as the size of the paging file at that time. The operating system can enlarge the paging file up to the maximum size set by the administrator.

On computers with more than 4 GB of memory, the **MEMORYSTATUS** structure can return incorrect information. Windows 2000 reports a value of –1 to indicate an overflow. Earlier versions of Windows NT report a value that is the real amount of memory, module 4 GB. If your application is at risk for this behavior, use the **GlobalMemoryStatusEx** function instead of the **GlobalMemoryStatus** 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 winbase.h; include windows.h.

See Also

Memory Management Overview, *Memory Management Structures*, **GlobalMemoryStatus**, **GlobalMemoryStatusEx**

PROCESS_HEAP_ENTRY

The **PROCESS_HEAP_ENTRY** structure contains information about a heap element. The **HeapWalk** function uses a **PROCESS_HEAP_ENTRY** structure to enumerate the elements of a heap.

```
typedef struct _PROCESS_HEAP_ENTRY {
    PVOID lpData;
    DWORD cbData;
    BYTE cbOverhead;
    BYTE iRegionIndex;
    WORD wFlags;
    union {
        struct {
            HANDLE hMem;
            DWORD dwReserved[ 3 ];
        } Block;
        struct {
            DWORD dwCommittedSize;
            DWORD dwUnCommittedSize;
            LPVOID lpFirstBlock;
            LPVOID lpLastBlock;
        } Region;
    };
} PROCESS_HEAP_ENTRY, *LPPROCESS_HEAP_ENTRY;
```

Members

lpData

Pointer to the data portion of the heap element.

To initiate a **HeapWalk** heap enumeration, set **lpData** to NULL.

If the PROCESS_HEAP_REGION bit flag is set in the **wFlags** member, **lpData** points to the first virtual address used by the region.

If the PROCESS_HEAP_UNCOMMITTED_RANGE bit flag is set in **wFlags**, **lpData** points to the beginning of the range of uncommitted memory.

cbData

Specifies the size, in bytes, of the data portion of the heap element.

If the PROCESS_HEAP_REGION bit flag is set in **wFlags**, **cbData** specifies the total size, in bytes, of the address space that is reserved for this region.

If the PROCESS_HEAP_UNCOMMITTED_RANGE bit flag is set in **wFlags**, **cbData** specifies the size, in bytes, of the range of uncommitted memory.

cbOverhead

Specifies the size, in bytes, of the data used by the system to maintain information about the heap element. These overhead bytes are in addition to the **cbData** bytes of the data portion of the heap element.

If the PROCESS_HEAP_REGION bit flag is set in **wFlags**, **cbOverhead** specifies the size, in bytes, of the heap control structures that describe the region.

If the PROCESS_HEAP_UNCOMMITTED_RANGE bit flag is set in **wFlags**, **cbOverhead** specifies the size, in bytes, of the control structures that describe this uncommitted range.

iRegionIndex
Handle to the heap region that contains the heap element. A heap consists of one or more regions of virtual memory, each with a unique region index.

In the first heap entry returned for most heap regions, **HeapWalk** sets the PROCESS_HEAP_REGION flag in the **wFlags** member. When this flag is set, the members of the **Region** structure contain additional information about the region.

The **HeapAlloc** function sometimes uses the **VirtualAlloc** function to allocate large blocks from a growable heap. The heap manager treats such a large block allocation as a separate region with a unique region index. **HeapWalk** does not set the PROCESS_HEAP_REGION flag in the heap entry returned for a large block region, so the members of the **Region** structure are not valid. You can use the **VirtualQuery** function to get additional information about a large block region.

wFlags
A set of bit flags that specify properties of the heap element. Some of these flags affect the meaning of other members of this **PROCESS_HEAP_ENTRY** data structure. The following bit-flag constants are defined:

Value	Meaning
PROCESS_HEAP_ENTRY_BUSY	If this flag is set, the heap element is an allocated block. If both this flag and the PROCESS_HEAP_ENTRY_MOVEABLE flag are set, the **Block** structure becomes valid. The **hMem** member of the **Block** structure contains a handle to the allocated, moveable memory block.
PROCESS_HEAP_ENTRY_DDESHARE	This flag is only valid if the PROCESS_HEAP_ENTRY_BUSY flag is set, indicating that the heap element is an allocated block. If this flag is valid and set, the block was allocated with the GMEM_DDESHARE flag. For more information on the GMEM_DDESHARE flag, see **GlobalAlloc**.
PROCESS_HEAP_ENTRY_MOVEABLE	This flag is only valid if the PROCESS_HEAP_ENTRY_BUSY flag is set, indicating that the heap element is an allocated block.

	If this flag is valid and set, the block was allocated with the LMEM_MOVEABLE or GMEM_MOVEABLE flag, and the **Block** structure becomes valid. The **hMem** member of the **Block** structure contains a handle to the allocated, moveable memory block.
PROCESS_HEAP_REGION	If this flag is set, the heap element is located at the beginning of a region of contiguous virtual memory in use by the heap. If this flag is set, the **lpData** member of the structure points to the first virtual address used by the region; the **cbData** member specifies the total size, in bytes, of the address space that is reserved for this region; and the **cbOverhead** member specifies the size, in bytes, of the heap control structures that describe the region. If this flag is set, the **Region** structure becomes valid. The **dwCommittedSize**, **dwUnCommittedSize**, **lpFirstBlock**, and **lpLastBlock** members of the structure contain additional information about the region.
PROCESS_HEAP_UNCOMMITTED_RANGE	If this flag is set, the heap element is located in a range of uncommitted memory within the heap region. If this flag is set, the **lpData** member points to the beginning of the range of uncommitted memory; the **cbData** member specifies the size, in bytes, of the range of uncommitted memory; and the **cbOverhead** member specifies the size, in bytes, of the control structures that describe this uncommitted range.

Block

This structure is valid only if both the PROCESS_HEAP_ENTRY_BUSY and PROCESS_HEAP_ENTRY_MOVEABLE flags in **wFlags** are set.

The members of the **Block** structure are as follows:

Member	Description
hMem	Contains a handle to the allocated, moveable memory block.
dwReserved	Reserved; not used.

Region

This structure is valid only if the PROCESS_HEAP_REGION flag is set in the **wFlags** member.

The members of the **Region** structure are as follows:

Member	Description
dwCommittedSize	Specifies the number of bytes in the heap region that are currently committed as free memory blocks, busy memory blocks, or heap control structures. This is an optional field that is set to zero if the number of committed bytes is not available.
dwUnCommittedSize	Specifies the number of bytes in the heap region that are currently uncommitted. This is an optional field that is set to zero if the number of uncommitted bytes is not available.
lpFirstBlock	Pointer to the first valid memory block in this heap region.
lpLastBlock	Pointer to the first invalid memory block in this heap region.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

See Also

Memory Management Overview, *Memory Management Structures*, **GlobalAlloc**, **HeapAlloc**, **HeapWalk**, **VirtualAlloc**, **VirtualQuery**

CHAPTER 8

Interprocess Communications

Interprocess Communications

The Microsoft Win32 API provides mechanisms for facilitating communications and data sharing between applications. Collectively, the activities enabled by these mechanisms are called *interprocess communications* (IPC). Some forms of IPC facilitate the division of labor among several specialized processes. Other forms of IPC facilitate the division of labor among computers on a network.

Typically, applications can use IPC categorized as clients or servers. A *client* is an application or a process that requests a service from some other application or process. A *server* is an application or a process that responds to a client request. Many applications act as both a client and a server, depending on the situation. For example, a word processing application might act as a client in requesting a summary table of manufacturing costs from a spreadsheet application acting as a server. The spreadsheet application, in turn, might act as a client in requesting the latest inventory levels from an automated inventory control application.

About Interprocess Communications

As computer users become more sophisticated, they demand more power from the applications they use. To meet this demand, developers add more features to the applications, and the applications become larger. Large applications eventually become unmanageable, both from a development standpoint and from a user-interface standpoint.

One method you can use to manage larger applications is to produce a specialized application that provides a limited number of features, then enable it to communicate and share data with other specialized applications using some form of IPC. It is not necessary for a single application meet all its users' expectations; applications can communicate and cooperate.

The following IPC mechanisms are supported by the Win32 API:

- Clipboard
- COM
- Dynamic Data Exchange (DDE)
- File Mapping
- Mailslots
- Pipes

- RPC
- Windows Sockets
- **WM_COPYDATA**

Choosing an IPC Mechanism

After you decide that your application would benefit from IPC, you must decide which of the available IPC methods to use. It is likely that an application will use several IPC mechanisms. For example, all Win32-based applications should provide at least minimal support for the clipboard. In addition, COM and DDE offer the application an opportunity to communicate with applications that support these protocols. By supporting the protocols for the clipboard, COM, and DDE, you enable your application to share data with other applications, without knowing anything about the applications themselves.

The answers to these questions determine whether an application can benefit by using one or more of the IPC mechanisms available in the Win32 API.

- Should the application be able to communicate with other applications running on other computers on a network, or is it sufficient for the application to communicate only with applications on the local computer?
- Should the application be able to communicate with applications running on other computers that may be running under different operating systems (that is, MS-DOS, 16-bit Windows, UNIX)?
- Should the user of the application have to choose the other application(s) with which the application communicates, or can the application implicitly find its cooperating partners?
- Should the application communicate with many different applications in a general way, such as allowing cut-and-paste operations with any other application, or should its communications requirements be limited to a restricted set of interactions with specific other applications?
- Is performance a critical aspect of the application? All IPC mechanisms include some amount of overhead.
- Should the application be a GUI application or a console application? Some IPC mechanisms require a GUI application.

Using the Clipboard for IPC

The clipboard acts as a central depository for data sharing among applications. When a user performs a cut or copy operation in an application, the application puts the selected data on the clipboard in one or more standard or application-defined formats. Any other application can then retrieve the data from the clipboard, choosing from the available formats that it understands. The clipboard is a very loosely coupled exchange medium, where applications need only agree on the data format. The applications can reside on the same computer or on different computers on a network.

Key Point All Win32-based applications should support the clipboard for those data formats that they understand. For example, a text editor or word processor should be able at least to produce and accept clipboard data in pure text format. For more information, see *Clipboard*.

Using COM for IPC

Applications that use OLE manage *compound documents*—that is, documents made up of data from a variety of different applications. OLE provides services that make it easy for applications to call on other applications for data editing. For example, a word processor that uses OLE could embed a graph from a spreadsheet. The user could start the spreadsheet automatically from within the word processor by choosing the embedded chart for editing. OLE takes care of starting the spreadsheet and presenting the graph for editing. When the user quit the spreadsheet, the graph would be updated in the original word processor document. The spreadsheet appears to be an extension of the word processor.

The foundation of OLE is the Component Object Model (COM). A software component that uses COM can communicate with a wide variety of other components, even those that have not yet been written. The components interact as objects and clients. Distributed COM extends the COM programming model so that it works across a network.

Key Point OLE supports compound documents and enables an application to include embedded or linked data that, when chosen, automatically starts another application for data editing. This enables the application to be extended by any other application that uses OLE. COM objects provide access to an object's data through one or more sets of related functions, known as *interfaces*. For more information, see *COM and ActiveX Object Services*.

Using DDE for IPC

The *Microsoft Win32 Developer's Reference Library* does not include programmatic reference about DDE, because there are better technologies that should be used instead of DDE (see the Key Point at the bottom of this section).

DDE is a protocol that enables applications to exchange data in a variety of formats. Applications can use DDE for one-time data exchanges or ongoing exchanges in which the applications update one another as new data becomes available.

The data formats used by DDE are the same as those used by the clipboard. DDE can be thought of as an extension of the clipboard mechanism. The clipboard is almost always used for a one-time response to a user command, such as choosing the Paste command from a menu. DDE is also usually initiated by a user command, but it often continues to function without further user interaction. You also can define custom DDE

data formats for special-purpose IPC between applications with more tightly coupled communications requirements.

DDE exchanges can occur between applications running on the same computer or on different computers on a network.

Key Point DDE is not as efficient as newer technologies. However, you can still use DDE if other IPC mechanisms are not suitable or if you must interface with an existing application that only supports DDE. For more information, see *Dynamic Data Exchange* and *Dynamic Data Exchange Management Library*.

Using a File Mapping for IPC

File mapping enables a process to treat the contents of a file as if they were a block of memory in the process's address space. The process can use simple pointer operations to examine and modify the contents of the file. When two or more processes access the same file mapping, each process receives a pointer to memory in its own address space that it can use to read or modify the contents of the file. The processes must use a synchronization object, such as a semaphore, to prevent data corruption in a multitasking environment.

You can use a special case of file mapping to provide *named shared memory* between processes. If you specify the system swapping file when creating a file-mapping object, the file-mapping object is treated as a shared memory block. Other processes can access the same block of memory by opening the same file-mapping object.

File mapping is quite efficient and also provides operating system supported security attributes that can help prevent unauthorized data corruption. File mapping can be used only between processes on a local computer; it cannot be used over a network.

Key Point File mapping is an efficient way for two or more processes on the same computer to share data, but you must provide synchronization between the processes. For more information, see *File Mapping* and *Synchronization*.

Using a Mailslot for IPC

Mailslots provide one-way communication. Any process that creates a mailslot is a *mailslot server*. Other processes, called *mailslot clients*, send messages to the mailslot server by writing a message to its mailslot. Incoming messages are always appended to the mailslot. The mailslot saves the messages until the mailslot server has read them. A process can be both a mailslot server and a mailslot client, so two-way communication is possible using multiple mailslots.

A mailslot client can send a message to a mailslot on its local computer, to a mailslot on another computer, or to all mailslots with the same name on all computers in a specified network domain. Messages broadcast to all mailslots on a domain can be no longer than

400 bytes, whereas messages sent to a single mailslot are limited only by the maximum message size specified by the mailslot server when it created the mailslot.

Key Point Mailslots offer an easy way for applications to send and receive short messages. They also provide the ability to broadcast messages across all computers in a network domain. For more information, see *Mailslots*.

Using Pipes for IPC

The Win32 API provides two types of pipes for two-way communication: anonymous pipes and named pipes. Anonymous (or unnamed) pipes enable related processes to transfer information to each other. Typically, an anonymous pipe is used for redirecting the standard input or output of a child process so that it can exchange data with its parent process. To exchange data in both directions (duplex operation), you must create two anonymous pipes. The parent process writes data to one pipe using its write handle, while the child process reads the data from that pipe using its read handle. Similarly, the child process writes data to the other pipe and the parent process reads from it. Anonymous pipes cannot be used over a network, nor can they be used between unrelated processes.

Named pipes are used to transfer data between processes that are not related processes and between processes on different computers. Typically, a *named-pipe server* process creates a named pipe with a well-known name or a name that is to be communicated to its clients. A *named-pipe client* process that knows the name of the pipe can open its other end, subject to access restrictions specified by named-pipe server process. After both the server and client have connected to the pipe, they can exchange data by performing read and write operations on the pipe.

Key Point Anonymous pipes provide an efficient way to redirect standard input or output to child processes on the same computer. Named pipes provide a simple programming interface for transferring data between two processes, whether they reside on the same computer or over a network. For more information, see *Pipes*.

Using RPC for IPC

The Win32 API provides RPC to enable applications to call functions remotely. Therefore, RPC makes IPC as easy as calling a function. RPC operates between processes on a single computer or on different computers on a network.

The RPC provided by the Win32 API is compliant with the Open Software Foundation (OSF) Distributed Computing Environment (DCE). This means that Win32-based applications that use RPC are able to communicate with applications running with other operating systems that support DCE. RPC automatically supports data conversion to account for different hardware architectures and for byte-ordering between dissimilar environments.

RPC clients and servers are tightly coupled but still maintain high performance. The system makes extensive use of RPC to facilitate a client-server relationship between different parts of the operating system.

Key Point RPC is a function-level interface, with support for automatic data conversion and for communications with other operating systems. Using RPC, you can create high-performance, tightly coupled distributed applications. For more information, see *Microsoft RPC Components*.

Using Windows Sockets for IPC

Windows Sockets is a protocol-independent interface. It takes advantage of the communications capabilities of the underlying protocols. In Windows Sockets 2, a socket handle can be used optionally as a file handle with the standard file I/O functions.

Windows Sockets are based on the sockets first popularized by Berkeley Software Distribution (BSD). An application that uses Windows Sockets can communicate with other socket implementation on other types of systems. However, not all transport service providers support all available options.

Key Point Windows Sockets is a protocol-independent interface capable of supporting current and emerging networking capabilities. For more information, see *Overview of Windows Sockets 2*.

Using WM_COPYDATA for IPC

The **WM_COPYDATA** message enables you to send data from one application to another. The receiving application gets the data in a **COPYDATASTRUCT** structure. When data is being passed between a 16-bit Windows-based application and a Win32-based application, the system translates any pointers.

For example code, see the following Platform SDK samples:

- SPY. This sample, located in MSTOOLS\SAMPLES\SDKTOOLS\SPY, uses **WM_COPYDATA** to pass data to another application.
- INTEROP. This sample, located in SCT\SAMPLES, uses **WM_COPYDATA** to dispatch calls from a 16-bit Windows-based application to a Win32-based DLL.

Interprocess Communications Reference

Interprocess Communications Structures

COPYDATASTRUCT

The **COPYDATASTRUCT** structure contains data to be passed to another application by the **WM_COPYDATA** message.

```
typedef struct tagCOPYDATASTRUCT {
    ULONG_PTR dwData;
    DWORD     cbData;
    PVOID     lpData;
} COPYDATASTRUCT, *PCOPYDATASTRUCT;
```

Members

dwData

Specifies data to be passed to the receiving application.

cbData

Specifies the size, in bytes, of the data pointed to by the **lpData** member.

lpData

Pointer to data to be passed to the receiving application. This member can be 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.

See Also

Interprocess Communications Overview, *Interprocess Communications Structures*, **WM_COPYDATA**

Interprocess Communications Messages

The following message is used for interprocess communication:

WM_COPYDATA

WM_COPYDATA

An application sends the **WM_COPYDATA** message to pass data to another application.

To send this message, call the **SendMessage** function with the following parameters (do not call the **PostMessage** function).

```
SendMessage(
  (HWND) hWnd,              // handle to destination window
  WM_COPYDATA,              // message to send
  (WPARAM) wParam;          // handle to window (HWND)
  (LPARAM) lParam;          // data (PCOPYDATASTRUCT)
);
```

Parameters

wParam
  Handle to the window passing the data.

lParam
  Pointer to a **COPYDATASTRUCT** structure that contains the data to be passed.

Return Values

If the receiving application processes this message, it should return TRUE; otherwise, it should return FALSE.

Remarks

The data being passed must not contain pointers or other references to objects not accessible to the application receiving the data.

While this message is being sent, the referenced data must not be changed by another thread of the sending process.

The receiving application should consider the data read-only. The *lParam* parameter is valid only during the processing of the message. The receiving application should not free the memory referenced by *lParam*. If the receiving application must access the data after **SendMessage** returns, it must copy the data into a local buffer.

Requirements

Windows NT/2000: Requires Windows 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.

Interprocess Communications Overview, *Interprocess Communications Messages*, **SendMessage**, **COPYDATASTRUCT**

Atoms

An *atom table* is a system-defined table that stores strings and corresponding identifiers. An application places a string in an atom table and receives a 16-bit integer, called an *atom*, that can be used to access the string. A string that has been placed in an atom table is called an *atom name*.

About Atom Tables

The system provides a number of atom tables. Each atom table serves a different purpose. For example, dynamic data exchange (DDE) applications use the global atom table to share item-name and topic-name strings with other applications. Instead of passing actual strings, a DDE application passes global atoms to its partner application. The partner uses the atoms to obtain the strings from the atom table.

Applications can use local atom tables to store their own item-name associations.

The system uses atom tables that are not directly accessible to applications. However, the application uses these atoms when calling the Win32 application programming interface (API). For example, registered clipboard formats are stored in an internal atom table used by the system. An application adds atoms to this atom table using the **RegisterClipboardFormat** function. Also, registered classes are stored in an internal atom table used by the system. An application adds atoms to this atom table using the **RegisterClass** or **RegisterClassEx** function.

Global Atom Tables

The *global atom table* is available to all applications. When an application places a string in the global atom table, the system generates an atom that is unique throughout the system. Any application that has the atom can obtain the string it identifies by querying the global atom table.

An application that defines a private DDE-data format for sharing data with other applications should place the format name in the global atom table. This technique prevents conflicts with the names of formats defined by the system or by other applications, and makes the identifiers (atoms) for the messages or formats available to the other applications.

Local Atom Tables

An application can use a *local atom table* to efficiently manage a large number of strings used only within the application. These strings, and the associated atoms, are available only to the application that created the table.

An application requiring the same string in a number of structures can reduce memory usage by using a local atom table. Rather than copying the string into each structure, the application can place the string in the atom table and include the resulting atom in the structures. In this way, a string appears in memory only once, but can be used many times in the application.

Applications also can use local atom tables to save time when searching for a particular string. To perform a search, an application need only place the search string in the atom table and compare the resulting atom with the atoms in the relevant structures. Comparing atoms is typically faster than comparing strings.

Atom tables are implemented as hash tables. By default, a local atom table uses 37 buckets for its hash table. However, you can change the number of buckets used by calling the **InitAtomTable** function. If the application calls **InitAtomTable**, however, it must do so before calling any other atom-management functions.

Atom Types

Applications can create two types of atoms: string atoms and integer atoms. The values of integer atoms and string atoms do not overlap, so both types of atoms can be used in the same block of code.

Several Win32 functions accept either strings or atoms as parameters. When passing an atom to these functions, an application can use the **MAKEINTATOM** macro to convert the atom into a form that can be used by the function.

Atom Reference

Atom Functions

AddAtom

The **AddAtom** function adds a character string to the local atom table and returns a unique value (an atom) identifying the string.

```
ATOM AddAtom(
  LPCTSTR lpString   // string to add
);
```

Parameters

lpString

[in] Pointer to the null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings differing only in case are considered identical. The case of the first string added is preserved and returned by the **GetAtomName** function.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the newly created atom.

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

Remarks

The **AddAtom** function stores no more than one copy of a given string in the atom table. If the string is already in the table, the function returns the existing atom and, in the case of a string atom, increments the string's reference count.

If *lpString* has the form "#1234", **AddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000, or greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **AddAtom** returns a string atom.

Requirements

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

Atoms Overview, *Atom Functions*, **DeleteAtom**, **FindAtom**, **GetAtomName**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalFindAtom**, **GlobalGetAtomName**, **MAKEINTATOM**

DeleteAtom

The **DeleteAtom** function decrements the reference count of a local string atom. If the atom's reference count is reduced to zero, **DeleteAtom** removes the string associated with the atom from the local atom table.

```
ATOM DeleteAtom(
  ATOM nAtom   // atom to delete
);
```

Parameters

nAtom
[in] Identifies the atom to be deleted.

Return Values

If the function succeeds, the return value is zero.

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

Remarks

A string atom's reference count specifies the number of times the atom has been added to the atom table. The **AddAtom** function increments the count on each call. The **DeleteAtom** function decrements the count on each call but removes the string only if the atom's reference count is zero.

Each call to **AddAtom** should have a corresponding call to **DeleteAtom**. Do not call **DeleteAtom** more times than you call **AddAtom**, or you might delete the atom while other clients are using it.

The **DeleteAtom** function has no effect on an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF). The function always returns zero for an integer atom.

Requirements

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.

See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **FindAtom**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalFindAtom**, **MAKEINTATOM**

FindAtom

The **FindAtom** function searches the local atom table for the specified character string, and retrieves the atom associated with that string.

```
ATOM FindAtom(
  LPCTSTR lpString   // string to find
);
```

Parameters

lpString

[in] Pointer to the null-terminated character string to search for.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the atom associated with the given string.

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

Remarks

Even though the system preserves the case of a string in an atom table, the search performed by the **FindAtom** function is not case-sensitive.

If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalFindAtom**

GetAtomName

The **GetAtomName** function retrieves a copy of the character string associated with the specified local atom.

```
UINT GetAtomName(
  ATOM nAtom,        // atom identifying character string
  LPTSTR lpBuffer,   // buffer for atom string
  int nSize          // size of buffer
);
```

Parameters

nAtom
[in] Specifies the local atom that identifies the character string to be retrieved.

lpBuffer
[out] Pointer to the buffer for the character string.

nSize
[in] Specifies the size, in **TCHARs**, of the buffer.

Return Values

If the function succeeds, the return value is the length of the string copied to the buffer, in **TCHARs**, not including the terminating null character.

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

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **FindAtom**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalFindAtom**, **GlobalGetAtomName**, **MAKEINTATOM**

GlobalAddAtom

The **GlobalAddAtom** function adds a character string to the global atom table, and returns a unique value (an atom) identifying the string.

```
ATOM GlobalAddAtom(
  LPCTSTR lpString   // string to add
);
```

Parameters

lpString
[in] Pointer to the null-terminated string to be added. The string can have a maximum size of 255 bytes. Strings that differ in case only are considered identical. The case of the first string of this name added to the table is preserved and returned by the **GlobalGetAtomName** function.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the newly created atom.

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

Remarks

If the string already exists in the global atom table, the atom for the existing string is returned, and the atom's reference count is incremented.

The string associated with the atom is not deleted from memory until its reference count is zero. For more information, see the **GlobalDeleteAtom** function.

Global atoms are not deleted automatically when the application terminates. For every call to the **GlobalAddAtom** function, there must be a corresponding call to the **GlobalDeleteAtom** function.

If the *lpString* parameter has the form "#1234", **GlobalAddAtom** returns an integer atom whose value is the 16-bit representation of the decimal number specified in the string (0x04D2, in this example). If the decimal value specified is 0x0000, or greater than or equal to 0xC000, the return value is zero, indicating an error. If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

If *lpString* has any other form, **GlobalAddAtom** returns a string atom.

Requirements

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

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **FindAtom**, **GetAtomName**, **GlobalDeleteAtom**, **GlobalFindAtom**, **GlobalGetAtomName**, **MAKEINTATOM**

GlobalDeleteAtom

The **GlobalDeleteAtom** function decrements the reference count of a global string atom. If the atom's reference count reaches zero, **GlobalDeleteAtom** removes the string associated with the atom from the global atom table.

```
ATOM GlobalDeleteAtom(
  ATOM nAtom   // atom to delete
);
```

Parameters

nAtom
[in] Identifies the atom and character string to be deleted.

Return Values

If the function succeeds, the return value is zero.

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

Remarks

A string atom's reference count specifies the number of times the string has been added to the atom table. The **GlobalAddAtom** function increments the reference count of a string that already exists in the global atom table each time it is called.

Each call to **GlobalAddAtom** should have a corresponding call to **GlobalDeleteAtom**. Do not call **GlobalDeleteAtom** more times than you call **GlobalAddAtom**, or you might delete the atom while other clients are using it. Applications using DDE should follow the rules on global atom management to prevent leaks and premature deletion.

GlobalDeleteAtom has no effect on an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF). The function always returns zero for an integer atom.

Requirements

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.

See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **FindAtom**, **GlobalAddAtom**, **GlobalFindAtom**, **MAKEINTATOM**

GlobalFindAtom

The **GlobalFindAtom** function searches the global atom table for the specified character string, and retrieves the global atom associated with that string.

```
ATOM GlobalFindAtom(
  LPCTSTR lpString    // string to find
);
```

Parameters

lpString

[in] Pointer to the null-terminated character string for which to search.

Alternatively, you can use an integer atom that has been converted using the **MAKEINTATOM** macro. See the Remarks for more information.

Return Values

If the function succeeds, the return value is the global atom associated with the given string.

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

Remarks

Even though the system preserves the case of a string in an atom table as it was originally entered, the search performed by **GlobalFindAtom** is not case-sensitive.

If *lpString* was created by the **MAKEINTATOM** macro, the low-order word must be in the range 0x0001 through 0xBFFF. If the low-order word is not in this range, the function fails.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **FindAtom**, **GetAtomName**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalGetAtomName**

GlobalGetAtomName

The **GlobalGetAtomName** function retrieves a copy of the character string associated with the specified global atom.

```
UINT GlobalGetAtomName(
  ATOM nAtom,        // atom identifier
  LPTSTR lpBuffer,   // buffer for atom string
  int nSize          // size of buffer
);
```

Parameters

nAtom
[in] Identifies the global atom associated with the character string to be retrieved.

lpBuffer
[out] Pointer to the buffer for the character string.

nSize
[in] Specifies the size, in characters, of the buffer.

Return Values

If the function succeeds, the return value is the length of the string copied to the buffer, in characters, not including the terminating null character.

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

Remarks

The string returned for an integer atom (an atom whose value is in the range 0x0001 to 0xBFFF) is a null-terminated string in which the first character is a pound sign (#) and the remaining characters represent the unsigned integer atom value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **FindAtom**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalFindAtom**, **MAKEINTATOM**

InitAtomTable

The **InitAtomTable** function initializes the local atom table, and sets the number of hash buckets to the specified size.

```
BOOL InitAtomTable(
  DWORD nSize   // size of atom table
);
```

Parameters

nSize

[in] Specifies the number of hash buckets to use for the atom table. If this parameter is zero, the default number of hash buckets are created.

To achieve better performance, specify a prime number in *nSize*.

Return Values

If the function succeeds, the return value is nonzero.

If the function fails, the return value is zero.

Remarks

An application does not need to use this function to use a local atom table. The default number of hash buckets used is 37. If an application uses **InitAtomTable**, however, it should call the function before any other atom-management function.

If an application uses a large number of local atoms, it can reduce the time required to add an atom to the local atom table or to find an atom in the table by increasing the size of the table. However, this increases the amount of memory required to maintain the table.

The number of buckets in the global atom table cannot be changed. If the atom table has already been initialized, either explicitly by a prior call to **InitAtomTable**, or implicitly by the use of any atom-management function, **InitAtomTable** returns success without changing the number of hash buckets.

! Requirements

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.

+ See Also

Atoms Overview, *Atom Functions*, **AddAtom**, **DeleteAtom**, **FindAtom**, **GetAtomName**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalFindAtom**, **GlobalGetAtomName**

Atom Macros

MAKEINTATOM

The **MAKEINTATOM** macro converts the specified atom into a string, so it can be passed to functions which accept either atoms or strings.

```
LPTSTR MAKEINTATOM(
  WORD wInteger  // integer to make into atom
);
```

Parameters

wInteger

Specifies the numeric value to be made into an integer atom. This parameter can be either an integer atom or a string atom.

Return Values

The return value is an integer atom cast to a string pointer.

Remarks

Although the return value of the **MAKEINTATOM** macro is cast as an **LPTSTR** value, it cannot be used as a string pointer except when it is passed to atom-management functions that require an **LPTSTR** argument.

Requirements

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.

See Also

Atoms Overview, *Atom Macros*, **AddAtom**, **DeleteAtom**, **GetAtomName**, **GlobalAddAtom**, **GlobalDeleteAtom**, **GlobalGetAtomName**

Clipboard

The *clipboard* is a set of functions and messages that enable Win32-based applications to transfer data. Because all applications have access to the clipboard, data can be easily transferred between applications or within an application.

This overview does not describe how to copy and paste linked or embedded objects. For information on these subjects, see the COM documentation on MSDN.

About the Clipboard

A memory object on the clipboard can be in any data format, called a *clipboard format*. Each format is identified by an unsigned integer value. For standard (predefined) clipboard formats, this value is a constant defined by the Microsoft Win32 API; for registered clipboard formats, it is the return value of the **RegisterClipboardFormat** function.

Except for registering clipboard formats, individual windows perform most clipboard operations. Typically, a window procedure transfers information to or from the clipboard, in response to the **WM_COMMAND** message.

The clipboard is user-driven. A window should transfer data to or from the clipboard only in response to a command from the user. A window must not use the clipboard to transfer data without the user's knowledge.

Clipboard Formats

A window can place more than one object on the clipboard, each representing the same information in a different clipboard format. Users need not be aware of the clipboard formats used for an object on the clipboard.

Standard Clipboard Formats

The clipboard formats defined by the system are called *standard clipboard formats*. These clipboard formats are described in the following table:

Value	Meaning
CF_BITMAP	A handle to a bitmap (**HBITMAP**).
CF_DIB	A memory object containing a **BITMAPINFO** structure followed by the bitmap bits.
CF_DIBV5	**Windows 2000:** A memory object containing a **BITMAPV5HEADER** structure followed by the bitmap color space information and the bitmap bits.
CF_DIF	Software Arts' Data Interchange Format.
CF_DSPBITMAP	Bitmap display format associated with a private format. The *hMem* parameter must be a handle to data that can be displayed in bitmap format in place of the privately formatted data.
CF_DSPENHMETAFILE	Enhanced metafile display format associated with a private format. The *hMem* parameter must be a handle to data that can be displayed in enhanced metafile format in place of the privately formatted data.

(continued)

(continued)

Value	Meaning
CF_DSPMETAFILEPICT	Metafile-picture display format associated with a private format. The *hMem* parameter must be a handle to data that can be displayed in metafile-picture format in place of the privately formatted data.
CF_DSPTEXT	Text display format associated with a private format. The *hMem* parameter must be a handle to data that can be displayed in text format in place of the privately formatted data.
CF_ENHMETAFILE	A handle to an enhanced metafile (**HENHMETAFILE**).
CF_GDIOBJFIRST through CF_GDIOBJLAST	Range of integer values for application-defined GDI object clipboard formats. Handles associated with clipboard formats in this range are not automatically deleted using the **GlobalFree** function when the clipboard is emptied. Also, when using values in this range, the *hMem* parameter is not a handle to a GDI object, but is a handle allocated by the **GlobalAlloc** function with the GMEM_DDESHARE and GMEM_MOVEABLE flags.
CF_HDROP	A handle to type **HDROP** that identifies a list of files. An application can retrieve information about the files by passing the handle to the **DragQueryFile** functions.
CF_LOCALE	The data is a handle to the locale identifier associated with text in the clipboard. When you close the clipboard, if it contains CF_TEXT data but no CF_LOCALE data, the system automatically sets the CF_LOCALE format to the current input locale. You can use the CF_LOCALE format to associate a different locale with the clipboard text. An application that pastes text from the clipboard can retrieve this format to determine which character set was used to generate the text. Note that the clipboard does not support plain text in multiple character sets. To achieve this, use a formatted text data type, such as RTF, instead. **Windows NT/2000:** The system uses the code page associated with CF_LOCALE to implicitly convert from CF_TEXT to CF_UNICODETEXT. Therefore, the correct code page table is used for the conversion.

Value	Meaning
CF_METAFILEPICT	Handle to a metafile picture format, as defined by the **METAFILEPICT** structure. When passing a CF_METAFILEPICT handle by means of dynamic data exchange (DDE), the application responsible for deleting *hMem* should also free the metafile referred to by the CF_METAFILEPICT handle.
CF_OEMTEXT	Text format containing characters in the OEM character set. Each line ends with a carriage return/line feed (CR-LF) combination. A null character signals the end of the data.
CF_OWNERDISPLAY	Owner-display format. The clipboard owner must display and update the clipboard viewer window, and receive the **WM_ASKCBFORMATNAME**, **WM_HSCROLLCLIPBOARD**, **WM_PAINTCLIPBOARD**, **WM_SIZECLIPBOARD**, and **WM_VSCROLLCLIPBOARD** messages. The *hMem* parameter must be NULL.
CF_PALETTE	Handle to a color palette. Whenever an application places data in the clipboard that depends on or assumes a color palette, it should place the palette on the clipboard, too. If the clipboard contains data in the CF_PALETTE (logical color palette) format, the application should use the **SelectPalette** and **RealizePalette** functions to realize (compare) any other data in the clipboard against that logical palette. When displaying clipboard data, the clipboard always uses, as its current palette, any object on the clipboard that is in the CF_PALETTE format.
CF_PENDATA	Data for the pen extensions to the Microsoft Windows for Pen Computing.
CF_PRIVATEFIRST through CF_PRIVATELAST	Range of integer values for private clipboard formats. Handles associated with private clipboard formats are not freed automatically; the clipboard owner must free such handles, typically in response to the **WM_DESTROYCLIPBOARD** message.
CF_RIFF	Represents audio data more complex than can be represented in a CF_WAVE standard wave format.
CF_SYLK	Microsoft Symbolic Link (SYLK) format.
CF_TEXT	Text format. Each line ends with a carriage return/line feed (CR-LF) combination. A null character signals the end of the data. Use this format for ANSI text.

(continued)

(continued)

Value	Meaning
CF_WAVE	Represents audio data in one of the standard wave formats, such as 11-kHz or 22-kHz pulse code modulation (PCM).
CF_TIFF	Tagged-image file format.
CF_UNICODETEXT	**Windows NT/2000:** Unicode text format. Each line ends with a carriage return/line feed (CR-LF) combination. A null character signals the end of the data.

Registered Clipboard Formats

Many applications work with data that cannot be translated into a standard clipboard format without loss of information. These applications can create their own clipboard formats. A clipboard format that is defined by an application is called a *registered clipboard format*. For example, if a word-processing application copied formatted text to the clipboard using a standard text format, the formatting information would be lost. The solution would be to register a new clipboard format, such as Rich Text Format (RTF).

To register a new clipboard format, use the **RegisterClipboardFormat** function. This function takes the name of the format and returns and unsigned integer value that represents the registered clipboard format. To retrieve the name of the registered clipboard format, pass the unsigned integer value to the **GetClipboardFormatName** function.

If more than one application registers a clipboard format with exactly the same name, the clipboard format is registered only once. Both calls to the **RegisterClipboardFormat** function return the same value. In this way, two different applications can share data by using a registered clipboard format.

Private Clipboard Formats

An application can identify a private clipboard format by defining a value in the range CF_PRIVATEFIRST through CF_PRIVATELAST. An application can use a private clipboard format for an application-defined data format that does not need to be registered with the system.

Data handles associated with private clipboard formats are automatically freed by the system. Windows that use private clipboard formats can use the **WM_DESTROYCLIPBOARD** message to free any related resources that are no longer needed.

For more information about the **WM_DESTROYCLIPBOARD** message, see *Clipboard Ownership*.

An application can place data handles on the clipboard by defining a private format in the range CF_GDIOBJFIRST through CF_GDIOBJLAST. When using values in this range, the data handle is not a handle to a GDI object, but a handle allocated by the **GlobalAlloc** function with the GMEM_DDESHARE and GMEM_MOVEABLE flags. When the clipboard is emptied the system automatically deletes the object using the **GlobalFree** function.

Multiple Clipboard Formats

A window can place more than one clipboard object on the clipboard, each representing the same information in a different clipboard format. When placing information on the clipboard, the window should provide data in as many formats as possible. To find out how many formats are currently used on the clipboard, call the **CountClipboardFormats** function.

Clipboard formats that contain the most information should be placed on the clipboard first, followed by less descriptive formats. A window pasting information from the clipboard typically retrieves a clipboard object in the first format it recognizes. Because clipboard formats are enumerated in the order they are placed on the clipboard, the first recognized format is also the most descriptive.

For example, suppose a user copies styled text from a word-processing document. The window containing the document might first place data on the clipboard in a registered format, such as RTF. Subsequently, the window would place data on the clipboard in a less descriptive format, such as text (CF_TEXT).

When the content of the clipboard is pasted into another window, the window retrieves data in the most descriptive format it recognizes. If the window recognizes RTF, the corresponding data is pasted into the document. Otherwise, the text data is pasted into the document, and the formatting information is lost.

Synthesized Clipboard Formats

The system implicitly converts data between certain clipboard formats: if a window requests data in a format that is not on the clipboard, the system converts an available format to the requested format. The system can convert data as indicated in the following table:

Clipboard Format	Conversion Format	Platform Support
CF_BITMAP	CF_DIB	Windows NT/2000, Windows 95/98
CF_BITMAP	CF_DIBV5	Windows 2000
CF_DIB	CF_BITMAP	Windows NT/2000, Windows 95/98
CF_DIB	CF_PALETTE	Windows NT/2000, Windows 95/98
CF_DIB	CF_DIBV5	Windows 2000
CF_DIBV5	CF_BITMAP	Windows 2000
CF_DIBV5	CF_DIB	Windows 2000
CF_DIBV5	CF_PALETTE	Windows 2000
CF_ENHMETAFILE	CF_METAFILEPICT	Windows NT/2000, Windows 95/98

(continued)

(continued)

Clipboard Format	Conversion Format	Platform Support
CF_METAFILEPICT	CF_ENHMETAFILE	Windows NT/2000, Windows 95/ 98
CF_OEMTEXT	CF_TEXT	Windows NT/2000, Windows 95/98
CF_OEMTEXT	CF_UNICODETEXT	Windows NT/2000
CF_TEXT	CF_OEMTEXT	Windows NT/2000, Windows 95/98
CF_TEXT	CF_UNICODETEXT	Windows NT/2000
CF_UNICODETEXT	CF_OEMTEXT	Windows NT/2000
CF_UNICODETEXT	CF_TEXT	Windows NT/2000

If the system provides an automatic type conversion for a particular clipboard format, there is no advantage to placing the conversion format(s) on the clipboard.

If the system provides an automatic type conversion for a particular clipboard format, and you call **EnumClipboardFormats** to enumerate the clipboard data formats, the system first enumerates the format that is on the clipboard, followed by the formats to which it can be converted.

When copying bitmaps, it is best to place the CF_DIB or CF_DIBV5 format on the clipboard, because the colors in a device-dependent bitmap (CF_BITMAP) are relative to the system palette, which may change before the bitmap is pasted. If the CF_DIB or CF_DIBV5 format is on the clipboard, and a window requests the CF_BITMAP format, the system renders the device-independent bitmap (DIB) using the current palette at that time.

If you place the CF_BITMAP format on the clipboard (and not CF_DIB), the system renders the CF_DIB or CF_DIBV5 clipboard format as soon as the clipboard is closed. This ensures that the correct palette is used to generate the DIB. If you place the CF_DIBV5 format with the bitmap color space information in the clipboard, the system will convert the bitmap bits from the bitmap color space to the sRGB color space when CF_DIB or CF_DIBV5 is requested. If CF_DIBV5 is requested when there is no color space information in the clipboard, the system returns sRGB color space information in the **BITMAPV5HEADER** structure. Conversions between other clipboard formats occur upon demand.

If the clipboard contains data in the CF_PALETTE format, the application should use the **SelectPalette** and **RealizePalette** functions to realize any other data in the clipboard against that logical palette.

There are two clipboard formats for metafiles: CF_ENHMETAFILE and CF_METAFILEPICT. Specify CF_ENHMETAFILE for enhanced metafiles and CF_METAFILEPICT for Windows metafiles.

Clipboard Reference

Clipboard Functions

ChangeClipboardChain

The **ChangeClipboardChain** function removes a specified window from the chain of clipboard viewers.

```
BOOL ChangeClipboardChain(
  HWND hWndRemove,  // handle to window to remove
  HWND hWndNewNext  // handle to next window
);
```

Parameters

hWndRemove

[in] Handle to the window to be removed from the chain. The handle must have been passed to the **SetClipboardViewer** function.

hWndNewNext

[in] Handle to the window that follows the *hWndRemove* window in the clipboard viewer chain. (This is the handle returned by **SetClipboardViewer**, unless the sequence was changed in response to a **WM_CHANGECBCHAIN** message.)

Return Values

The return value indicates the result of passing the **WM_CHANGECBCHAIN** message to the windows in the clipboard viewer chain. Because a window in the chain typically returns FALSE when it processes **WM_CHANGECBCHAIN**, the return value from **ChangeClipboardChain** is typically FALSE. If there is only one window in the chain, the return value is typically TRUE.

Remarks

The window identified by *hWndNewNext* replaces the *hWndRemove* window in the chain. The **SetClipboardViewer** function sends a **WM_CHANGECBCHAIN** message to the first window in the clipboard viewer chain.

Requirements

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

Clipboard Overview, *Clipboard Functions*, **ChangeClipboardChain**, **SetClipboardViewer**, **WM_CHANGECBCHAIN**

CloseClipboard

The **CloseClipboard** function closes the clipboard.

```
BOOL CloseClipboard(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

When the window has finished examining or changing the clipboard, close the clipboard by calling **CloseClipboard**. This enables other windows to access the clipboard.

Do not place an object on the clipboard after calling **CloseClipboard**.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **GetOpenClipboardWindow**, **OpenClipboard**

CountClipboardFormats

The **CountClipboardFormats** function retrieves the number of different data formats that are currently on the clipboard.

```
int CountClipboardFormats(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the number of different data formats currently on the clipboard.

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

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **EnumClipboardFormats**, **RegisterClipboardFormat**

EmptyClipboard

The **EmptyClipboard** function empties the clipboard and frees handles to data in the clipboard. The function then assigns ownership of the clipboard to the window that currently has the clipboard open.

```
BOOL EmptyClipboard(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

Before calling **EmptyClipboard**, an application must open the clipboard by using the **OpenClipboard** function. If the application specifies a NULL window handle when opening the clipboard, **EmptyClipboard** succeeds but sets the clipboard owner to 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

Clipboard Overview, *Clipboard Functions*, **OpenClipboard**, **SetClipboardData**, **WM_DESTROYCLIPBOARD**

EnumClipboardFormats

The **EnumClipboardFormats** function lets you enumerate the data formats that are available currently on the clipboard.

Clipboard data formats are stored in an ordered list. To perform an enumeration of clipboard data formats, you make a series of calls to the **EnumClipboardFormats** function. For each call, the *format* parameter specifies an available clipboard format, and the function returns the next available clipboard format.

```
UINT EnumClipboardFormats(
  UINT format    // available clipboard format
);
```

Parameters

format

[in] Specifies a clipboard format that is known to be available.

To start an enumeration of clipboard formats, set *format* to zero. When *format* is zero, the function retrieves the first available clipboard format. For subsequent calls during an enumeration, set *format* to the result of the previous **EnumClipboardFormat** call.

Return Values

If the function succeeds, the return value is the clipboard format that follows the specified format. In other words, the next available clipboard format.

If the function fails, the return value is zero. To get extended error information, call **GetLastError**. If the clipboard is not open, the function fails.

If there are no more clipboard formats to enumerate, the return value is zero. In this case, the **GetLastError** function returns the value NO_ERROR. This lets you distinguish between function failure and the end of enumeration.

Remarks

You must open the clipboard before enumerating its formats. Use the **OpenClipboard** function to open the clipboard. The **EnumClipboardFormats** function fails if the clipboard is not open.

The **EnumClipboardFormats** function enumerates formats in the order that they were placed on the clipboard. If you are copying information to the clipboard, add clipboard objects, in order, from the most descriptive clipboard format to the least descriptive clipboard format. If you are pasting information from the clipboard, retrieve the first clipboard format that you can handle. That will be the most descriptive clipboard format that you can handle.

The system provides automatic type conversions for certain clipboard formats. In the case of such a format, this function enumerates the specified format, then enumerates the formats to which it can be converted. For more information, see *Standard Clipboard Formats* and *Synthesized Clipboard Formats*.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **CountClipboardFormats**, **OpenClipboard**, **RegisterClipboardFormat**

GetClipboardData

The **GetClipboardData** function retrieves data from the clipboard in a specified format. The clipboard must have been opened previously.

```
HANDLE GetClipboardData(
  UINT uFormat   // clipboard format
);
```

Parameters

uFormat

[in] Specifies a clipboard format. For a description of the standard clipboard formats, see *Standard Clipboard Formats*.

Return Values

If the function succeeds, the return value is the handle to a clipboard object in the specified format.

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

Remarks

An application can enumerate the available formats in advance by using the **EnumClipboardFormats** function.

The clipboard controls the handle that the **GetClipboardData** function returns, not the application. The application should copy the data immediately. The application must neither free the handle nor leave it locked. The application must not use the handle after the **EmptyClipboard** or **CloseClipboard** function is called, or after the **SetClipboardData** function is called with the same clipboard format.

The system performs implicit data format conversions between certain clipboard formats when an application calls the **GetClipboardData** function. For example, if the CF_OEMTEXT format is on the clipboard, a window can retrieve data in the CF_TEXT format. The format on the clipboard is converted to the requested format on demand. For more information, see *Synthesized Clipboard Formats*.

! Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **CloseClipboard**, **EmptyClipboard**, **EnumClipboardFormats**, **SetClipboardData**

GetClipboardFormatName

The **GetClipboardFormatName** function retrieves from the clipboard the name of the specified registered format. The function copies the name to the specified buffer.

```
int GetClipboardFormatName(
  UINT format,             // clipboard format to retrieve
  LPTSTR lpszFormatName,   // format name
  int cchMaxCount          // length of format name buffer
);
```

Parameters

format

[in] Specifies the type of format to be retrieved. This parameter must not specify any of the predefined clipboard formats.

lpszFormatName

[out] Pointer to the buffer that is to receive the format name.

cchMaxCount

[in] Specifies the maximum length, in characters, of the string to be copied to the buffer. If the name exceeds this limit, it is truncated.

Return Values

If the function succeeds, the return value is the length, in characters, of the string copied to the buffer.

If the function fails, the return value is zero, indicating that the requested format does not exist or is predefined. 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

Clipboard Overview, *Clipboard Functions*, **EnumClipboardFormats**, **RegisterClipboardFormat**

GetClipboardOwner

The **GetClipboardOwner** function retrieves the window handle of the current owner of the clipboard.

```
HWND GetClipboardOwner(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the handle to the window that owns the clipboard.

If the clipboard is not owned, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The clipboard can still contain data even if the clipboard is not owned currently.

In general, the clipboard owner is the window that last placed data in clipboard. The **EmptyClipboard** function assigns clipboard ownership.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **EmptyClipboard**, **GetClipboardViewer**

GetClipboardSequenceNumber

The **GetClipboardSequenceNumber** function returns the clipboard sequence number for the current window station.

```
DWORD GetClipboardSequenceNumber(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is the clipboard sequence number. If you do not have WINSTA_ACCESSCLIPBOARD access to the window station, the function returns zero.

Remarks

The system keeps a serial number for the clipboard for each window station. This number is incremented whenever the contents of the clipboard change or the clipboard is emptied. You can track this value to determine whether the clipboard contents have changed, and optimize creating DataObjects. If clipboard rendering is delayed, the sequence number is not incremented until the changes are rendered.

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

Clipboard Overview, *Clipboard Functions*

GetClipboardViewer

The **GetClipboardViewer** function retrieves the handle to the first window in the clipboard viewer chain.

```
HWND GetClipboardViewer(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the handle to the first window in the clipboard viewer chain.

If there is no clipboard viewer, 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 winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Clipboard Overview, *Clipboard Functions*, **GetClipboardOwner**, **SetClipboardViewer**

GetOpenClipboardWindow

The **GetOpenClipboardWindow** function retrieves the handle to the window that currently has the clipboard open.

```
HWND GetOpenClipboardWindow(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is the handle to the window that has the clipboard open. If no window has the clipboard open, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

If an application or dynamic-link library (DLL) specifies a NULL window handle when calling the **OpenClipboard** function, the clipboard is opened but not associated with a window. In such a case, **GetOpenClipboardWindow** returns 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

Clipboard Overview, *Clipboard Functions*, **GetClipboardOwner**, **GetClipboardViewer**, **OpenClipboard**

GetPriorityClipboardFormat

The **GetPriorityClipboardFormat** function returns the first available clipboard format in the specified list.

```
int GetPriorityClipboardFormat(
  UINT *paFormatPriorityList,  // array of clipboard formats
  int cFormats                 // number of entries in array
);
```

Parameters

paFormatPriorityList

[in] Pointer to an array of unsigned integers identifying clipboard formats, in priority order. For a description of the standard clipboard formats, see *Standard Clipboard Formats*.

cFormats

[in] Specifies the number of entries in the *paFormatPriorityList* array. This value must not be greater than the number of entries in the list.

Return Values

If the function succeeds, the return value is the first clipboard format in the list for which data is available. If the clipboard is empty, the return value is NULL. If the clipboard contains data, but not in any of the specified formats, 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: Requires version 1.0 or later.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Clipboard Overview, *Clipboard Functions*, **CountClipboardFormats**, **EnumClipboardFormats**, **GetClipboardFormatName**, **IsClipboardFormatAvailable**, **RegisterClipboardFormat**

IsClipboardFormatAvailable

The **IsClipboardFormatAvailable** function determines whether the clipboard contains data in the specified format.

```
BOOL IsClipboardFormatAvailable(
  UINT format   // clipboard format
);
```

Parameters

format

[in] Specifies a standard or registered clipboard format. For a description of the standard clipboard formats, see *Standard Clipboard Formats*.

Return Values

If the clipboard format is available, the return value is nonzero.

If the clipboard format is not available, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Typically, an application that recognizes only one clipboard format would call this function when processing the **WM_INITMENU** or **WM_INITMENUPOPUP** message. The application would then enable or disable the **Paste** menu item, depending on the return value. Applications that recognize more than one clipboard format should use the **GetPriorityClipboardFormat** function for this purpose.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **CountClipboardFormats**, **EnumClipboardFormats**, **GetPriorityClipboardFormat**, **RegisterClipboardFormat**, **WM_INITMENU**, **WM_INITMENUPOPUP**

OpenClipboard

The **OpenClipboard** function opens the clipboard for examination, and prevents other applications from modifying the clipboard content.

```
BOOL OpenClipboard(
  HWND hWndNewOwner   // handle to window
);
```

Parameters

hWndNewOwner
[in] Handle to the window to be associated with the open clipboard. If this parameter is NULL, the open clipboard is associated with the current task.

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

OpenClipboard fails if another window has the clipboard open.

An application should call the **CloseClipboard** function after every successful call to **OpenClipboard**.

The window identified by the *hWndNewOwner* parameter does not become the clipboard owner unless the **EmptyClipboard** function is called.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **CloseClipboard**, **EmptyClipboard**

RegisterClipboardFormat

The **RegisterClipboardFormat** function registers a new clipboard format. This format can be used then as a valid clipboard format.

```
UINT RegisterClipboardFormat(
  LPCTSTR lpszFormat   // name of new format
);
```

Parameters

lpszFormat
[in] Pointer to a null-terminated string that names the new format.

Return Values

If the function succeeds, the return value identifies the registered clipboard format.

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

Remarks

If a registered format with the specified name already exists, a new format is not registered and the return value identifies the existing format. This enables more than one application to copy and paste data using the same registered clipboard format. Note that the format name comparison is case-insensitive.

Registered clipboard formats are identified by values in the range 0xC000 through 0xFFFF.

When registered clipboard formats are placed on or retrieved from the clipboard, they must be in the form of an **HGLOBAL** 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.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

Clipboard Overview, *Clipboard Functions*, **CountClipboardFormats**, **EnumClipboardFormats**, **GetClipboardFormatName**

SetClipboardData

The **SetClipboardData** function places data on the clipboard in a specified clipboard format. The window must be the current clipboard owner, and the application must have called the **OpenClipboard** function. (When responding to the **WM_RENDERFORMAT** and **WM_RENDERALLFORMATS** messages, the clipboard owner must not call **OpenClipboard** before calling **SetClipboardData**.)

```
HANDLE SetClipboardData(
  UINT uFormat,  // clipboard format
  HANDLE hMem    // data handle
);
```

Parameters

uFormat

[in] Specifies a clipboard format. This parameter can be a registered format or any of the standard clipboard formats. For more information, see *Registered Clipboard Formats* and *Standard Clipboard Formats*.

hMem

[in] Handle to the data in the specified format. This parameter can be NULL, indicating that the window provides data in the specified clipboard format (renders the format) upon request. If a window delays rendering, it must process the **WM_RENDERFORMAT** and **WM_RENDERALLFORMATS** messages.

After **SetClipboardData** is called, the system owns the object identified by the *hMem* parameter. The application can read the data, but must not free the handle or leave it locked until the **CloseClipboard** function is called. (The application can access the data after calling **CloseClipboard**.) If the *hMem* parameter identifies a memory

object, the object must have been allocated using the **GlobalAlloc** function with the GMEM_MOVEABLE and GMEM_DDESHARE flags.

Return Values

If the function succeeds, the return value is the handle to the data.

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

Remarks

The *uFormat* parameter can identify a registered clipboard format, or it can be one of the standard clipboard formats. For more information, see *Registered Clipboard Formats* and *Standard Clipboard Formats*.

If an application calls **SetClipboardData**, in response to **WM_RENDERFORMAT** or **WM_RENDERALLFORMATS**, the application should not use the handle after **SetClipboardData** has been called.

The system performs implicit data format conversions between certain clipboard formats when an application calls the **GetClipboardData** function. For example, if the CF_OEMTEXT format is on the clipboard, a window can retrieve data in the CF_TEXT format. The format on the clipboard is converted to the requested format on demand. For more information, see *Synthesized Clipboard Formats*.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Functions*, **CloseClipboard**, **GetClipboardData**, **OpenClipboard**, **RegisterClipboardFormat**, **WM_RENDERALLFORMATS**, **WM_RENDERFORMAT**

SetClipboardViewer

The **SetClipboardViewer** function adds the specified window to the chain of clipboard viewers. Clipboard viewer windows receive a **WM_DRAWCLIPBOARD** message whenever the content of the clipboard changes.

```
HWND SetClipboardViewer(
  HWND hWndNewViewer   // handle to clipboard viewer window
);
```

Parameters

hWndNewViewer
[in] Handle to the window to be added to the clipboard chain.

Return Values

If the function succeeds, the return value identifies the next window in the clipboard viewer chain. If an error occurs, or there are no other windows in the clipboard viewer chain, the return value is NULL. To get extended error information, call **GetLastError**.

Remarks

The windows that are part of the clipboard viewer chain, called clipboard viewer windows, must process the clipboard messages **WM_CHANGECBCHAIN** and **WM_DRAWCLIPBOARD**. Each clipboard viewer window calls the **SendMessage** function to pass these messages to the next window in the clipboard viewer chain.

A clipboard viewer window must remove itself eventually from the clipboard viewer chain by calling the **ChangeClipboardChain** function—for example, in response to the **WM_DESTROY** 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

Clipboard Overview, *Clipboard Functions*, **ChangeClipboardChain**, **GetClipboardViewer**, **SendMessage**

Clipboard Structures

METAFILEPICT

The **METAFILEPICT** structure defines the metafile picture format used for exchanging metafile data through the clipboard.

```
typedef struct tagMETAFILEPICT {
    LONG        mm;
```

```
    LONG      xExt;
    LONG      yExt;
    HMETAFILE hMF;
} METAFILEPICT, *LPMETAFILEPICT;
```

Members

mm

Specifies the mapping mode in which the picture is drawn.

xExt

Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. (For more information about these modes, see the **yExt** member.) The x-extent specifies the width of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.

yExt

Specifies the size of the metafile picture for all modes except the MM_ISOTROPIC and MM_ANISOTROPIC modes. The y-extent specifies the height of the rectangle within which the picture is drawn. The coordinates are in units that correspond to the mapping mode.

For MM_ISOTROPIC and MM_ANISOTROPIC modes, which can be scaled, the **xExt** and **yExt** members contain an optional suggested size in MM_HIMETRIC units. For MM_ANISOTROPIC pictures, **xExt** and **yExt** can be zero when no suggested size is supplied. For MM_ISOTROPIC pictures, an aspect ratio must be supplied even when no suggested size is given. (If a suggested size is given, the aspect ratio is implied by the size.) To give an aspect ratio without implying a suggested size, set **xExt** and **yExt** to negative values whose ratio is the appropriate aspect ratio. The magnitude of the negative **xExt** and **yExt** values is ignored; only the ratio is used.

hMF

Handle to a memory metafile.

! Requirements

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

See Also

Clipboard Overview, *Clipboard Structures*, **SetClipboardData**

Clipboard Messages

WM_ASKCBFORMATNAME

The **WM_ASKCBFORMATNAME** message is sent to the clipboard owner by a clipboard viewer window to request the name of a CF_OWNERDISPLAY clipboard format.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,         // handle to window
  UINT uMsg,         // WM_ASKCBFORMATNAME
  WPARAM wParam,     // size of buffer
  LPARAM lParam      // format name (LPTSTR)
);
```

Parameters

wParam
Specifies the size, in characters, of the buffer pointed to by the *lParam* parameter.

lParam
Pointer to the buffer that is to receive the clipboard format name.

Return Values

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

Remarks

In response to this message, the clipboard owner should copy the name of the owner-display format to the specified buffer, not exceeding the buffer size specified by the *wParam* parameter.

A clipboard viewer window sends this message to the clipboard owner to determine the name of the CF_OWNERDISPLAY format—for example, to initialize a menu listing available formats.

! Requirements

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

Clipboard Overview, *Clipboard Messages*

WM_CHANGECBCHAIN

The **WM_CHANGECBCHAIN** message is sent to the first window in the clipboard viewer chain when a window is being removed from the chain.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_CHANGECBCHAIN
  WPARAM wParam,    // handle to window to be removed (HWND)
  LPARAM lParam     // handle to next window in chain (HWND)
);
```

Parameters

wParam

Handle to the window being removed from the clipboard viewer chain.

lParam

Handle to the next window in the chain following the window being removed. This parameter is NULL if the window being removed is the last window in the chain.

Return Values

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

Remarks

Each clipboard viewer window saves the handle to the next window in the clipboard viewer chain. Initially, this handle is the return value of the **SetClipboardViewer** function.

When a clipboard viewer window receives the **WM_CHANGECBCHAIN** message, it should call the **SendMessage** function to pass the message to the next window in the chain, unless the next window is the window being removed. In this case, the clipboard viewer should save the handle specified by the *lParam* parameter as the next window in the chain.

Requirements

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

Clipboard Overview, *Clipboard Messages*, **SendMessage**, **SetClipboardViewer**

WM_CLEAR

An application sends a **WM_CLEAR** message to an edit control or combo box to delete (clear) the current selection, if any, from the edit control.

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

```
SendMessage(
  (HWND) hWnd,              // handle to destination window
  WM_CLEAR,                 // 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

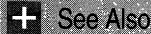
This message does not return a value.

Remarks

The deletion performed by the **WM_CLEAR** message can be undone by sending the edit control an **EM_UNDO** message.

To delete the current selection and place the deleted content on the clipboard, use the **WM_CUT** message.

When sent to a combo box, the **WM_CLEAR** message is handled by its edit control. This message has no effect when sent to a combo box with the CBS_DROPDOWNLIST style.



Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Messages*, **EM_UNDO**, **WM_COPY**, **WM_CUT**, **WM_PASTE**

WM_COPY

An application sends the **WM_COPY** message to an edit control or combo box to copy the current selection to the clipboard in CF_TEXT format.

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

```
SendMessage(
  (HWND) hWnd,              // handle to destination window
  WM_COPY,                  // 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 does not return a value.

Remarks

When sent to a combo box, the **WM_COPY** message is handled by its edit control. This message has no effect when sent to a combo box with the 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

Clipboard Overview, *Clipboard Messages*, **WM_CLEAR**, **WM_CUT**, **WM_PASTE**

WM_CUT

An application sends a **WM_CUT** message to an edit control or combo box to delete (cut) the current selection, if any, in the edit control, and copy the deleted text to the clipboard in CF_TEXT format.

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

```
SendMessage(
  (HWND) hWnd,              // handle to destination window
```

(continued)

(continued)

```
  WM_CUT,                    // 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 does not return a value.

Remarks

The deletion performed by the **WM_CUT** message can be undone by sending the edit control an **EM_UNDO** message.

To delete the current selection without placing the deleted text on the clipboard, use the **WM_CLEAR** message.

When sent to a combo box, the **WM_CUT** message is handled by its edit control. This message has no effect when sent to a combo box with the 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

Clipboard Overview, *Clipboard Messages*, **EM_UNDO**, **WM_CLEAR**, **WM_COPY**, **WM_PASTE**

WM_DESTROYCLIPBOARD

The **WM_DESTROYCLIPBOARD** message is sent to the clipboard owner when a call to the **EmptyClipboard** function empties the clipboard.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_DESTROYCLIPBOARD
  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.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Messages*, **EmptyClipboard**

WM_DRAWCLIPBOARD

The **WM_DRAWCLIPBOARD** message is sent to the first window in the clipboard viewer chain when the content of the clipboard changes. This enables a clipboard viewer window to display the new content of the clipboard.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_DRAWCLIPBOARD
  WPARAM wParam,   // not used
  LPARAM lParam    // not used
);
```

Parameters

This message has no parameters.

Remarks

Only clipboard viewer windows receive this message. These are windows that have been added to the clipboard viewer chain by using the **SetClipboardViewer** function.

Each window that receives the **WM_DRAWCLIPBOARD** message must call the **SendMessage** function to forward the message to the next window in the clipboard viewer chain. The handle to the next window in the chain is returned by **SetClipboardViewer**, and may change in response to a **WM_CHANGECBCHAIN** message.

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

Clipboard Overview, *Clipboard Messages*, **SendMessage**, **SetClipboardViewer**, **WM_CHANGECBCHAIN**

WM_HSCROLLCLIPBOARD

The **WM_HSCROLLCLIPBOARD** message is sent to the clipboard owner by a clipboard viewer window. This occurs when the clipboard contains data in the CF_OWNERDISPLAY format and an event occurs in the clipboard viewer's horizontal scroll bar. The owner should scroll the clipboard image and update the scroll bar values.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_HSCROLLCLIPBOARD
  WPARAM wParam,   // handle to window (HWND)
  LPARAM lParam    // scroll bar code and scroll box position
);
```

Parameters

wParam
Handle to the clipboard viewer window.

lParam
The low-order word of *lParam* specifies a scroll bar event. This parameter can be one of the following values:

Value	Meaning
SB_ENDSCROLL	End scroll.
SB_LEFT	Scroll to upper left.
SB_LINELEFT	Scroll left by one unit.
SB_LINERIGHT	Scroll right by one unit.
SB_PAGELEFT	Scroll left by the width of the window.
SB_PAGERIGHT	Scroll right by the width of the window.
SB_RIGHT	Scroll to lower right.

SB_THUMBPOSITION	Scroll to absolute position. The current position is specified by the high-order word.

The high-order word of *lParam* specifies the current position of the scroll box, if the low-order word of *lParam* is SB_THUMBPOSITION; otherwise, the high-order word is not used.

Return Values

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

Remarks

The clipboard owner can use the **ScrollWindow** function to scroll the image in the clipboard viewer window and invalidate the appropriate region.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

WM_PAINTCLIPBOARD

The **WM_PAINTCLIPBOARD** message is sent to the clipboard owner by a clipboard viewer window when the clipboard contains data in the CF_OWNERDISPLAY format and the clipboard viewer's client area needs repainting.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_PAINTCLIPBOARD
  WPARAM wParam,    // handle to window (HWND)
  LPARAM lParam     // handle to memory object (HGLOBAL)
);
```

Parameters

wParam
  Handle to the clipboard viewer window.

lParam
  Handle to a global memory object that contains a **PAINTSTRUCT** structure. The structure defines the part of the client area to paint.

Return Values

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

Remarks

To determine whether the entire client area, or just a portion of it, needs repainting, the clipboard owner must compare the dimensions of the drawing area given in the **rcpaint** member of **PAINTSTRUCT** to the dimensions given in the most recent **WM_SIZECLIPBOARD** message.

The clipboard owner must use the **GlobalLock** function to lock the memory that contains the **PAINTSTRUCT** structure. Before returning, the clipboard owner must unlock that memory by using the **GlobalUnlock** 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

Clipboard Overview, *Clipboard Messages*, **GlobalLock**, **GlobalUnlock**, **PAINTSTRUCT**, **WM_SIZECLIPBOARD**

WM_PASTE

An application sends a **WM_PASTE** message to an edit control or combo box to copy the current content of the clipboard to the edit control at the current caret position. Data is inserted only if the clipboard contains data in CF_TEXT format.

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

```
SendMessage(
  (HWND) hWnd,              // handle to destination window
  WM_PASTE,                 // 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 does not return a value.

Remarks

When sent to a combo box, the **WM_PASTE** message is handled by its edit control. This message has no effect when sent to a combo box with the 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

Clipboard Overview, *Clipboard Messages*, **WM_CLEAR**, **WM_COPY**, **WM_CUT**

WM_RENDERALLFORMATS

The **WM_RENDERALLFORMATS** message is sent to the clipboard owner before it is destroyed, if the clipboard owner has delayed rendering one or more clipboard formats. For the content of the clipboard to remain available to other applications, the clipboard owner must render data in all the formats it is capable of generating, and place the data on the clipboard by calling the **SetClipboardData** function.

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

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

When responding to a **WM_RENDERALLFORMATS** message, the clipboard owner must call the **OpenClipboard** and **EmptyClipboard** functions before calling **SetClipboardData**.

When the application returns, the system removes any unrendered formats from the list of available clipboard formats. For information about delayed rendering, see **SetClipboardData**.

Requirements

Windows NT/2000: Requires Windows 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

Clipboard Overview, *Clipboard Messages*, **EmptyClipboard**, **OpenClipboard**, **SetClipboardData**, **WM_RENDERFORMAT**

WM_RENDERFORMAT

The **WM_RENDERFORMAT** message is sent to the clipboard owner if it has delayed rendering a specific clipboard format and if an application has requested data in that format. The clipboard owner must render data in the specified format and place it on the clipboard by calling the **SetClipboardData** function.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,        // handle to window
  UINT uMsg,        // WM_RENDERFORMAT
  WPARAM wParam,    // clipboard format (UINT)
  LPARAM lParam     // not used
);
```

Parameters

wParam
Specifies the clipboard format to be rendered.

lParam
This parameter is not used.

Return Values

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

Remarks

When responding to a **WM_RENDERFORMAT** message, the clipboard owner must not open the clipboard before calling **SetClipboardData**.

Requirements

Windows NT/2000: Requires Windows 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.

Clipboard Overview, *Clipboard Messages*, **SetClipboardData**, **WM_RENDERALLFORMATS**

WM_SIZECLIPBOARD

The **WM_SIZECLIPBOARD** message is sent to the clipboard owner by a clipboard viewer window, when the clipboard contains data in the CF_OWNERDISPLAY format and the clipboard viewer's client area has changed size.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_SIZECLIPBOARD
  WPARAM wParam,   // handle to window (HWND)
  LPARAM lParam    // handle to memory object (HGLOBAL)
);
```

Parameters

wParam
Handle to the clipboard viewer window.

lParam
Handle to a global memory object that contains a **RECT** structure. The structure specifies the new dimensions of the clipboard viewer's client area.

Return Values

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

Remarks

When the clipboard viewer window is about to be destroyed or resized, a **WM_SIZECLIPBOARD** message is sent with a null rectangle (0, 0, 0, 0) as the new size. This permits the clipboard owner to free its display resources.

The clipboard owner must use the **GlobalLock** function to lock the memory object that contains **RECT**. Before returning, the clipboard owner must unlock the object by using the **GlobalUnlock** 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

Clipboard Overview, *Clipboard Messages*, **GlobalLock**, **GlobalUnlock**, **RECT**

WM_VSCROLLCLIPBOARD

The **WM_VSCROLLCLIPBOARD** message is sent to the clipboard owner by a clipboard viewer window, when the clipboard contains data in the CF_OWNERDISPLAY format and an event occurs in the clipboard viewer's vertical scroll bar. The owner should scroll the clipboard image and update the scroll bar values.

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

```
LRESULT CALLBACK WindowProc(
  HWND hwnd,       // handle to window
  UINT uMsg,       // WM_VSCROLLCLIPBOARD
  WPARAM wParam,   // handle to window (HWND)
  LPARAM lParam    // scroll bar code and scroll box position
);
```

Parameters

wParam
Handle to the clipboard viewer window.

lParam
The low-order word of *lParam* specifies a scroll bar event. This parameter can be one of the following values:

Value	Meaning
SB_BOTTOM	Scroll to lower right.
SB_ENDSCROLL	End scroll.
SB_LINEDOWN	Scroll one line down.
SB_LINEUP	Scroll one line up.
SB_PAGEDOWN	Scroll one page down.
SB_PAGEUP	Scroll one page up.
SB_THUMBPOSITION	Scroll to absolute position. The current position is specified by the high-order word.
SB_TOP	Scroll to upper left.

The high-order word of *lParam* specifies the current position of the scroll box, if the low-order word of *lParam* is SB_THUMBPOSITION; otherwise, the high-order word of *lParam* is not used.

Return Values

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

Remarks

The clipboard owner can use the **ScrollWindow** function to scroll the image in the clipboard viewer window and invalidate the appropriate region.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

+ See Also

Clipboard Overview, *Clipboard Messages*, **HIWORD**, **LOWORD**, **ScrollWindow**

Handles and Objects

An *object* is a data structure that represents a system resource, such as a file, thread, or graphic image. An application cannot directly access object data or the system resource that an object represents. Instead, an application must obtain an object *handle*, which it can use to examine or modify the system resource. Each handle has an entry in an internally maintained table. These entries contain the addresses of the resources and the means to identify the resource type.

About Handles and Objects

The system uses objects and handles to regulate access to system resources for two reasons. First, the use of objects ensures that Microsoft can update system functionality, as long as the original object interface is maintained. When subsequent versions of the system are released, you can use the updated object with little or no additional work.

Secondly, the use of objects enables you to take advantage of Microsoft Windows NT/Windows 2000 security. Each object has its own access-control list (ACL) that specifies the actions a process can perform on the object. Windows NT/Windows 2000 examines an object's ACL each time an application creates a handle to the object. For more information, see *Access Control*.

Object Manager

An object consists of a standard header and object-specific attributes. Because all objects have the same structure, there is a single *object manager* that maintains all objects.

The object header includes items such as the object name, so that other processes can reference the object by name, and a security descriptor, so that the object manager can control the processes that access the system resource.

The tasks that the object manager performs include the following:

- Creating objects
- Verifying that a process has the right to use the object
- Creating object handles and returning them to the caller
- Maintaining resource quotas
- Creating duplicate handles
- Closing handles to objects

Object Interface

The Microsoft Win32 application programming interface (API) provides functions that perform the following tasks:

- Create an object
- Get an object handle
- Get information about the object
- Set information about the object
- Close the object handle
- Destroy the object

Some of these tasks are not necessary for each object. Some of these tasks are combined for certain objects. For example, an application can create an event object. Other applications can open the event to obtain a unique handle to this event object. As each application finishes using the event, it closes its handle to the object. When there are no remaining open handles to the event object, the system destroys the event object. In contrast, an application can obtain a handle to an existing window object. When the window object is no longer needed, the application must destroy the object, which invalidates the window handle.

Occasionally, an object remains in memory after all object handles have been closed. For example, a thread could create an event object and wait on the event handle. While the thread is waiting, another thread could close the same event object handle. The event object remains in memory, without any event object handles, until the event object is set to the signaled state and the wait operation is completed. At this time, the system removes the object from memory.

Handles and objects consume memory. Therefore, to preserve system performance, you should close handles and delete objects as soon as they are no longer needed. If you do not do this, your application can hurt system performance, due to excessive use of the paging file.

Note When a process terminates, the system automatically closes handles and deletes objects created by the process. However, when a thread terminates, the system usually does not close handles or delete objects. The only exceptions are window, hook, window position, and dynamic data exchange (DDE) conversation objects; these objects are destroyed when the creating thread terminates.

Handle Limitations

Some objects support only one handle at a time. The system provides the handle when an application creates the object, and invalidates the handle when the application destroys the object. Other objects support multiple handles to a single object. The operating system automatically removes the object from memory after the last handle to the object is closed.

The total number of open handles in the system is limited only by the amount of memory available. However, a single process can have no more than 65,536 handles. Some object types support a limited number of handles per process, while other object types support a limited number of handles in the system.

Handle Inheritance

A child process can inherit handles from its parent process. An inherited handle is valid only in the context of the child process. To enable a child process to inherit open handles from its parent process, use the following steps:

1. Create the handle with the **bInheritHandle** member of the **SECURITY_ATTRIBUTES** structure set to TRUE.
2. Create the child process using the **CreateProcess** function, with the *bInheritHandles* parameter set to TRUE.

The **DuplicateHandle** function duplicates a handle to be used either in the current process or in another process. If an application duplicates one of its handles for another process, the duplicated handle is valid only in the context of the other process.

A duplicated or inherited handle is a unique value, but it refers to the same object as the original handle. Processes can inherit or duplicate handles to the following types of objects:

Access Token	Mailslot
Communications device	Mutex
Console input	Pipe
Console screen buffer	Process
Desktop	Registry key
Directory	Semaphore
Event	Socket
File	Thread
File mapping	Timer
Job	Window station

All other objects are private to the process that created them; their object handles cannot be duplicated or inherited.

For more information, see *Inheritance*.

Object Categories

The system provides three categories of objects: user, graphical device interface (GDI), and kernel. The system uses user objects to support window management; GDI objects to support graphics; and kernel objects to support memory management, process execution, and interprocess communications (IPC). For information about creating and using a specific object, refer to the associated overview:

User object	Overview
Accelerator table	*Keyboard Accelerators*
Caret	*Carets*
Cursor	*Cursors*
DDE conversation	*Dynamic Data Exchange Management Library*
Desktop	*Window Stations and Desktops*
Hook	*Hooks*
Icon	*Icons*
Menu	*Menus*
Window	*Windows*
Window position	*Windows*
Window station	*Window Stations and Desktops*

GDI object	Overview
Bitmap	*Bitmaps*
Brush	*Brushes*

DC	*Device Contexts*
Enhanced metafile	*Metafiles*
Enhanced-metafile DC	*Metafiles*
Font	*Fonts and Text*
Memory DC	*Device Contexts*
Metafile	*Metafiles*
Metafile DC	*Metafiles*
Palette	*Colors*
Pen and extended pen	*Pens*
Region	*Regions*

Kernel object	Overview
Access token	*Access Control*
Change notification	*File I/O*
Communications device	*Communications*
Console input	*Consoles and Character-Mode Support*
Console screen buffer	*Consoles and Character-Mode Support*
Event	*Synchronization*
Event log	*Event Logging*
File	*File I/O*
File mapping	*File Mapping*
Find file	*File I/O*
Heap	*Memory Management*
Job	*Job Objects*
Mailslot	*Mailslots*
Module	*Dynamic-Link Libraries*
Mutex	*Synchronization*
Pipe	*Pipes*
Process	*Processes and Threads*
Semaphore	*Synchronization*
Socket	*Windows Sockets 2*
Thread	*Processes and Threads*
Timer	*Synchronization*
Update resource	*Resources*

User Objects

User objects support only one handle per object. Processes cannot inherit or duplicate handles to user objects. There is no per-process limit on user handles, but there is a system-wide limit of 65,536 user handles.

Handles to user objects are public to all processes. That is, any process can use the user object handle, provided that the process has security access to the object.

In Figure 8-1, an application creates a window object. The **CreateWindow** function creates the window object and returns an object handle.

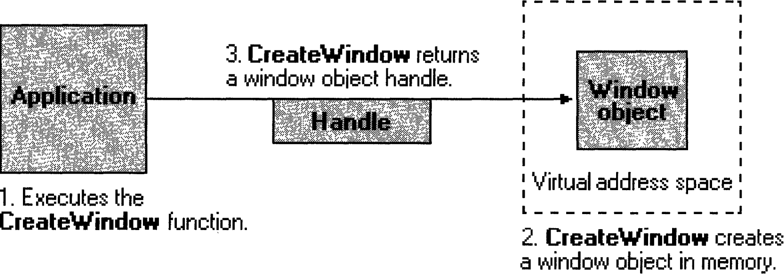


Figure 8-1: Creating a window object.

After the window object has been created, the application can use the window handle to display or change the window. The handle remains valid until the window object is destroyed.

In Figure 8-2, the application destroys the window object. The **DestroyWindow** function removes the window object from memory, which invalidates the window handle.

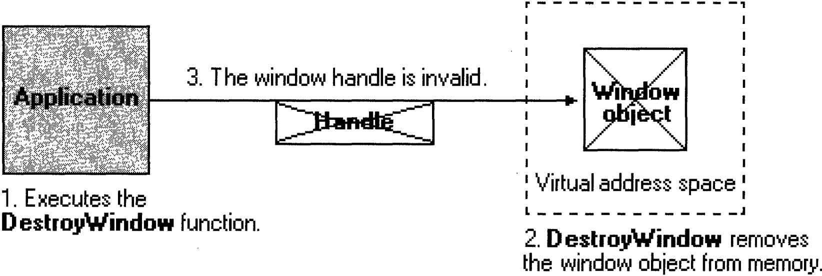


Figure 8-2: Destroying a window object.

The following table lists the user objects, along with each object's creator and destroyer functions. The creator functions either create the object and an object handle or simply return the existing object handle. The destroyer functions remove the object from memory, which invalidates the object handle:

User Objects

Object	Creator function	Destroyer function
Accelerator table	**CreateAcceleratorTable**	**DestroyAcceleratorTable**
Caret	**CreateCaret**	**DestroyCaret**

Cursor	**CreateCursor**, **LoadCursor**, **LoadImage**	**DestroyCursor**
DDE conversation	**DdeConnect**, **DdeConnectList**	**DdeDisconnect**, **DdeDisconnectList**
Desktop	**GetThreadDesktop**	Applications cannot delete this object.
Hook	**SetWindowsHookEx**	**UnhookWindowsHookEx**
Icon	**CreateIconIndirect**, **LoadIcon**, **LoadImage**	**DestroyIcon**
Menu	**CreateMenu**, **CreatePopupMenu**, **LoadMenu**, **LoadMenuIndirect**	**DestroyMenu**
Window	**CreateWindow**, **CreateWindowEx**, **CreateDialogParam**, **CreateDialogIndirectParam**, **CreateMDIWindow**	**DestroyWindow**
Window position	**BeginDeferWindowPos**	**EndDeferWindowPos**
Window station	**GetProcessWindowStation**	Applications cannot delete this object.

GDI Objects

GDI objects support only one handle per object. Handles to GDI objects are private to a process. That is, only the process that created the GDI object can use the object handle. A single process may have no more than 16,384 open GDI object handles.

The following table lists the GDI objects, along with each object's creator and destroyer functions. The creator functions either create the object and an object handle or simply return the existing object handle. The destroyer functions remove the object from memory, which invalidates the object handle.

GDI Objects

Object	Creator function	Destroyer function
Bitmap	**CreateBitmap**, **CreateBitmapIndirect**, **CreateCompatibleBitmap**, **CreateDIBitmap**, **CreateDIBSection**, **CreateDiscardableBitmap**	**DeleteObject**

(continued)

(continued)

Object	Creator function	Destroyer function
Brush	**CreateBrushIndirect**, **CreateDIBPatternBrush**, **CreateDIBPatternBrushPt**, **CreateHatchBrush**, **CreatePatternBrush**, **CreateSolidBrush**	**DeleteObject**
DC	**CreateDC**	**DeleteDC**, **ReleaseDC**
Enhanced metafile	**CreateEnhMetaFile**	**DeleteEnhMetaFile**
Enhanced-metafile DC	**CreateEnhMetaFile**	**CloseEnhMetaFile**
Font	**CreateFont**, **CreateFontIndirect**	**DeleteObject**
Memory DC	**CreateCompatibleDC**	**DeleteDC**
Metafile	**CreateMetaFile**	**DeleteMetaFile**
Metafile DC	**CreateMetaFile**	**CloseMetaFile**
Palette	**CreatePalette**	**DeleteObject**
Pen and extended pen	**CreatePen**, **CreatePenIndirect**, **ExtCreatePen**	**DeleteObject**
Region	**CombineRgn**, **CreateEllipticRgn**, **CreateEllipticRgnIndirect**, **CreatePolygonRgn**, **CreatePolyPolygonRgn**, **CreateRectRgn**, **CreateRectRgnIndirect**, **CreateRoundRectRgn**, **ExtCreateRegion**, **PathToRegion**	**DeleteObject**

Kernel Objects

Kernel object handles are process specific. That is, a process must either create the object or open an existing object to obtain a kernel object handle.

The per-process limit on kernel handles is 2^30.

Any process can create a new handle to an existing kernel object (even one created by another process), provided that the process knows the name of the object and has security access to the object. Kernel object handles include access rights that indicate the actions that can be granted or denied to a process. An application specifies access rights when it creates an object or obtains an existing object handle. Each type of kernel object supports its own set of access rights. For example, event handles can have "set" or "wait" access (or both), file handles can have "read" or "write" access (or both), and so on. For more information, see *Securable Objects*.

In Figure 8-3, an application creates an event object. The **CreateEvent** function creates the event object and returns an object handle.

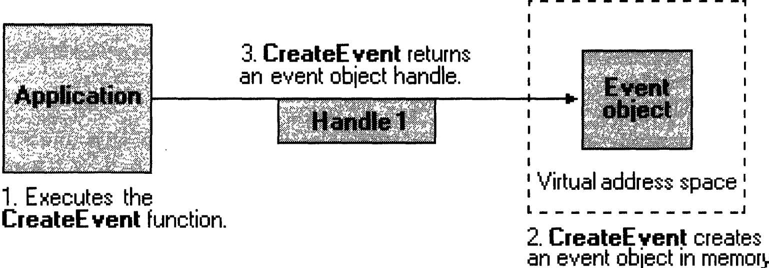


Figure 8-3: Creating an event object.

After the event object has been created, the application can use the event handle to set or wait on the event. The handle remains valid until the application closes the handle or terminates.

Most kernel objects support multiple handles to a single object. For example, the application in Figure 8-3 could obtain additional event object handles by using the **OpenEvent** function, as shown in Figure 8-4.

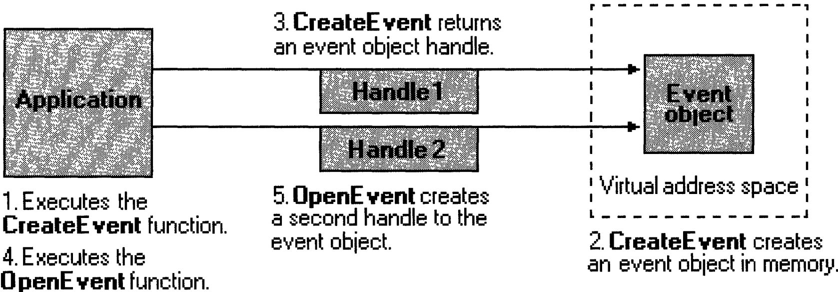


Figure 8-4: Obtaining additional event object handles.

This method enables an application to have handles with different access rights. For example, Handle 1 might have “set” and “wait” access to the event, and Handle 2 might have only “wait” access.

If another process knows the event name and has security access to the object, it can create its own event object handle by using **OpenEvent**. The creating application could also duplicate one of its handles into the same process, or into another process, by using the **DuplicateHandle** function.

An object remains in memory as long as at least one object handle exists. In the following illustration, the applications use the **CloseHandle** function to close their event object handles. When there are no event handles, the system removes the object from memory, as shown in Figure 8-5.

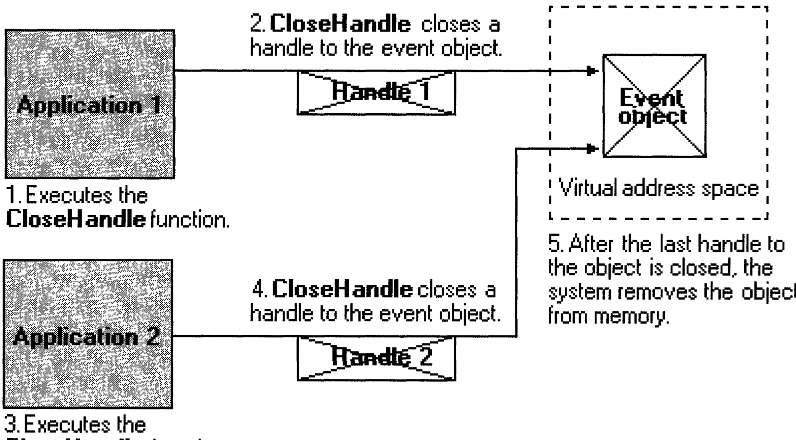


Figure 8-5: Closing event object handles.

The system manages file objects somewhat differently from other kernel objects. File objects contain the file pointer—the pointer to the next byte to be read or written in a file. Whenever an application creates a new file handle, the system creates a new file object. Therefore, more than one file object can refer to a single file on disk, as shown in Figure 8-6.

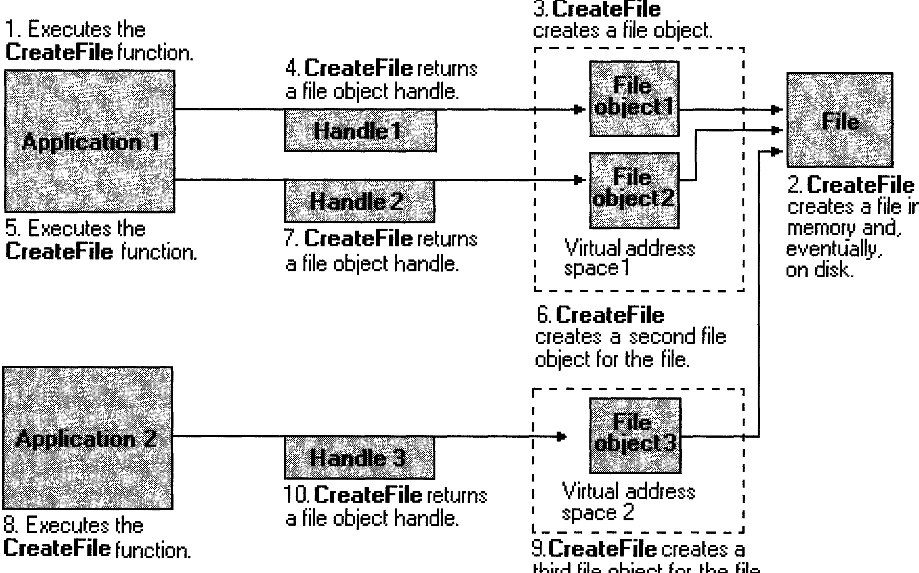


Figure 8-6: Multiple file objects referring to a single file on disk.

Only through duplication or inheritance can more than one file handle refer to the same file object, as shown in Figure 8-7.

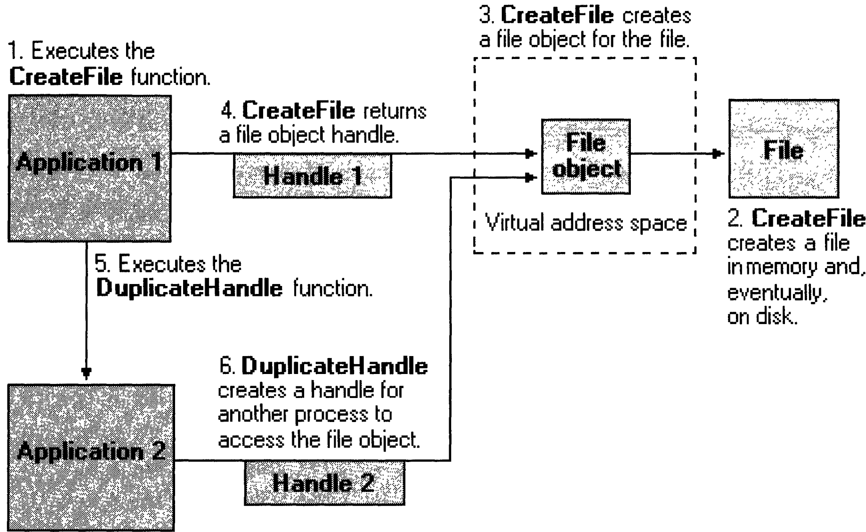


Figure 8-7: Multiple file handles referring to the same file object.

The following table lists each of the kernel objects, along with each object's creator and destroyer functions. The creator functions either create the object and an object handle or create a new existing object handle. The destroyer functions close the object handle. When an application closes the last handle to a kernel object, the system removes the object from memory:

Kernel Objects

Object	Creator function	Destroyer function
Access token	**CreateRestrictedToken**, **DuplicateToken**, **DuplicateTokenEx**, **OpenProcessToken**, **OpenThreadToken**	**CloseHandle**
Change notification	**FindFirstChangeNotification**	**FindCloseChangeNotification**
Communications device	**CreateFile**	**CloseHandle**
Console input	**CreateFile**, with CONIN$	**CloseHandle**
Console screen buffer	**CreateFile**, with CONOUT$	**CloseHandle**
Event	**CreateEvent**, **OpenEvent**	**CloseHandle**
Event log	**OpenBackupEventLog**, **OpenEventLog**, **RegisterEventSource**	**CloseEventLog**
File	**CreateFile**	**CloseHandle**, **DeleteFile**

(continued)

(continued)

Object	Creator function	Destroyer function
File mapping	**CreateFileMapping**, **OpenFileMapping**	**CloseHandle**
Find file	**FindFirstFile**	**FindClose**
Heap	**HeapCreate**	**HeapDestroy**
Job	**CreateJobObject**	**CloseHandle**
Mailslot	**CreateMailslot**	**CloseHandle**
Module	**GetModuleHandle**, **LoadLibrary**	**FreeLibrary**
Mutex	**CreateMutex**, **OpenMutex**	**CloseHandle**
Pipe	**CreateNamedPipe**, **CreatePipe**	**CloseHandle**, **DisconnectNamedPipe**
Process	**CreateProcess**, **GetCurrentProcess**, **OpenProcess**	**CloseHandle**, **TerminateProcess**
Semaphore	**CreateSemaphore**, **OpenSemaphore**	**CloseHandle**
Socket	**socket**, **accept**	**CloseHandle**
Thread	**CreateRemoteThread**, **CreateThread**, **GetCurrentThread**	**CloseHandle**, **TerminateThread**
Timer	**CreateWaitableTimer**, **OpenWaitableTimer**	**CloseHandle**
Update resource	**BeginUpdateResource**	**EndUpdateResource**

Handle and Object Reference

Handle and Object Functions

CloseHandle

The **CloseHandle** function closes an open object handle.

```
BOOL CloseHandle(
  HANDLE hObject   // handle to object
);
```

Parameters

hObject

[in/out] Handle to an open object.

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**.

Windows NT/2000: Closing an invalid handle raises an exception when the application is running under a debugger. This includes closing a handle twice, and using **CloseHandle** on a handle returned by the **FindFirstFile** function.

Remarks

The **CloseHandle** function closes handles to the following objects:

Access token
Communications device
Console input
Console screen buffer
Event
File
File mapping
Job
Mailslot
Mutex
Named pipe
Process
Semaphore
Socket
Thread

CloseHandle invalidates the specified object handle, decrements the object's handle count, and performs object retention checks. After the last handle to an object is closed, the object is removed from the system.

Closing a thread handle does not terminate the associated thread. To remove a thread object, you must terminate the thread, then close all handles to the thread.

Use **CloseHandle** to close handles returned by calls to the **CreateFile** function. Use **FindClose** to close handles returned by calls to **FindFirstFile**.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Handles and Objects Overview, *Handle and Object Functions*, **CreateFile**, **DeleteFile**, **FindClose**, **FindFirstFile**

DuplicateHandle

The **DuplicateHandle** function duplicates an object handle. The duplicate handle refers to the same object as the original handle. Therefore, any changes to the object are reflected through both handles. For example, the current file mark for a file handle is always the same for both handles.

```
BOOL DuplicateHandle(
  HANDLE hSourceProcessHandle,  // handle to source process
  HANDLE hSourceHandle,         // handle to duplicate
  HANDLE hTargetProcessHandle,  // handle to target process
  LPHANDLE lpTargetHandle,      // duplicate handle
  DWORD dwDesiredAccess,        // requested access
  BOOL bInheritHandle,          // handle inheritance option
  DWORD dwOptions               // optional actions
);
```

Parameters

hSourceProcessHandle

[in] Handle to the process with the handle to duplicate.

Windows NT/2000: The handle must have PROCESS_DUP_HANDLE access. For more information, see *Process Security and Access Rights*.

hSourceHandle

[in] Handle to duplicate. This is an open object handle that is valid in the context of the source process. For a list of objects whose handles can be duplicated, see the following Remarks section.

hTargetProcessHandle

[in] Handle to the process that is to receive the duplicated handle. The handle must have PROCESS_DUP_HANDLE access.

lpTargetHandle

[out] Pointer to a variable that receives the value of the duplicate handle. This handle value is valid in the context of the target process.

If *lpTargetHandle* is NULL, the function duplicates the handle, but does not return the duplicate handle value to the caller. This behavior exists only for backward compatibility with previous versions of this function. You should not use this feature, as you will lose system resources until the target process terminates.

dwDesiredAccess

[in] Specifies the access requested for the new handle. This parameter is ignored if the *dwOptions* parameter specifies the DUPLICATE_SAME_ACCESS flag. Otherwise, the flags that can be specified depend on the type of object whose handle is being duplicated. For the flags that can be specified for each object type, see the following Remarks section. Note that the new handle can have more access than the original handle.

bInheritHandle

[in] Indicates whether the handle is inheritable. If TRUE, the duplicate handle can be inherited by new processes created by the target process. If FALSE, the new handle cannot be inherited.

dwOptions

[in] Specifies optional actions. This parameter can be zero, or any combination of the following values:

Value	Meaning
DUPLICATE_CLOSE_SOURCE	Closes the source handle. This occurs regardless of any error status returned.
DUPLICATE_SAME_ACCESS	Ignores the *dwDesiredAccess* parameter. The duplicate handle has the same access as the source handle.

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

DuplicateHandle can be called by either the source process or the target process. It also can be invoked where the source and target process are the same. For example, a process can use **DuplicateHandle** to create a noninheritable duplicate of an inheritable handle, or a handle with different access than the original handle.

The duplicating process uses the **GetCurrentProcess** function to get a handle of itself. To get the other process handle, it might be necessary to use a form of interprocess communication (for example, named pipe or shared memory) to communicate the process identifier to the duplicating process. This identifier is used then in the **OpenProcess** function to open a handle.

If the process that calls **DuplicateHandle** is not the target process, the duplicating process must use interprocess communication to pass the value of the duplicate handle to the target process.

DuplicateHandle can duplicate handles to the following types of objects:

Object	Description
Access token	The handle is returned by the **CreateRestrictedToken**, **DuplicateToken**, **DuplicateTokenEx**, **OpenProcessToken**, or **OpenThreadToken** function.
Communications device	The handle is returned by the **CreateFile** function.

(continued)

(continued)

Object	Description
Console input	The handle is returned by the **CreateFile** function when CONIN$ is specified, or by the **GetStdHandle** function when STD_INPUT_HANDLE is specified. Console handles can be duplicated for use only in the same process.
Console screen buffer	The handle is returned by the **CreateFile** function when CONOUT$ is specified, or by the **GetStdHandle** function when STD_OUTPUT_HANDLE is specified. Console handles can be duplicated for use only in the same process.
Desktop	The handle is returned by the **GetThreadDesktop** function.
Directory	The handle is returned by the **CreateDirectory** function.
Event	The handle is returned by the **CreateEvent** or **OpenEvent** function.
File	The handle is returned by the **CreateFile** function.
File mapping	The handle is returned by the **CreateFileMapping** function.
Job	The handle is returned by the **CreateJobObject** function.
Mailslot	The handle is returned by the **CreateMailslot** function.
Mutex	The handle is returned by the **CreateMutex** or **OpenMutex** function.
Pipe	A named pipe handle is returned by the **CreateNamedPipe** or **CreateFile** function. An anonymous pipe handle is returned by the **CreatePipe** function.
Process	The handle is returned by the **CreateProcess**, **GetCurrentProcess**, or **OpenProcess** function.
Registry key	**Windows NT/2000:** The handle is returned by the **RegCreateKey**, **RegCreateKeyEx**, **RegOpenKey**, or **RegOpenKeyEx** function. Note that registry key handles returned by the **RegConnectRegistry** function cannot be used in a call to **DuplicateHandle**. **Windows 95/98:** You cannot use **DuplicateHandle** to duplicate registry key handles.
Semaphore	The handle is returned by the **CreateSemaphore** or **OpenSemaphore** function.
Socket	The handle is returned by the **socket** or **accept** function.
Thread	The handle is returned by the **CreateProcess**, **CreateThread**, **CreateRemoteThread**, or **GetCurrentThread** function.
Timer	The handle is returned by the **CreateWaitableTimer** or **OpenWaitableTimer** function.
Window station	The handle is returned by the **GetProcessWindowStation** function.

Note that **DuplicateHandle** should not be used to duplicate handles to I/O completion ports. In this case, no error is returned, but the duplicate handle cannot be used.

In addition to STANDARD_RIGHTS_REQUIRED, the following access flags can be specified in the *dwDesiredAccess* parameter for the different object types. Note that the new handle can have more access than the original handle. However, in some cases, **DuplicateHandle** cannot create a duplicate handle with more access permission than the original handle. For example, a file handle created with GENERIC_READ access cannot be duplicated so that it has both GENERIC_READ and GENERIC_WRITE access.

Any combination of the following access flags is valid for handles to communications devices, console input, console screen buffers, files, and pipes:

Access	Description
GENERIC_READ	Enables read access.
GENERIC_WRITE	Enables write access.

Any combination of the following access flags is valid for file-mapping objects:

Access	Description
FILE_MAP_ALL_ACCESS	Specifies all access flags possible for the file-mapping object.
FILE_MAP_READ	Enables mapping the object into memory that permits read access.
FILE_MAP_WRITE	Enables mapping the object into memory that permits write access. For write access, PAGE_READWRITE protection must have been specified when the file-mapping object was created by the **CreateFileMapping** function.

Any combination of the following flags is valid for mutex objects:

Access	Description
MUTEX_ALL_ACCESS	Specifies all access flags possible for the mutex object.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the mutex handle in any of the wait functions to acquire ownership of the mutex, or in the **ReleaseMutex** function to release ownership.

Any combination of the following access flags is valid for semaphore objects:

Access	Description
SEMAPHORE_ALL_ACCESS	Specifies all access flags possible for the semaphore object.

(continued)

(continued)

Access	Description
SEMAPHORE_MODIFY_STATE	Enables use of the semaphore handle in the **ReleaseSemaphore** function to modify the semaphore's count.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the semaphore handle in any of the wait functions to wait for the semaphore's state to be signaled.

Any combination of the following access flags is valid for event objects:

Access	Description
EVENT_ALL_ACCESS	Specifies all access flags possible for the event object.
EVENT_MODIFY_STATE	Enables use of the event handle in the **SetEvent** and **ResetEvent** functions to modify the event's state.
SYNCHRONIZE	**Windows NT/2000:** Enables use of the event handle in any of the wait functions to wait for the event's state to be signaled.

Any combination of the following access flags is valid for handles to registry keys:

Value	Meaning
KEY_ALL_ACCESS	Specifies all possible flags for the registry key.
KEY_CREATE_LINK	Enables using the handle to create a link to a registry-key object.
KEY_CREATE_SUB_KEY	Enables using the handle to create a subkey of a registry-key object.
KEY_ENUMERATE_SUB_KEYS	Enables using the handle to enumerate the subkeys of a registry-key object.
KEY_EXECUTE	Equivalent to KEY_READ.
KEY_NOTIFY	Enables using the handle to request change notifications for either a registry key or subkeys of a registry key.
KEY_QUERY_VALUE	Enables using the handle to query a value of a registry-key object.
KEY_READ	Combines the STANDARD_RIGHTS_READ, KEY_QUERY_VALUE, KEY_ENUMERATE_SUB_KEYS, and KEY_NOTIFY values.

(continued)

(continued)

Value	Meaning
KEY_SET_VALUE	Enables using the handle to create or set a value of a registry-key object.
KEY_WRITE	Combines the STANDARD_RIGHTS_WRITE, KEY_SET_VALUE, and KEY_CREATE_SUB_KEY values.

Any combination of the following access flags is valid for process objects:

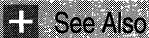
Access	Description
PROCESS_ALL_ACCESS	Specifies all possible access flags for the process object.
PROCESS_CREATE_PROCESS	Used internally.
PROCESS_CREATE_THREAD	Enables using the process handle in the **CreateRemoteThread** function to create a thread in the process.
PROCESS_DUP_HANDLE	Enables using the process handle as either the source or target process in the **DuplicateHandle** function to duplicate a handle.
PROCESS_QUERY_INFORMATION	Enables using the process handle in the **GetExitCodeProcess** and **GetPriorityClass** functions to read information from the process object.
PROCESS_SET_INFORMATION	Enables using the process handle in the **SetPriorityClass** function to set the process's priority class.
PROCESS_TERMINATE	Enables using the process handle in the **TerminateProcess** function to terminate the process.
PROCESS_VM_OPERATION	Enables using the process handle in the **VirtualProtectEx** and **WriteProcessMemory** functions to modify the virtual memory of the process.
PROCESS_VM_READ	Enables using the process handle in the **ReadProcessMemory** function to read from the virtual memory of the process.
PROCESS_VM_WRITE	Enables using the process handle in the **WriteProcessMemory** function to write to the virtual memory of the process.
SYNCHRONIZE	**Windows NT/2000:** Enables using the process handle in any of the wait functions to wait for the process to terminate.

Any combination of the following access flags is valid for thread objects:

Access	Description
SYNCHRONIZE	**Windows NT/2000:** Enables using the thread handle in any of the wait functions to wait for the thread to terminate.
THREAD_ALL_ACCESS	Specifies all possible access flags for the thread object.
THREAD_DIRECT_IMPERSONATION	Used internally.
THREAD_GET_CONTEXT	Enables using the thread handle in the **GetThreadContext** function to read the thread's context.
THREAD_IMPERSONATE	Used internally.
THREAD_QUERY_INFORMATION	Enables using the thread handle in the **GetExitCodeThread**, **GetThreadPriority**, and **GetThreadSelectorEntry** functions to read information from the thread object.
THREAD_SET_CONTEXT	Enables using the thread handle in the **SetThreadContext** function to set the thread's context.
THREAD_SET_INFORMATION	Enables using the thread handle in the **SetThreadPriority** function to set the thread's priority.
THREAD_SET_THREAD_TOKEN	Used internally.
THREAD_SUSPEND_RESUME	Enables using the thread handle in the **SuspendThread** or **ResumeThread** functions to suspend or resume a thread.
THREAD_TERMINATE	Enables using the thread handle in the **TerminateThread** function to terminate the 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 winbase.h; include windows.h.
Library: Use kernel32.lib.



See Also

Handles and Objects Overview, *Handle and Object Functions*, **CloseHandle**

GetHandleInformation

The **GetHandleInformation** function obtains information about certain properties of an object handle. The information is obtained as a set of bit flags.

```
BOOL GetHandleInformation(
  HANDLE hObject,      // handle to object
  LPDWORD lpdwFlags    // handle properties
);
```

Parameters

hObject

[in] Specifies a handle to an object. The **GetHandleInformation** function obtains information about this object handle.

You can specify a handle to one of the following types of objects: access token, event, file, file mapping, job, mailslot, mutex, pipe, printer, process, registry key, semaphore, serial communication device, socket, thread, or waitable timer.

Windows 2000: This parameter also can be a handle to a console input buffer or a console screen buffer.

lpdwFlags

[out] Pointer to a variable that receives a set of bit flags that specify properties of the object handle. The following flags are defined:

Value	Meaning
HANDLE_FLAG_INHERIT	If this flag is set, a child process created with the *bInheritHandles* parameter of **CreateProcess** set to TRUE will inherit the object handle.
HANDLE_FLAG_PROTECT_FROM_CLOSE	If this flag is set, calling the **CloseHandle** function will not close the object handle.

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 3.51 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

Handles and Objects Overview, *Handle and Object Functions*, **CloseHandle**, **CreateProcess**, **SetHandleInformation**

SetHandleInformation

The **SetHandleInformation** function sets certain properties of an object handle. The information is specified as a set of bit flags.

```
BOOL SetHandleInformation(
  HANDLE hObject,  // handle to object
  DWORD dwMask,    // flags to change
  DWORD dwFlags    // new values for flags
);
```

Parameters

hObject

[in] Handle to an object. The **SetHandleInformation** function sets information associated with this object handle.

You can specify a handle to one of the following types of objects: access token, event, file, file mapping, job, mailslot, mutex, pipe, printer, process, registry key, semaphore, serial communication device, socket, thread, or waitable timer.

Windows 2000: This parameter also can be a handle to a console input buffer or a console screen buffer.

dwMask

[in] A mask that specifies the bit flags to be changed. Use the same flag constants shown in the description of *dwFlags*.

dwFlags

[in] A set of bit flags that specify properties of the object handle. This parameter can be one of the following values:

Value	Meaning
HANDLE_FLAG_INHERIT	If this flag is set, a child process created with the *bInheritHandles* parameter of **CreateProcess** set to TRUE will inherit the object handle.
HANDLE_FLAG_PROTECT_FROM_CLOSE	If this flag is set, calling the **CloseHandle** function will not close the object handle.

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 or clear the associated bit flag in *dwFlags*, you must set a change mask bit flag in *dwMask*.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Handles and Objects Overview, *Handle and Object Functions*, **CreateProcess**, **CloseHandle**, **GetHandleInformation**

Hooks

A *hook* is a point in the system message-handling mechanism where an application can install a subroutine to monitor the message traffic in the system and process certain types of messages before they reach the target window procedure.

About Hooks

Hooks tend to slow down the system because they increase the amount of processing the system must perform for each message. You should install a hook only when necessary, and remove it as soon as possible.

Hook Chains

The system supports many different types of hook; each type provides access to a different aspect of its message-handling mechanism. For example, an application can use the WH_MOUSE hook to monitor the message traffic for mouse messages.

The system maintains a separate hook chain for each type of hook. A *hook chain* is a list of pointers to special, application-defined callback functions called *hook procedures*. When a message occurs that is associated with a particular type of hook, the system passes the message to each hook procedure referenced in the hook chain, one after the other. The action a hook procedure can take depends on the type of hook involved. The hook procedures for some types of hooks can only monitor messages; others can modify messages or stop their progress though the chain, preventing them from reaching the next hook procedure or the destination window.

Hook Procedures

To take advantage of a particular type of hook, the developer provides a hook procedure and uses the **SetWindowsHookEx** function to install it into the chain associated with the hook. A hook procedure must have the following syntax.

```
LRESULT CALLBACK HookProc(
  int nCode,
  WPARAM wParam,
  LPARAM lParam
);
```

HookProc is a placeholder for an application-defined name.

The *nCode* parameter is a hook code that the hook procedure uses to determine the action to perform. The value of the hook code depends on the type of the hook; each type has its own characteristic set of hook codes. The values of the *wParam* and *lParam* parameters depend on the hook code, but they typically contain information about a message that was sent or posted.

The **SetWindowsHookEx** function always installs a hook procedure at the beginning of a hook chain. When an event occurs that is monitored by a particular type of hook, the system calls the procedure at the beginning of the hook chain associated with the hook. Each hook procedure in the chain determines whether to pass the event to the next procedure. A hook procedure passes an event to the next procedure by calling the **CallNextHookEx** function.

Note that the hook procedures for some types of hooks can only monitor messages. The system passes messages to each hook procedure, regardless of whether a particular procedure calls **CallNextHookEx**.

A *global hook* monitors messages for all threads in the same desktop as the calling thread. A *thread-specific hook* monitors messages for only an individual thread. A global hook procedure can be called in the context of any application in the same desktop as the calling thread, so the procedure must be in a separate dynamic link library (DLL) module. A thread-specific hook procedure is called only in the context of the associated thread. If an application installs a hook procedure for one of its own threads, the hook procedure can be either in the same module as the rest of the application's code or a DLL. If the application installs a hook procedure for a thread of a different application, the procedure must be in a DLL. For information, see *Dynamic-Link Libraries*.

Note You should use global hooks only for debugging purposes; otherwise, you should avoid them. Global hooks hurt system performance and cause conflicts with other applications that implement the same type of global hook.

Hook Types

Each type of hook enables an application to monitor a different aspect of the system's message-handling mechanism. The available hooks are described in this section:

WH_CALLWNDPROC and WH_CALLWNDPROCRET
WH_CBT
WH_DEBUG
WH_FOREGROUNDIDLE
WH_GETMESSAGE
WH_JOURNALPLAYBACK
WH_JOURNALRECORD
WH_KEYBOARD
WH_KEYBOARD_LL
WH_MOUSE
WH_MOUSE_LL
WH_MSGFILTER and WH_SYSMSGFILTER
WH_SHELL

WH_CALLWNDPROC and WH_CALLWNDPROCRET Hooks

The WH_CALLWNDPROC and WH_CALLWNDPROCRET hooks enable you to monitor messages sent to window procedures. The system calls a WH_CALLWNDPROC hook procedure before passing the message to the receiving window procedure, and calls the WH_CALLWNDPROCRET hook procedure after the window procedure has processed the message.

The WH_CALLWNDPROCRET hook passes a pointer to a **CWPRETSTRUCT** structure to the hook procedure. The structure contains the return value from the window procedure that processed the message, as well as the message parameters associated with the message. Subclassing the window does not work for messages set between processes.

For more information, see the **CallWndProc** and **CallWndRetProc** functions.

WH_CBT Hook

The system calls a WH_CBT hook procedure before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the input focus; or before synchronizing with the system message queue. The value the hook procedure returns determines whether the system allows or prevents one of these operations. The WH_CBT hook is intended primarily for computer-based training (CBT) applications.

For more information, see the **CBTProc** function.

For information, see *WinEvents*.

WH_DEBUG Hook

The system calls a WH_DEBUG hook procedure before calling hook procedures associated with any other hook in the system. You can use this hook to determine whether to allow the system to call hook procedures associated with other types of hooks.

For more information, see the **DebugProc** function.

WH_FOREGROUNDIDLE Hook

The WH_FOREGROUNDIDLE hook enables you to perform low priority tasks during times when its foreground thread is idle. The system calls a WH_FOREGROUNDIDLE hook procedure when the application's foreground thread is about to become idle.

For more information, see the **ForegroundIdleProc** function.

WH_GETMESSAGE Hook

The WH_GETMESSAGE hook enables an application to monitor messages about to be returned by the **GetMessage** or **PeekMessage** function. You can use the WH_GETMESSAGE hook to monitor mouse and keyboard input and other messages posted to the message queue.

For more information, see the **GetMsgProc** function.

WH_JOURNALPLAYBACK Hook

The WH_JOURNALPLAYBACK hook enables an application to insert messages into the system message queue. You can use this hook to play back a series of mouse and keyboard events recorded earlier by using the WH_JOURNALRECORD hook. Regular mouse and keyboard input is disabled as long as a WH_JOURNALPLAYBACK hook is installed. A WH_JOURNALPLAYBACK hook is a global hook—it cannot be used as a thread-specific hook.

The WH_JOURNALPLAYBACK hook returns a time-out value. This value tells the system how many milliseconds to wait before processing the current message from the playback hook. This enables the hook to control the timing of the events it plays back.

For more information, see the **JournalPlaybackProc** function.

WH_JOURNALRECORD Hook

The WH_JOURNALRECORD hook enables you to monitor and record input events. Typically, you use this hook to record a sequence of mouse and keyboard events to play back later by using the WH_JOURNALPLAYBACK hook. The WH_JOURNALRECORD hook is a global hook—it cannot be used as a thread-specific hook.

For more information, see the **JournalRecordProc** function.

WH_KEYBOARD Hook

The WH_KEYBOARD hook enables an application to monitor message traffic for **WM_KEYDOWN** and **WM_KEYUP** messages about to be returned by the **GetMessage**

or **PeekMessage** function. You can use the WH_KEYBOARD hook to monitor keyboard input posted to a message queue.

For more information, see the **KeyboardProc** function.

WH_KEYBOARD_LL Hook

The WH_KEYBOARD_LL hook enables you to monitor keyboard input events about to be posted in a thread input queue.

For more information, see the **LowLevelKeyboardProc** function.

WH_MOUSE Hook

The WH_MOUSE hook enables you to monitor mouse messages about to be returned by the **GetMessage** or **PeekMessage** function. You can use the WH_MOUSE hook to monitor mouse input posted to a message queue.

For more information, see the **MouseProc** function.

WH_MOUSE_LL Hook

The WH_MOUSE_LL hook enables you to monitor mouse input events about to be posted in a thread input queue.

For more information, see the **LowLevelMouseProc** function.

WH_MSGFILTER and WH_SYSMSGFILTER Hooks

The WH_MSGFILTER and WH_SYSMSGFILTER hooks enable you to monitor messages about to be processed by a menu, scroll bar, message box, or dialog box, and to detect when a different window is about to be activated as a result of the user's pressing the ALT+TAB or ALT+ESC key combination. The WH_MSGFILTER hook can monitor only messages passed to a menu, scroll bar, message box, or dialog box created by the application that installed the hook procedure. The WH_SYSMSGFILTER hook monitors such messages for all applications.

The WH_MSGFILTER and WH_SYSMSGFILTER hooks enable you to perform message filtering during modal loops that is equivalent to the filtering done in the main message loop. For example, an application often examines a new message in the main loop between the time it retrieves the message from the queue and the time it dispatches the message, performing special processing as appropriate. However, during a modal loop, the system retrieves and dispatches messages without allowing an application the chance to filter the messages in its main message loop. If an application installs a WH_MSGFILTER or WH_SYSMSGFILTER hook procedure, the system calls the procedure during the modal loop.

An application can call the WH_MSGFILTER hook directly by calling the **CallMsgFilter** function. By using this function, the application can use the same code to filter messages during modal loops as it uses in the main message loop. To do so, encapsulate the filtering operations in a WH_MSGFILTER hook procedure, and call **CallMsgFilter** between the calls to the **GetMessage** and **DispatchMessage** functions.

```
while (GetMessage(&msg, (HWND) NULL, 0, 0))
{
    if (!CallMsgFilter(&qmsg, 0))
        DispatchMessage(&qmsg);
}
```

The last argument of **CallMsgFilter** is passed to the hook procedure; you can enter any value. The hook procedure, by defining a constant such as MSGF_MAINLOOP, can use this value to determine from where the procedure was called.

For more information, see the **MessageProc** and **SysMsgProc** functions.

WH_SHELL Hook

A shell application can use the WH_SHELL hook to receive important notifications. The system calls a WH_SHELL hook procedure when the shell application is about to be activated, and when a top-level window is created or destroyed.

For more information, see the **ShellProc** function.

Hook Reference

Hook Functions

CallMsgFilter

The **CallMsgFilter** function passes the specified message and hook code to the hook procedures associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks. A WH_SYSMSGFILTER or WH_MSGFILTER hook procedure is an application-defined callback function that examines and, optionally, modifies messages for a dialog box, message box, menu, or scroll bar.

```
BOOL CallMsgFilter(
  LPMSG lpMsg,  // message data
  int nCode     // hook code
);
```

Parameters

lpMsg
: [in] Pointer to an **MSG** structure that contains the message to be passed to the hook procedures.

nCode
: [in] Specifies an application-defined code used by the hook procedure to determine how to process the message. The code must not have the same value as system-defined hook codes (MSGF_ and HC_) associated with the WH_SYSMSGFILTER and WH_MSGFILTER hooks.

Return Values

If the application should process the message further, the return value is zero.

If the application should not process the message further, the return value is nonzero.

Remarks

The system calls **CallMsgFilter** to enable applications to examine and control the flow of messages during internal processing of dialog boxes, message boxes, menus, and scroll bars, or when the user activates a different window by pressing the ALT+TAB key combination.

Install this hook procedure by using the **SetWindowsHookEx** 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

Hooks Overview, *Hook Functions*, **MessageProc**, **MSG**, **SetWindowsHookEx**, **SysMsgProc**

CallNextHookEx

The **CallNextHookEx** function passes the hook information to the next hook procedure in the current hook chain. A hook procedure can call this function either before or after processing the hook information.

```
LRESULT CallNextHookEx(
  HHOOK hhk,        // handle to current hook
  int nCode,        // hook code passed to hook procedure
  WPARAM wParam,    // value passed to hook procedure
  LPARAM lParam     // value passed to hook procedure
);
```

Parameters

hhk

[in] Handle to the current hook. An application receives this handle as a result of a previous call to the **SetWindowsHookEx** function.

nCode
[in] Specifies the hook code passed to the current hook procedure. The next hook procedure uses this code to determine how to process the hook information.

wParam
[in] Specifies the *wParam* value passed to the current hook procedure. The meaning of this parameter depends on the type of hook associated with the current hook chain.

lParam
[in] Specifies the *lParam* value passed to the current hook procedure. The meaning of this parameter depends on the type of hook associated with the current hook chain.

Return Values

The return value is the value returned by the next hook procedure in the chain. The current hook procedure must also return this value. The meaning of the return value depends on the hook type. For more information, see the descriptions of the individual hook procedures.

Remarks

Hook procedures are installed in chains for particular hook types. **CallNextHookEx** calls the next hook in the chain.

Calling **CallNextHookEx** is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call **CallNextHookEx** unless you absolutely need to prevent the notification from being seen by other 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

Hooks Overview, *Hook Functions*, **SetWindowsHookEx**, **UnhookWindowsHookEx**

CallWndProc

The **CallWndProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function.

Windows 95/98 and Windows NT 3.51: The system calls this function whenever the thread calls the **SendMessage** function. The WH_CALLWNDPROC hook is called in the context of the thread that calls **SendMessage**, not the thread that receives the message.

Windows NT 4.0 and later: The system calls this function before calling the window procedure to process a message sent to the thread.

The **HOOKPROC** type defines a pointer to this callback function. **CallWndProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK CallWndProc(
  int nCode,       // hook code
  WPARAM wParam,   // current-process flag
  LPARAM lParam    // message data
);
```

Parameters

nCode
: [in] Specifies whether the hook procedure must process the message. If *nCode* is HC_ACTION, the hook procedure must process the message. If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and must return the value returned by **CallNextHookEx**.

wParam
: [in] Specifies whether the message was sent by the current thread. If the message was sent by the current thread, it is nonzero; otherwise, it is zero.

lParam
: [in] Pointer to a **CWPSTRUCT** structure that contains details about the message.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_CALLWNDPROC hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure does not call **CallNextHookEx**, the return value should be zero.

Remarks

The **CallWndProc** hook procedure can examine the message, but it cannot modify it. After the hook procedure returns control to the system, the message is passed to the window procedure.

An application installs the hook procedure by specifying the WH_CALLWNDPROC hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** 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.

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **CWPSTRUCT**, **SendMessage**, **SetWindowsHookEx**

CallWndRetProc

The **CallWndRetProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function after the **SendMessage** function is called. The hook procedure can examine the message; it cannot modify it.

The **HOOKPROC** type defines a pointer to this callback function. **CallWndRetProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK CallWndRetProc(
  int nCode,      // hook code
  WPARAM wParam,  // current-process flag
  LPARAM lParam   // message data
);
```

Parameters

nCode
[in] Specifies whether the hook procedure must process the message. If *nCode* is HC_ACTION, the hook procedure must process the message. If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam
[in] Specifies whether the message is sent by the current process. If the message is sent by the current process, it is nonzero; otherwise, it is NULL.

lParam
[in] Pointer to a **CWPRETSTRUCT** structure that contains details about the message.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_CALLWNDPROCRET hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure does not call **CallNextHookEx**, the return value should be zero.

Remarks

An application installs the hook procedure by specifying the WH_CALLWNDPROCRET hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** 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.

See Also

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **CallWndProc**, **CWPRETSTRUCT**, **SendMessage**, **SetWindowsHookEx**

CBTProc

The **CBTProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function before activating, creating, destroying, minimizing, maximizing, moving, or sizing a window; before completing a system command; before removing a mouse or keyboard event from the system message queue; before setting the keyboard focus; or before synchronizing with the system message queue. A computer-based training (CBT) application uses this hook procedure to receive useful notifications from the system.

The **HOOKPROC** type defines a pointer to this callback function. **CBTProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK CBTProc(
  int nCode,      // hook code
  WPARAM wParam,  // depends on hook code
  LPARAM lParam   // depends on hook code
);
```

Parameters

nCode
[in] Specifies a code that the hook procedure uses to determine how to process the message. This parameter can be one of the following values:

Value	Meaning
HCBT_ACTIVATE	The system is about to activate a window.
HCBT_CLICKSKIPPED	The system has removed a mouse message from the system message queue. Upon receiving this hook code, a CBT application must install a WH_JOURNALPLAYBACK hook procedure in response to the mouse message.
HCBT_CREATEWND	A window is about to be created. The system calls the hook procedure before sending the **WM_CREATE** or **WM_NCCREATE** message to the window. If the hook procedure returns a nonzero value, the system destroys the window; the **CreateWindow** function returns NULL, but the **WM_DESTROY** message is not sent to the window. If the hook procedure returns zero, the window is created normally. At the time of the HCBT_CREATEWND notification, the window has been created, but its final size and position may not have been determined and its parent window may not have been established. It is possible to send messages to the newly created window, although it has not yet received **WM_NCCREATE** or **WM_CREATE** messages. It is also possible to change the position in the Z order of the newly created window by modifying the **hwndInsertAfter** member of the **CBT_CREATEWND** structure.
HCBT_DESTROYWND	A window is about to be destroyed.
HCBT_KEYSKIPPED	The system has removed a keyboard message from the system message queue. Upon receiving this hook code, a CBT application must install a WH_JOURNALPLAYBACK hook procedure in response to the keyboard message.
HCBT_MINMAX	A window is about to be minimized or maximized.
HCBT_MOVESIZE	A window is about to be moved or sized.
HCBT_QS	The system has retrieved a **WM_QUEUESYNC** message from the system message queue.
HCBT_SETFOCUS	A window is about to receive the keyboard focus.
HCBT_SYSCOMMAND	A system command is about to be carried out. This allows a CBT application to prevent task switching by means of hot keys.

If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] Depends on the *nCode* parameter. For details, see the following Remarks section.

lParam

[in] Depends on the *nCode* parameter. For details, see the following Remarks section.

Return Values

The value returned by the hook procedure determines whether the system allows or prevents one of these operations. For operations corresponding to the following CBT hook codes, the return value must be 0 to allow the operation, or 1 to prevent it:

HCBT_ACTIVATE
HCBT_CREATEWND
HCBT_DESTROYWND
HCBT_MINMAX
HCBT_MOVESIZE
HCBT_SETFOCUS
HCBT_SYSCOMMAND

For operations corresponding to the following CBT hook codes, the return value is ignored:

HCBT_CLICKSKIPPED
HCBT_KEYSKIPPED
HCBT_QS

Remarks

The hook procedure should not install a WH_JOURNALPLAYBACK hook procedure except in the situations described in the preceding list of hook codes.

This hook procedure must be in a dynamic-link library (DLL). An application installs the hook procedure by specifying the WH_CBT hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

The following table describes the *wParam* and *lParam* parameters for each HCBT_ hook code:

Value	wParam	lParam
HCBT_ACTIVATE	Specifies the handle to the window about to be activated.	Specifies a long pointer to a **CBTACTIVATESTRUCT** structure containing the handle to the active window, and specifies whether the activation is changing because of a mouse click.

(continued)

(continued)

Value	wParam	lParam
HCBT_CLICKSKIPPED	Specifies the mouse message removed from the system message queue.	Specifies a long pointer to a **MOUSEHOOKSTRUCT** structure containing the hit-test code and the handle to the window for which the mouse message is intended. The HCBT_CLICKSKIPPED value is sent to a **CBTProc** hook procedure only if a WH_MOUSE hook is installed. For a list of hit-test codes, see **WM_NCHITTEST**.
HCBT_CREATEWND	Specifies the handle to the new window.	Specifies a long pointer to a **CBT_CREATEWND** structure containing initialization parameters for the window. The parameters include the coordinates and dimensions of the window. By changing these parameters, a **CBTProc** hook procedure can set the initial size and position of the window.
HCBT_DESTROYWND	Specifies the handle to the window about to be destroyed.	Is undefined and must be set to zero.
HCBT_KEYSKIPPED	Specifies the virtual-key code.	Specifies the repeat count, scan code, key-transition code, previous key state, and context code. The HCBT_KEYSKIPPED value is sent to a **CBTProc** hook procedure only if a WH_KEYBOARD hook is installed. For more information, see **WM_KEYUP** or **WM_KEYDOWN**.
HCBT_MINMAX	Specifies the handle to the window being minimized or maximized.	Specifies, in the low-order word, a show-window value (SW_) specifying the operation. For a list of show-window values, see the **ShowWindow**. The high-order word is undefined.
HCBT_MOVESIZE	Specifies the handle to the window to be moved or sized.	Specifies a long pointer to a **RECT** structure containing the coordinates of the window. By changing the values in the structure, a **CBTProc** hook procedure can set the final coordinates of the window.
HCBT_QS	Is undefined and must be zero.	Is undefined and must be zero.

Value	wParam	lParam
HCBT_SETFOCUS	Specifies the handle to the window gaining the keyboard focus.	Specifies the handle to the window losing the keyboard focus.
HCBT_SYSCOMMAND	Specifies a system-command value (SC_) specifying the system command. For more information about system-command values, see **WM_SYSCOMMAND**.	Contains the same data as the *lParam* value of a **WM_SYSCOMMAND** message: If a system menu command is chosen with the mouse, the low-order word contains the x-coordinate of the cursor, in screen coordinates, and the high-order word contains the y-coordinate; otherwise, the parameter is not used.

For information, see *WinEvents*.

Requirements

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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **CreateWindow**, **SetWindowsHookEx**, **WM_SYSCOMMAND**

DebugProc

The **DebugProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function before calling the hook procedures associated with any type of hook. The system passes information about the hook to be called to the **DebugProc** hook procedure, which examines the information and determines whether to allow the hook to be called.

The **HOOKPROC** type defines a pointer to this callback function. **DebugProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK DebugProc(
  int nCode,      // hook code
  WPARAM wParam,  // hook type
  LPARAM lParam   // debugging information
);
```

Parameters

nCode

[in] Specifies whether the hook procedure must process the message. If *nCode* is HC_ACTION, the hook procedure must process the message. If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] Specifies the type of hook about to be called. This parameter can be one of the following values:

Value	Description
WH_CALLWNDPROC	Installs a hook procedure that monitors messages sent to a window procedure. For more information, see the description of the **CallWndProc** hook procedure.
WH_CALLWNDPROCRET	Installs a hook procedure that monitors messages that have just been processed by a window procedure. For more information, see the description of the **CallWndRetProc** hook procedure.
WH_CBT	Installs a hook procedure that receives notifications useful to a computer-based training (CBT) application. For more information, see the description of the **CBTProc** hook procedure.
WH_DEBUG	Installs a hook procedure useful for debugging other hook procedures. For more information, see the description of the **DebugProc** hook procedure.
WH_GETMESSAGE	Installs a hook procedure that monitors messages posted to a message queue. For more information, see the description of the **GetMsgProc** hook procedure.
WH_JOURNALPLAYBACK	Installs a hook procedure that posts messages previously recorded by a WH_JOURNALRECORD hook procedure. For more information, see the description of the **JournalPlaybackProc** hook procedure.
WH_JOURNALRECORD	Installs a hook procedure that records input messages posted to the system message queue. This hook is useful for recording macros. For more information, see the description of the **JournalRecordProc** hook procedure.
WH_KEYBOARD	Installs a hook procedure that monitors keystroke messages. For more information, see the description of the **KeyboardProc** hook procedure.

Value	Description
WH_MOUSE	Installs a hook procedure that monitors mouse messages. For more information, see the description of the **MouseProc** hook procedure.
WH_MSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. The hook procedure monitors these messages only for the application that installed the hook procedure. For more information, see the description of the **MessageProc** hook procedure.
WH_SHELL	Installs a hook procedure that receives notifications useful to a shell application. For more information, see the description of the **ShellProc** hook procedure.
WH_SYSMSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. The hook procedure monitors these messages for all applications in the system. For more information, see the description of the **SysMsgProc** hook procedure.

lParam
[in] Pointer to a **DEBUGHOOKINFO** structure that contains the parameters to be passed to the destination hook procedure.

Return Values

To prevent the system from calling the hook, the hook procedure must return a nonzero value. Otherwise, the hook procedure must call **CallNextHookEx**.

Remarks

An application installs this hook procedure by specifying the WH_DEBUG hook type and the pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

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.

See Also

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **CallWndProc**, **CallWndRetProc**, **CBTProc**, **DEBUGHOOKINFO**, **GetMsgProc**, **JournalPlaybackProc**, **JournalRecordProc**, **KeyboardProc**, **MessageProc**, **MouseProc**, **SetWindowsHookEx**, **ShellProc**, **SysMsgProc**

ForegroundIdleProc

The **ForegroundIdleProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function whenever the foreground thread is about to become idle.

The **HOOKPROC** type defines a pointer to this callback function. **ForegroundIdleProc** is a placeholder for the application-defined or library-defined function name.

```
DWORD CALLBACK ForegroundIdleProc(
  int code,        // hook code
  DWORD wParam,    // not used
  LONG lParam      // not used
);
```

Parameters

code

[in] Specifies whether the hook procedure must process the message. If *code* is HC_ACTION, the hook procedure must process the message. If *code* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

This parameter is not used.

lParam

This parameter is not used.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_FOREGROUNDIDLE hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure does not call **CallNextHookEx**, the return value should be zero.

Remarks

An application installs this hook procedure by specifying the WH_FOREGROUNDIDLE hook type and the pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

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.

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **SetWindowsHookEx**

GetMsgProc

The **GetMsgProc** function is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function whenever the **GetMessage** or **PeekMessage** function has retrieved a message from an application message queue. Before returning the retrieved message to the caller, the system passes the message to the hook procedure.

The **HOOKPROC** type defines a pointer to this callback function. **GetMsgProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK GetMsgProc(
  int code,        // hook code
  WPARAM wParam,   // removal option
  LPARAM lParam    // message
);
```

Parameters

code
[in] Specifies whether the hook procedure must process the message. If *code* is HC_ACTION, the hook procedure must process the message. If *code* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam
[in] Specifies whether the message has been removed from the queue. This parameter can be one of the following values:

Value	Meaning
PM_NOREMOVE	Specifies that the message has not been removed from the queue. (An application called the **PeekMessage** function, specifying the PM_NOREMOVE flag.)
PM_REMOVE	Specifies that the message has been removed from the queue. (An application called **GetMessage**, or it called the **PeekMessage** function, specifying the PM_REMOVE flag.)

lParam
[in] Pointer to an **MSG** structure that contains details about the message.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_GETMESSAGE hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure does not call **CallNextHookEx**, the return value should be zero.

Remarks

The **GetMsgProc** hook procedure can examine or modify the message. After the hook procedure returns control to the system, the **GetMessage** or **PeekMessage** function returns the message, along with any modifications, to the application that originally called it.

An application installs this hook procedure by specifying the WH_GETMESSAGE hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** 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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **GetMessage**, **MSG**, **PeekMessage**, **SetWindowsHookEx**

JournalPlaybackProc

The **JournalPlaybackProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. Typically, an application uses this function to play back a series of mouse and keyboard messages recorded previously by the **JournalRecordProc** hook procedure. As long as a **JournalPlaybackProc** hook procedure is installed, regular mouse and keyboard input is disabled.

The **HOOKPROC** type defines a pointer to this callback function. **JournalPlaybackProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK JournalPlaybackProc(
  int code,       // hook code
  WPARAM wParam,  // not used
```

```
    LPARAM lParam    // message being processed
);
```

Parameters

code

[in] Specifies a code the hook procedure uses to determine how to process the message. This parameter can be one of the following values.

Value	Meaning
HC_GETNEXT	The hook procedure must copy the current mouse or keyboard message to the **EVENTMSG** structure pointed to by the *lParam* parameter.
HC_NOREMOVE	An application has called the **PeekMessage** function with *wRemoveMsg* set to PM_NOREMOVE, indicating that the message is not removed from the message queue after **PeekMessage** processing.
HC_SKIP	The hook procedure must prepare to copy the next mouse or keyboard message to the **EVENTMSG** structure pointed to by *lParam*. Upon receiving the HC_GETNEXT code, the hook procedure must copy the message to the structure.
HC_SYSMODALOFF	A system-modal dialog box has been destroyed. The hook procedure must resume playing back the messages.
HC_SYSMODALON	A system-modal dialog box is being displayed. Until the dialog box is destroyed, the hook procedure must stop playing back messages.

If *code* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

This parameter is not used.

lParam

[in] Pointer to an **EVENTMSG** structure that represents a message being processed by the hook procedure. This parameter is valid only when the *code* parameter is HC_GETNEXT.

Return Values

To have the system wait before processing the message, the return value must be the amount of time, in clock ticks, that the system should wait. (This value can be computed by calculating the difference between the **time** members in the current and previous input messages.) To process the message immediately, the return value should be zero. The return value is used only if the hook code is HC_GETNEXT; otherwise, it is ignored.

Remarks

A **JournalPlaybackProc** hook procedure should copy an input message to the *lParam* parameter. The message must have been previously recorded by using a **JournalRecordProc** hook procedure, which should not modify the message.

To retrieve the same message over and over, the hook procedure can be called several times with the *code* parameter set to HC_GETNEXT without an intervening call with *code* set to HC_SKIP.

If *code* is HC_GETNEXT and the return value is greater than zero, the system sleeps for the number of milliseconds specified by the return value. When the system continues, it calls the hook procedure again with *code* set to HC_GETNEXT to retrieve the same message. The return value from this new call to **JournalPlaybackProc** should be zero; otherwise, the system will go back to sleep for the number of milliseconds specified by the return value, call **JournalPlaybackProc** again, and so on. The system will appear to have stopped responding.

Unlike most other global hook procedures, the **JournalRecordProc** and **JournalPlaybackProc** hook procedures are always called in the context of the thread that set the hook.

After the hook procedure returns control to the system, the message continues to be processed. If *code* is HC_SKIP, the hook procedure must prepare to return the next recorded event message on its next call.

Install the **JournalPlaybackProc** hook procedure by specifying the WH_JOURNALPLAYBACK hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

If the user presses CTRL+ESC or CTRL+ALT+DEL during journal playback, the system stops the playback, unhooks the journal playback procedure, and posts a **WM_CANCELJOURNAL** message to the journaling application.

If the hook procedure returns a message in the range WM_KEYFIRST to WM_KEYLAST, the following conditions apply:

- The *paramL* member of the EVENTMSG structure specifies the virtual key code of the key that was pressed.
- The *paramH* member of the EVENTMSG structure specifies the scan code.
- There's no way to specify a repeat count. The event is always taken to represent one key event.

Requirements

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.

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **EVENTMSG**, **JournalRecordProc**, **PeekMessage**, **SetWindowsHookEx**, **WM_CANCELJOURNAL**

JournalRecordProc

The **JournalRecordProc** hook procedure is an application-defined callback function used with the **SetWindowsHookEx** function. The function records messages the system removes from the system message queue. Later, an application can use a **JournalPlaybackProc** hook procedure to play back the messages.

The **HOOKPROC** type defines a pointer to this callback function. **JournalRecordProc** is a placeholder for the application-defined or library-defined function name:

```
LRESULT CALLBACK JournalRecordProc(
  int code,       // hook code
  WPARAM wParam,  // not used
  LPARAM lParam   // message being processed
);
```

Parameters

code

[in] Specifies how to process the message. This parameter can be one of the following values:

Value	Meaning
HC_ACTION	The *lParam* parameter is a pointer to an **EVENTMSG** structure containing information about a message removed from the system queue. The hook procedure must record the contents of the structure by copying them to a buffer or file.
HC_SYSMODALOFF	A system-modal dialog box has been destroyed. The hook procedure must resume recording.
HC_SYSMODALON	A system-modal dialog box is being displayed. Until the dialog box is destroyed, the hook procedure must stop recording.

If *code* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

This parameter is not used.

lParam

[in] Pointer to an **EVENTMSG** structure that contains the message to be recorded.

Return Values

The return value is ignored.

Remarks

A **JournalRecordProc** hook procedure must copy but not modify the messages. After the hook procedure returns control to the system, the message continues to be processed.

Install the **JournalRecordProc** hook procedure by specifying the WH_JOURNALRECORD hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

A **JournalRecordProc** hook procedure does not need to live in a dynamic-link library (DLL). A **JournalRecordProc** hook procedure can live in the application itself.

Unlike most other global hook procedures, the **JournalRecordProc** and **JournalPlaybackProc** hook procedures are always called in the context of the thread that set the hook.

An application that has installed a **JournalRecordProc** hook procedure should watch for the VK_CANCEL virtual keycode (which is implemented as the CTRL+BREAK key combination on most keyboards). This virtual keycode should be interpreted by the application as a signal that the user wishes to stop journal recording. The application should respond by ending the recording sequence and removing the **JournalRecordProc** hook procedure. Removal is important; it prevents a journaling application from locking up the system by hanging inside a hook procedure.

This role as a signal to stop journal recording means that a CTRL+BREAK key combination cannot itself be recorded. Since the CTRL+C key combination has no such role as a journaling signal, it can be recorded. There are two other key combinations that cannot be recorded: CTRL+ESC and CTRL+ALT+DEL. Those two key combinations cause the system to stop all journaling activities (record or playback), remove all journaling hooks, and post a **WM_CANCELJOURNAL** message to the journaling application.

! Requirements

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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **EVENTMSG**, **JournalPlaybackProc**, **SetWindowsHookEx**, **WM_CANCELJOURNAL**

KeyboardProc

The **KeyboardProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function whenever an application calls the **GetMessage** or **PeekMessage** function and there is a keyboard message (**WM_KEYUP** or **WM_KEYDOWN**) to be processed.

The **HOOKPROC** type defines a pointer to this callback function. **KeyboardProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK KeyboardProc(
  int code,       // hook code
  WPARAM wParam,  // virtual-key code
  LPARAM lParam   // keystroke-message information
);
```

Parameters

code

[in] Specifies a code the hook procedure uses to determine how to process the message. This parameter can be one of the following values:

Value	Meaning
HC_ACTION	The *wParam* and *lParam* parameters contain information about a keystroke message.
HC_NOREMOVE	The *wParam* and *lParam* parameters contain information about a keystroke message, and the keystroke message has not been removed from the message queue. (An application called the **PeekMessage** function, specifying the PM_NOREMOVE flag.)

If *code* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] Specifies the virtual-key code of the key that generated the keystroke message.

lParam

[in] Specifies the repeat count, scan code, extended-key flag, context code, previous key-state flag, and transition-state flag. This parameter can be one or more of the following values:

Value	Description
0–15	Specifies the repeat count. The value is the number of times the keystroke is repeated as a result of the user's holding down the key.
16–23	Specifies the scan code. The value depends on the original equipment manufacturer (OEM).

(continued)

(continued)

Value	Description
24	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if the key is an extended key; otherwise, it is 0.
25–28	Reserved.
29	Specifies the context code. The value is 1 if the ALT key is down; otherwise, it is 0.
30	Specifies the previous key state. The value is 1 if the key is down before the message is sent; it is 0 if the key is up.
31	Specifies the transition state. The value is 0 if the key is being pressed, and 1 if it is being released.

For more information about the *lParam* parameter, see *Keystroke Message Flags*.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_KEYBOARD hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the WH_KEYBOARD hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** 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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **GetMessage**, **PeekMessage**, **SetWindowsHookEx**, **WM_KEYDOWN**, **WM_KEYUP**

LowLevelKeyboardProc

The **LowLevelKeyboardProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function every time a new keyboard input event is about to be posted into a thread input queue. The keyboard input can come from the local keyboard driver or from calls to the **keybd_event** function. If the input comes from a call to **keybd_event**, the input was "injected."

The **HOOKPROC** type defines a pointer to this callback function. **LowLevelKeyboardProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK LowLevelKeyboardProc(
  int nCode,     // hook code
  WPARAM wParam, // message identifier
  LPARAM lParam  // message data
);
```

Parameters

nCode

[in] Specifies a code the hook procedure uses to determine how to process the message. This parameter has the following value:

Value	Meaning
HC_ACTION	The *wParam* and *lParam* parameters contain information about a keyboard message.

If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] Specifies the identifier of the keyboard message. This parameter can be one of the following messages: **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, or **WM_SYSKEYUP**.

lParam

[in] Pointer to a **KBDLLHOOKSTRUCT** structure.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_KEYBOARD_LL hooks will not receive hook notifications and may behave incorrectly as a result. If the hook

procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the WH_KEYBOARD_LL hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

This hook is called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

The hook procedure should process a message in less time than the data entry specified in the **LowLevelHooksTimeout** value in the following registry key:

HKEY_CURRENT_USER\Control Panel\Desktop

The value is in milliseconds. If the hook procedure does not return during this interval, the system will pass the message to the next hook.

Note that debug hooks cannot track this type of hook.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **KBDLLHOOKSTRUCT**, **keybd_event**, **SetWindowsHookEx**, **WM_KEYDOWN**, **WM_KEYUP**, **WM_SYSKEYDOWN**, **WM_SYSKEYUP**

LowLevelMouseProc

The **LowLevelMouseProc** hook procedure is an application-defined callback function used with the **SetWindowsHookEx** function. The system call this function every time a new mouse input event is about to be posted into a thread input queue. The mouse input can come from the local mouse driver or from calls to the **mouse_event** function. If the input comes from a call to **mouse_event**, the input was "injected."

The **HOOKPROC** type defines a pointer to this callback function. **LowLevelMouseProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK LowLevelMouseProc(
  int nCode,       // hook code
  WPARAM wParam,  // message identifier
```

```
  LPARAM lParam  // message data
);
```

Parameters

nCode

[in] Specifies a code the hook procedure uses to determine how to process the message. This parameter has the following value:

Value	Meaning
HC_ACTION	The *wParam* and *lParam* parameters contain information about a mouse message.

If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] Specifies the identifier of the mouse message. This parameter can be one of the following messages: **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_MOUSEMOVE**, **WM_MOUSEWHEEL**, **WM_RBUTTONDOWN**, or **WM_RBUTTONUP**.

lParam

[in] Pointer to an **MSLLHOOKSTRUCT** structure.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_MOUSE_LL hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the WH_MOUSE_LL hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

This hook is called in the context of the thread that installed it. The call is made by sending a message to the thread that installed the hook. Therefore, the thread that installed the hook must have a message loop.

The hook procedure should process a message in less time than the data entry specified in the **LowLevelHooksTimeout** value in the following registry key:

HKEY_CURRENT_USER\Control Panel\Desktop

The value is in milliseconds. If the hook procedure does not return during this interval, the system will pass the message to the next hook.

Note that debug hooks can not track this type of hook.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **mouse_event**, **MSLLHOOKSTRUCT**, **SetWindowsHookEx**, **WM_LBUTTONDOWN**, **WM_LBUTTONUP**, **WM_MOUSEMOVE**, **WM_MOUSEWHEEL**, **WM_RBUTTONDOWN**, **WM_RBUTTONUP**

MessageProc

The **MessageProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function after an input event occurs in a dialog box, message box, menu, or scroll bar, but before the message generated by the input event is processed. The hook procedure can monitor messages for a dialog box, message box, menu, or scroll bar created by a particular application or all applications.

The **HOOKPROC** type defines a pointer to this callback function. **MessageProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK MessageProc(
  int code,       // hook code
  WPARAM wParam,  // not used
  LPARAM lParam   // message data
);
```

Parameters

code

[in] Specifies the type of input event that generated the message. This parameter can be one of the following values:

Value	Meaning
MSGF_DDEMGR	The input event occurred while the Dynamic Data Exchange Management Library (DDEML) was waiting for a synchronous transaction to finish. For more information about DDEML, see *Dynamic Data Exchange Management Library*.
MSGF_DIALOGBOX	The input event occurred in a message box or dialog box.
MSGF_MENU	The input event occurred in a menu.
MSGF_SCROLLBAR	The input event occurred in a scroll bar.

If *code* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing, and return the value returned by **CallNextHookEx**.

wParam
This parameter is not used.

lParam
[in] Pointer to an **MSG** structure.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is highly recommended that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_MSGFILTER hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the rest of the hook chain or the target window procedure.

Remarks

An application installs the hook procedure by specifying the WH_MSGFILTER hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

If an application that uses the DDEML and performs synchronous transactions must process messages before they are dispatched, then it must use the WH_MSGFILTER hook.

! Requirements

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.

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **SetWindowsHookEx**, **MSG**

MouseProc

The **MouseProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function whenever an application calls the **GetMessage** or **PeekMessage** function and there is a mouse message to be processed.

The **HOOKPROC** type defines a pointer to this callback function. **MouseProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK MouseProc(
  int nCode,        // hook code
  WPARAM wParam,    // message identifier
  LPARAM lParam     // mouse coordinates
);
```

Parameters

nCode

[in] Specifies a code the hook procedure uses to determine how to process the message. This parameter can be one of the following values:

Value	Meaning
HC_ACTION	The *wParam* and *lParam* parameters contain information about a mouse message.
HC_NOREMOVE	The *wParam* and *lParam* parameters contain information about a mouse message, and the mouse message has not been removed from the message queue. (An application called the **PeekMessage** function, specifying the PM_NOREMOVE flag.)

If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] Specifies the identifier of the mouse message.

lParam

[in] Pointer to a **MOUSEHOOKSTRUCT** structure.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is recommended highly that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_MOUSE hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the target window procedure.

Remarks

An application installs the hook procedure by specifying the WH_MOUSE hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

The hook procedure must not install a **JournalPlaybackProc** callback 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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **GetMessage**, **JournalPlaybackProc**, **MOUSEHOOKSTRUCT**, **PeekMessage**, **SetWindowsHookEx**

SetWindowsHookEx

The **SetWindowsHookEx** function installs an application-defined hook procedure into a hook chain. You would install a hook procedure to monitor the system for certain types of events. These events are associated with either a specific thread or all threads in the same desktop as the calling thread.

```
HHOOK SetWindowsHookEx(
  int idHook,          // hook type
  HOOKPROC lpfn,       // hook procedure
  HINSTANCE hMod,      // handle to application instance
  DWORD dwThreadId     // thread identifier
);
```

Parameters

idHook

[in] Specifies the type of hook procedure to be installed. This parameter can be one of the following values:

Value	Description
WH_CALLWNDPROC	Installs a hook procedure that monitors messages before the system sends them to the destination window procedure. For more information, see the **CallWndProc** hook procedure.
WH_CALLWNDPROCRET	Installs a hook procedure that monitors messages after they have been processed by the destination window procedure. For more information, see the **CallWndRetProc** hook procedure.
WH_CBT	Installs a hook procedure that receives notifications useful to a computer-based training (CBT) application. For more information, see the **CBTProc** hook procedure.
WH_DEBUG	Installs a hook procedure useful for debugging other hook procedures. For more information, see the **DebugProc** hook procedure.
WH_FOREGROUNDIDLE	Installs a hook procedure that will be called when the application's foreground thread is about to become idle. This hook is useful for performing low priority tasks during idle time. For more information, see the **ForegroundIdleProc** hook procedure.
WH_GETMESSAGE	Installs a hook procedure that monitors messages posted to a message queue. For more information, see the **GetMsgProc** hook procedure.
WH_JOURNALPLAYBACK	Installs a hook procedure that posts messages previously recorded by a WH_JOURNALRECORD hook procedure. For more information, see the **JournalPlaybackProc** hook procedure.
WH_JOURNALRECORD	Installs a hook procedure that records input messages posted to the system message queue. This hook is useful for recording macros. For more information, see the **JournalRecordProc** hook procedure.
WH_KEYBOARD	Installs a hook procedure that monitors keystroke messages. For more information, see the **KeyboardProc** hook procedure.
WH_KEYBOARD_LL	**Windows NT/2000:** Installs a hook procedure that monitors low-level keyboard input events. For more information, see the **LowLevelKeyboardProc** hook procedure.
WH_MOUSE	Installs a hook procedure that monitors mouse messages. For more information, see the **MouseProc** hook procedure.
WH_MOUSE_LL	**Windows NT/2000:** Installs a hook procedure that monitors low-level mouse input events. For more information, see the **LowLevelMouseProc** hook procedure.
WH_MSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. For more information, see the **MessageProc** hook procedure.

WH_SHELL	Installs a hook procedure that receives notifications useful to shell applications. For more information, see the **ShellProc** hook procedure.
WH_SYSMSGFILTER	Installs a hook procedure that monitors messages generated as a result of an input event in a dialog box, message box, menu, or scroll bar. The hook procedure monitors these messages for all applications in the same desktop as the calling thread. For more information, see the **SysMsgProc** hook procedure.

lpfn
[in] Pointer to the hook procedure. If the *dwThreadId* parameter is zero or specifies the identifier of a thread created by a different process, the *lpfn* parameter must point to a hook procedure in a dynamic-link library (DLL). Otherwise, *lpfn* can point to a hook procedure in the code associated with the current process.

hMod
[in] Handle to the DLL containing the hook procedure pointed to by the *lpfn* parameter. The *hMod* parameter must be set to NULL if the *dwThreadId* parameter specifies a thread created by the current process and if the hook procedure is within the code associated with the current process.

dwThreadId
[in] Specifies the identifier of the thread with which the hook procedure is to be associated. If this parameter is zero, the hook procedure is associated with all existing threads running in the same desktop as the calling thread.

Return Values

If the function succeeds, the return value is the handle to the hook procedure.

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

Remarks

An error can occur if the *hMod* parameter is NULL and the *dwThreadId* parameter is zero or specifies the identifier of a thread created by another process.

Calling the **CallNextHookEx** function to chain to the next hook procedure is optional, but it is highly recommended; otherwise, other applications that have installed hooks will not receive hook notifications and may behave incorrectly as a result. You should call **CallNextHookEx** unless you absolutely need to prevent the notification from being seen by other applications.

Before terminating, an application must call the **UnhookWindowsHookEx** function to free system resources associated with the hook.

The scope of a hook depends on the hook type. Some hooks can be set only with global scope; others can be set for only a specific thread, as shown in the following table:

Hook	Scope
WH_CALLWNDPROC	Thread or global
WH_CALLWNDPROCRET	Thread or global
WH_CBT	Thread or global
WH_DEBUG	Thread or global
WH_FOREGROUNDIDLE	Thread or global
WH_GETMESSAGE	Thread or global
WH_JOURNALPLAYBACK	Global only
WH_JOURNALRECORD	Global only
WH_KEYBOARD	Thread or global
WH_KEYBOARD_LL	Global only
WH_MOUSE	Thread or global
WH_MOUSE_LL	Global only
WH_MSGFILTER	Thread or global
WH_SHELL	Thread or global
WH_SYSMSGFILTER	Global only

For a specified hook type, first thread hooks are called, then global hooks.

The global hooks are a shared resource, and installing one affects all applications in the same desktop as the calling thread. All global hook functions must be in libraries. Global hooks should be restricted to special-purpose applications or to use as a development aid during application debugging. Libraries that no longer need a hook should remove its hook procedure.

Requirements

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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **CallWndProc**, **CallWndRetProc**, **CBTProc**, **DebugProc**, **ForegroundIdleProc**, **GetMsgProc**, **JournalPlaybackProc**, **JournalRecordProc**, **LowLevelKeyboardProc**, **LowLevelMouseProc**, **KeyboardProc**, **MessageProc**, **MouseProc**, **ShellProc**, **SysMsgProc**, **UnhookWindowsHookEx**

ShellProc

The **ShellProc** hook procedure is an application-defined or library-defined callback function used with the **SetWindowsHookEx** function. The function receives notifications of shell events from the system.

The **HOOKPROC** type defines a pointer to this callback function. **ShellProc** is a placeholder for the application-defined or library-defined function name.

```
LRESULT CALLBACK ShellProc(
  int nCode,        // hook code
  WPARAM wParam,    // event-specific information
  LPARAM lParam     // event-specific information
);
```

Parameters

nCode

[in] Specifies the hook code. This parameter can be one of the following values:

Value	Meaning
HSHELL_ACCESSIBILITYSTATE	**Windows 2000:** The accessibility state has changed.
HSHELL_ACTIVATESHELLWINDOW	The shell should activate its main window.
HSHELL_APPCOMMAND	**Windows 2000:** The user completed an input event (for example, the user pressed an application command button on the mouse or an application command key on the keyboard), and the application did not handle the **WM_APPCOMMAND** message generated by that input.
HSHELL_GETMINRECT	A window is being minimized or maximized. The system needs the coordinates of the minimized rectangle for the window.
HSHELL_LANGUAGE	Keyboard language was changed or a new keyboard layout was loaded.
HSHELL_REDRAW	The title of a window in the task bar has been redrawn.
HSHELL_TASKMAN	The user has selected the task list. A shell application that provides a task list should return TRUE to prevent Windows from starting its task list.
HSHELL_WINDOWACTIVATED	The activation has changed to a different top-level, unowned window.

(continued)

(continued)

Value	Meaning
HSHELL_WINDOWCREATED	A top-level, unowned window has been created. The window exists when the system calls a **ShellProc** function.
HSHELL_WINDOWDESTROYED	A top-level, unowned window is about to be destroyed. The window still exists when the system calls a **ShellProc** function.

If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

[in] The value depends on the value of the *nCode* parameter, as shown in the following table:

nCode	*wParam*
HSHELL_ACCESSIBILITYSTATE	Indicates which accessibility feature has changed state. This value is one of the following: ACCESS_FILTERKEYS, ACCESS_MOUSEKEYS, or ACCESS_STICKYKEYS.
HSHELL_APPCOMMAND	**Windows 2000:** Where the **WM_APPCOMMAND** message was sent originally; for example, the handle to a window. For more information, see *cmd* parameter in **WM_APPCOMMAND**.
HSHELL_GETMINRECT	Handle to the minimized or maximized window.
HSHELL_LANGUAGE	Handle to the window.
HSHELL_REDRAW	Handle to the redrawn window.
HSHELL_WINDOWACTIVATED	Handle to the activated window.
HSHELL_WINDOWCREATED	Handle to the created window.
HSHELL_WINDOWDESTROYED	Handle to the destroyed window.

lParam

[in] The value depends on the value of the *nCode* parameter, as shown in the following table:

nCode	*lParam*
HSHELL_APPCOMMAND	**Windows 2000:** GET_APPCOMMAND_LPARAM(lParam) is the application command corresponding to the input event. GET_DEVICE_LPARAM(lParam) indicates what generated the input event; for example, the

nCode	*lParam*
	mouse or keyboard. For more information, see the *uDevice* parameter description under **WM_APPCOMMAND**. GET_FLAGS_LPARAM(lParam) depends on the value of *cmd* in **WM_APPCOMMAND**; for example, it might indicate which virtual keys were held down when the **WM_APPCOMMAND** message was originally sent. For more information, see the *dwCmdFlags* description parameter under **WM_APPCOMMAND**.
HSHELL_GETMINRECT	Pointer to a **RECT** structure.
HSHELL_LANGUAGE	Handle to a keyboard layout.
HSHELL_REDRAW	The value is TRUE if the window is flashing, or FALSE otherwise.
HSHELL_WINDOWACTIVATED	The value is TRUE if the window is in full-screen mode, or FALSE otherwise.

Return Values

The return value should be zero.

Remarks

Install this hook procedure by specifying the WH_SHELL hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** 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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **SendMessage**, **SetWindowsHookEx**, **WM_APPCOMMAND**

SysMsgProc

The **SysMsgProc** hook procedure is a library-defined callback function used with the **SetWindowsHookEx** function. The system calls this function after an input event occurs in a dialog box, message box, menu, or scroll bar, but before the message generated by

the input event is processed. The function can monitor messages for any dialog box, message box, menu, or scroll bar in the system.

The **HOOKPROC** type defines a pointer to this callback function. **SysMsgProc** is a placeholder for the library-defined function name.

```
LRESULT CALLBACK SysMsgProc(
  int nCode,        // message flag
  WPARAM wParam,    // not used
  LPARAM lParam     // message data
);
```

Parameters

nCode

[in] Specifies the type of input event that generated the message. This parameter can be one of the following values:

Value	Meaning
MSGF_DIALOGBOX	The input event occurred in a message box or dialog box.
MSGF_MENU	The input event occurred in a menu.
MSGF_SCROLLBAR	The input event occurred in a scroll bar.

If *nCode* is less than zero, the hook procedure must pass the message to the **CallNextHookEx** function without further processing and should return the value returned by **CallNextHookEx**.

wParam

This parameter is not used.

lParam

[in] Pointer to an **MSG** message structure.

Return Values

If *nCode* is less than zero, the hook procedure must return the value returned by **CallNextHookEx**.

If *nCode* is greater than or equal to zero, and the hook procedure did not process the message, it is recommended highly that you call **CallNextHookEx** and return the value it returns; otherwise, other applications that have installed WH_SYSMSGFILTER hooks will not receive hook notifications and may behave incorrectly as a result. If the hook procedure processed the message, it may return a nonzero value to prevent the system from passing the message to the target window procedure.

Remarks

An application installs the hook procedure by specifying the WH_SYSMSGFILTER hook type and a pointer to the hook procedure in a call to the **SetWindowsHookEx** function.

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

Hooks Overview, *Hook Functions*, **CallNextHookEx**, **MSG**, **SetWindowsHookEx**

UnhookWindowsHookEx

The **UnhookWindowsHookEx** function removes a hook procedure installed in a hook chain by the **SetWindowsHookEx** function.

```
BOOL UnhookWindowsHookEx(
  HHOOK hhk    // handle to hook procedure
);
```

Parameters

hhk

[in] Handle to the hook to be removed. This parameter is a hook handle obtained by a previous call to **SetWindowsHookEx**.

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 hook procedure can be in the state of being called by another thread even after **UnhookWindowsHookEx** returns. If the hook procedure is not being called concurrently, the hook procedure is removed immediately before **UnhookWindowsHookEx** returns.

Requirements

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.

Hooks Overview, *Hook Functions*, **SetWindowsHookEx**, **UnhookWindowsHook**

Hook Structures

CBT_CREATEWND

The **CBT_CREATEWND** structure contains information passed to a WH_CBT hook procedure, **CBTProc**, before a window is created.

```
typedef struct tagCBT_CREATEWND {
    LPCREATESTRUCT lpcs;
    HWND           hwndInsertAfter;
} CBT_CREATEWND, *LPCBT_CREATEWND;
```

Members

lpcs
Pointer to a **CREATESTRUCT** structure that contains initialization parameters for the window about to be created.

hwndInsertAfter
Handle to the window whose position in the Z order precedes that of the window being created.

Requirements

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.

Hooks Overview, *Hook Structures*, **CBTProc**, **CREATESTRUCT**, **SetWindowsHookEx**

CBTACTIVATESTRUCT

The **CBTACTIVATESTRUCT** structure contains information passed to a WH_CBT hook procedure, **CBTProc**, before a window is activated.

```
typedef struct tagCBTACTIVATESTRUCT {
    BOOL fMouse;
```

```
    HWND hWndActive;
} CBTACTIVATESTRUCT, *LPCBTACTIVATESTRUCT;
```

Members

fMouse

Specifies whether the window is being activated as a result of a mouse click. This value is TRUE if a mouse click is causing the activation or FALSE if it is not.

hWndActive

Handle to the active 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

Hooks Overview, *Hook Structures*, **CBTProc**, **SetWindowsHookEx**

CWPRETSTRUCT

The **CWPRETSTRUCT** structure defines the message parameters passed to a WH_CALLWNDPROCRET hook procedure, **CallWndRetProc**.

```
typedef struct tagCWPRETSTRUCT {
    LRESULT lResult;
    LPARAM  lParam;
    WPARAM  wParam;
    UINT    message;
    HWND    hwnd;
} CWPRETSTRUCT, *PCWPRETSTRUCT;
```

Members

lResult

Specifies the return value of the window procedure that processed the message specified by the **message** value.

lParam

Specifies additional information about the message; the exact meaning depends on the **message** value.

wParam

Specifies additional information about the message; the exact meaning depends on the **message** value.

message
Specifies the message.

hwnd
Handle to the window that processed the message specified by the **message** value.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Hooks Overview, *Hook Structures*, **CallWndRetProc**, **SetWindowsHookEx**

CWPSTRUCT

The **CWPSTRUCT** structure defines the message parameters passed to a WH_CALLWNDPROC hook procedure, **CallWndProc**.

```
typedef struct tagCWPSTRUCT {
    LPARAM  lParam;
    WPARAM  wParam;
    UINT    message;
    HWND    hwnd;
} CWPSTRUCT, *PCWPSTRUCT;
```

Members

lParam
Specifies additional information about the message; the exact meaning depends on the **message** value.

wParam
Specifies additional information about the message; the exact meaning depends on the **message** value.

message
Specifies the message.

hwnd
Handle to the window to receive the 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.

Hooks Overview, *Hook Structures*, **CallWndProc**, **SetWindowsHookEx**

DEBUGHOOKINFO

The **DEBUGHOOKINFO** structure contains debugging information passed to a WH_DEBUG hook procedure, **DebugProc**.

```
typedef struct tagDEBUGHOOKINFO {
    DWORD  idThread;
    DWORD  idThreadInstaller;
    LPARAM lParam;
    WPARAM wParam;
    int    code;
} DEBUGHOOKINFO, *PDEBUGHOOKINFO;
```

Members

idThread
Handle to the thread containing the filter function.

idThreadInstaller
Handle to the thread that installed the debugging filter function.

lParam
Specifies the value to be passed to the hook in the *lParam* parameter of the **DebugProc** callback function.

wParam
Specifies the value to be passed to the hook in the *wParam* parameter of the **DebugProc** callback function.

code
Specifies the value to be passed to the hook in the *nCode* parameter of the **DebugProc** callback function.

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.

See Also

Hooks Overview, *Hook Structures*, **DebugProc**, **SetWindowsHookEx**

EVENTMSG

The **EVENTMSG** structure contains information about a hardware message sent to the system message queue. This structure is used to store message information for the **JournalPlaybackProc** callback function.

```
typedef struct tagEVENTMSG {
    UINT  message;
    UINT  paramL;
    UINT  paramH;
    DWORD time;
    HWND  hwnd;
} EVENTMSG, *PEVENTMSG;
```

Members

message
Specifies the message.

paramL
Specifies additional information about the message; the exact meaning depends on the **message** value.

paramH
Specifies additional information about the message; the exact meaning depends on the **message** value.

time
Specifies the time at which the message was posted.

hwnd
Handle to the window to 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: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Hooks Overview, *Hook Structures*, **JournalPlaybackProc**, **SetWindowsHookEx**

KBDLLHOOKSTRUCT

The **KBDLLHOOKSTRUCT** structure contains information about a low-level keyboard input event.

```
typedef struct tagKBDLLHOOKSTRUCT {
    DWORD     vkCode;
    DWORD     scanCode;
    DWORD     flags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
} KBDLLHOOKSTRUCT, *PKBDLLHOOKSTRUCT;
```

Members

vkCode

Specifies a virtual-key code. The code must be a value in the range 1 to 254.

scanCode

Specifies a hardware scan code for the key.

flags

Specifies the extended-key flag, event-injected flag, context code, and transition-state flag. This member is specified as follows:

Value	Description
0	Specifies whether the key is an extended key, such as a function key or a key on the numeric keypad. The value is 1 if the key is an extended key; otherwise, it is 0.
1–3	Reserved.
4	Specifies whether the event was injected. The value is 1 if the event was injected; otherwise, it is 0.
5	Specifies the context code. The value is 1 if the ALT key is pressed; otherwise, it is 0.
6	Reserved.
7	Specifies the transition state. The value is 0 if the key is pressed and 1 if it is being released.

An application can use the following values to test the keystroke flags:

Value	Purpose
LLKHF_ALTDOWN	Test the context code.
LLKHF_EXTENDED	Test the extended-key flag.
LLKHF_INJECTED	Test the event-injected flag.
LLKHF_UP	Test the transition-state flag.

time

Specifies the time stamp for this message.

dwExtraInfo

Specifies extra information associated with the message.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Hooks Overview, *Hook Structures*, **LowLevelKeyboardProc**, **SetWindowsHookEx**

MOUSEHOOKSTRUCT

The **MOUSEHOOKSTRUCT** structure contains information about a mouse event passed to a WH_MOUSE hook procedure, **MouseProc**.

```
typedef struct tagMOUSEHOOKSTRUCT {
    POINT     pt;
    HWND      hwnd;
    UINT      wHitTestCode;
    ULONG_PTR dwExtraInfo;
} MOUSEHOOKSTRUCT, *PMOUSEHOOKSTRUCT;
```

Members

pt
Specifies a **POINT** structure that contains the x-coordinates and y-coordinates of the cursor, in screen coordinates.

hwnd
Handle to the window that will receive the mouse message corresponding to the mouse event.

wHitTestCode
Specifies the hit-test value. For a list of hit-test values, see the description of the **WM_NCHITTEST** message.

dwExtraInfo
Specifies extra information associated with the 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.

Hooks Overview, *Hook Structures*, **MouseProc**, **POINT**, **SetWindowsHookEx**, **WM_NCHITTEST**

MOUSEHOOKSTRUCTEX

The **MOUSEHOOKSTRUCTEX** structure contains information about a mouse event passed to a WH_MOUSE hook procedure, **MouseProc**.

This is an extension of the **MOUSEHOOKSTRUCT** structure that includes information about wheel movement or the use of the X button.

```
typedef struct tagMOUSEHOOKSTRUCTEX {
    MOUSEHOOKSTRUCT;
    DWORD mouseData;
} MOUSEHOOKSTRUCTEX, *PMOUSEHOOKSTRUCTEX;
```

Members

MOUSEHOOKSTRUCT

The members of a **MOUSEHOOKSTRUCT** structure make up the first part of this structure.

mouseData

If the message is **WM_MOUSEWHEEL**, the HIWORD of this member is the wheel delta. The LOWORD is undefined and reserved.

If the message is **WM_XBUTTONDOWN**, **WM_XBUTTONUP**, or **WM_XBUTTONDBLCLK**, the HIWORD OF **mouseData** specifies which X button was pressed or released, and the LOWORD is undefined and reserved. This member can be one or more of the following values:

Value	Meaning
XBUTTON1	The first X button was pressed or released.
XBUTTON2	The second X button was pressed or released.

Otherwise, **mouseData** is not used.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

Hooks Overview, *Hook Structures*, **MouseProc**, **MOUSEHOOKSTRUCT**, **WM_MOUSEWHEEL**, **WM_XBUTTONDOWN**, **WM_XBUTTONDBLCLK**, **WM_XBUTTONUP**

MSLLHOOKSTRUCT

The **MSLLHOOKSTRUCT** structure contains information about a low-level keyboard input event.

```
typedef struct tagMSLLHOOKSTRUCT {
    POINT     pt;
    DWORD     mouseData;
    DWORD     flags;
    DWORD     time;
    ULONG_PTR dwExtraInfo;
} MSLLHOOKSTRUCT, *PMSLLHOOKSTRUCT;
```

Members

pt

Specifies a **POINT** structure that contains the x-coordinates and y-coordinates of the cursor, in screen coordinates.

mouseData

If the message is **WM_MOUSEWHEEL**, the high-order word of this member is the wheel delta. The low-order word is reserved.

If the message is **WM_XBUTTONDOWN**, **WM_XBUTTONUP**, or **WM_XBUTTONDBLCLK**, the high-order word specifies which X button was pressed or released, and the low-order word is reserved. This value can be one or more of the following values:

Value	Meaning
XBUTTON1	The first X button was pressed or released.
XBUTTON2	The second X button was pressed or released.

Otherwise, **mouseData** is not used.

flags

Specifies the event-injected flag.

Value	Meaning
0	Specifies whether the event was injected. The value is 1 if the event was injected; otherwise, it is 0.
1–15	Reserved.

An application can use the following value to test the mouse flags:

Value	Purpose
LLMHF_INJECTED	Test the event-injected flag.

time
Specifies the time stamp for this message.

dwExtraInfo
Specifies extra information associated with the message.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP3 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.

See Also

Hooks Overview, Hook Structures, **LowLevelMouseProc**, **POINT**, **SetWindowsHookEx**, **WM_MOUSEWHEEL**, **WM_XBUTTONDBLCLK**, **WM_XBUTTONDOWN**, **WM_XBUTTONUP**

Hook Messages

The following messages are used with hooks:

WM_CANCELJOURNAL
WM_QUEUESYNC

WM_CANCELJOURNAL

The **WM_CANCELJOURNAL** message is posted to an application when a user cancels the application's journaling activities. The message is posted with a NULL window handle.

Parameters

This message has no parameters.

Return Values

This message does not return a value. It is meant to be processed from within an application's main loop or a **GetMessage** hook procedure, not from a window procedure.

Remarks

Journal record and playback modes are modes imposed on the system that let an application sequentially record or play back user input. The system enters these modes when an application installs a **JournalRecordProc** or **JournalPlaybackProc** hook procedure. When the system is in either of these journaling modes, applications must take turns reading input from the input queue. If any one application stops reading input while the system is in a journaling mode, other applications are forced to wait.

To ensure a robust system, one that cannot be made to stop responding by any one application, the system automatically cancels any journaling activities when a user presses the CTRL+ESC or CTRL+ALT+DEL key combination. The system then unhooks any journaling hook procedures and posts a **WM_CANCELJOURNAL** message, with a NULL window handle, to the application that set the journaling hook.

The **WM_CANCELJOURNAL** message has a NULL window handle and, therefore, it cannot be dispatched to a window procedure. There are two ways for an application to see a **WM_CANCELJOURNAL** message: if the application is running in its own main loop, it must catch the message between its call to **GetMessage** or **PeekMessage** and its call to **DispatchMessage**; if the application is not running in its own main loop, it must set a **GetMsgProc** hook procedure (through a call to **SetWindowsHookEx** specifying the WH_GETMESSAGE hook type) that watches for the message.

When an application sees a **WM_CANCELJOURNAL** message, it can assume two things: the user has intentionally cancelled the journal record or playback mode, and the system has already unhooked any journal record or playback hook procedures.

Note that the key combinations mentioned above (CTRL+ESC or CTRL+ALT+DEL) cause the system to cancel journaling. If any one application stops respodning, these key combinations give the user a means of recovery. The VK_CANCEL virtual key code (usually implemented as the CTRL+BREAK key combination) is what an application that is in journal record mode should watch for as a signal that the user wishes to cancel the journaling activity. The difference is that watching for VK_CANCEL is a suggested behavior for journaling applications, whereas CTRL+ESC or CTRL+ALT+DEL causes the system to cancel journaling, regardless of a journaling application's behavior.

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.

See Also

Hooks Overview, *Hook Messages*, **DispatchMessage**, **GetMessage**, **GetMsgProc**, **JournalPlaybackProc**, **JournalRecordProc**, **PeekMessage**, **SetWindowsHookEx**

WM_QUEUESYNC

The **WM_QUEUESYNC** message is sent by a computer-based training (CBT) application to separate user-input messages from other messages sent through the WH_JOURNALPLAYBACK hook procedure.

Parameters

This message has no parameters.

Return Values

A CBT application should return zero if it processes this message.

Remarks

Whenever a CBT application uses the WH_JOURNALPLAYBACK hook procedure, the first and last messages are **WM_QUEUESYNC**. This allows the CBT application to intercept and examine user-initiated messages without doing so for events that it sends.

If an application specifies a NULL window handle, the message is posted to the message queue of the active 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.

Hooks Overview, *Hook Messages*, **SetWindowsHookEx**

CHAPTER 9

File I/O

About File I/O

A file is the basic unit of storage that enables a computer to distinguish one set of information from another. This overview describes the file input and output (I/O) operations and the information provided by the file I/O functions.

Files are stored on storage media, such as disks or tapes, and can be organized into groups called directories. The file I/O functions enable applications to create, open, modify, and delete files. They also enable applications to obtain system information, such as what disk drives are present.

File System Organization

A volume is a storage device, such as a fixed disk or floppy disk, formatted to store directories and files. Each volume has a root directory. Directories and files on the volume are organized in a tree structure that starts at the root directory. Each directory entry identifies the name, attributes, location, and size of a file or subdirectory.

A large volume can be divided into more than one logical volume, also called a partition. To the user and to the operating system, each partition appears to be a separate volume.

A file system is operating system software that manages the low-level organization of files on a volume. The operating system supports one or more of the following file systems:

- File Allocation Table (FAT)
- NT File System (NTFS)

The type of file system defines the file name conventions on a volume and may also provide specific file system features, such as security, recoverability, and high I/O performance. Each volume can use a different file system.

Accessing Files

The first time a file function accesses a volume and whenever a diskette is placed in a floppy-disk drive, the operating system examines the volume to determine its file system. Thereafter, the operating system manages all I/O to that volume through the device driver supporting the file system.

The file I/O functions enable applications to access files regardless of the underlying file system. However, capabilities may vary depending on the file system and/or operating system in use. For example, the **CreateFile** function includes a security parameter that provides no security benefits for files not residing on an NTFS volume.

The file I/O functions that create, open, and delete files and directories identify them by their names. These functions store or search for the file or directory in the current directory on the current disk drive, unless the name explicitly specifies a path to a different directory, disk drive, or both.

File Name Conventions

Although each file system can have specific rules about the formation of individual components in a directory or file name, all file systems follow the same general conventions: a base file name and an optional extension, separated by a period. For example, the MS-DOS FAT file system supports 8 characters for the base file name and 3 characters for the extension. This is known as an 8.3 file name. The FAT file system and NTFS support file names that can be up to 255 characters long. This is known as a long file name. To get an MS-DOS file name given a long file name, use the **GetShortPathName** function. To get the full path of a file, use the **GetFullPathName** function.

Both file systems use the backslash (\) character to separate directory names and the file name when forming a path.

General rules for applications creating names for directories and files or processing names supplied by the user include the following:

- Use any character in the current code page for a name, but do not use a path separator, a character in the range 0 through 31, or any character explicitly disallowed by the file system. A name can contain characters in the extended character set (128–255).
- Use the backslash (\), the forward slash (/), or both to separate components in a path. No other character is acceptable as a path separator. Note that UNC names must adhere to the following format: *server**share*.
- Use a period (.) as a directory component in a path to represent the current directory.
- Use two consecutive periods (..) as a directory component in a path to represent the parent of the current directory.
- Use a period (.) to separate the base file name from the extension in a directory name or file name.
- Do not use the following characters in directory names or file names, because they are reserved:

  < > : “ / \ |
- Do not use device names, such as *aux*, *con*, *lpt1*, and *prn*, as file names or directory names.

- Process a path as a null-terminated string. The maximum length for a path, including a trailing backslash, is given by MAX_PATH.

  The wide (Unicode) versions of the **CreateDirectory**, **FindFirstFile**, **GetFileAttributes**, and **SetFileAttributes** functions permit paths that exceed the MAX_PATH length if the path has the "\\?\" or "\\?\UNC\" prefix. However, each component in the path cannot be more than MAX_PATH characters long. Use the "\\?\" prefix with paths for local storage devices and the "\\?\UNC\" prefix with paths having the Universal Naming Convention (UNC) format.
- Do not assume case sensitivity. Consider names such as *OSCAR*, *Oscar*, and *oscar* to be the same.

By following the rules listed in this section, an application can create valid names for files and directories regardless of the file system in use.

Backslashes (\) are used as element dividers in paths (dividing the file name from the path to it, or directories from one another in a path). You cannot use them in file or directory names. They may be required as part of volume names (for example, "C:\").

Long File Names

The operating system stores the long file names on disk as special directory entries. When you create a long file name, the operating system creates a corresponding short 8.3 form of the name.

The operating system stores the long file names on disk in Unicode. This means that the original long file name is always preserved, even if it contains extended characters, and regardless of the code page that is active during a disk read or write operation. The case of the file name is preserved, but the file system is not case-sensitive.

The valid character set for these long file names is the NTFS character set, less one character: the colon (:) that NTFS uses for opening alternate file streams. This means that you can freely copy files between NTFS and FAT partitions without losing any file name information.

MS-DOS Device Names

MS-DOS device names are global. Once defined, an MS-DOS device name remains visible to all processes until either it is explicitly removed or the system reboots.

The **DefineDosDevice** and **SetVolumeMountPoint** functions are used to create and modify the symbolic links used to implement the MS-DOS device namespace. To obtains a list of all MS-DOS devices known to the system, use the **QueryDosDevice** function.

File Operations

Creating and Opening Files with the CreateFile Function

The **CreateFile** function can create a new file or open an existing file.

When an application uses **CreateFile**, it must specify whether it will read from the file, write to the file, or both. The application must also specify what action to take whether or not the file exists. For example, an application can specify that **CreateFile** always be used to create the file. As a result, the function creates the file if it does not exist and overwrites the file if it does exist.

CreateFile also enables an application to specify whether it wants to share the file for reading, writing, both, or neither. A file that is not shared cannot be opened more than once by the first application nor by another application until the first application has closed the file.

The operating system assigns a unique identifier, called a *file handle*, to each file that is opened or created. An application can use the file handle in functions that read from, write to, and describe the file. It is valid until the file is closed. When an application starts, it inherits all open file handles from the process that started it, if the handles are inheritable. For more information about processes, see *Processes and Threads*.

For information about the standard input, standard output, and standard error file handles, see *Consoles and Character-Mode Support*.

An application should check the return value of **CreateFile** before attempting to use the handle to access the file. If an error occurs, the application can use the **GetLastError** function to get extended error information.

Creating Temporary Files

Applications can obtain unique file names for temporary files with the **GetTempFileName** function. The **GetTempPath** function retrieves the path to the directory where temporary files are to be created.

Copying and Moving Files

Before a file can be copied, it must be closed or opened only for reading. No thread can have the file opened for writing. To copy an existing file to a new one, use the **CopyFile** or **CopyFileEx** function. Applications can specify whether **CopyFile** and **CopyFileEx** fail if the destination file already exists.

The **CopyFileEx** function also allows an application to specify the address of a callback function (see **CopyProgressRoutine**) that is called each time another portion of the file has been copied. The application can use this information to display am indicator that shows the total number of bytes copied as a percent of the total file size.

The **ReplaceFile** function replaces one file with another file, with the option of creating a backup copy of the original file. The operation is atomic; either all data is saved to the file, or the original file is left unchanged. The function preserves attributes of the original file, such as its creation time, ACLs, and encryption attribute.

A file must also be closed before an application can move it. The **MoveFile** and **MoveFileEx** functions copy an existing file to a new location and deletes the original.

The **MoveFileEx** function also allows an application to specify how to move the file. The function can replace an existing file, move a file across volumes, and delay moving the file until the operating system is restarted.

Reading from and Writing to a File

Every open file has a *file pointer* that specifies the next byte to be read or the location to receive the next byte written. When a file is opened for the first time, the system places the file pointer at the beginning of the file. As each byte is read or written, the system advances the file pointer. An application can also move the file pointer by using the **SetFilePointer** function.

An application reads from and writes to a file by using the **ReadFile** and **WriteFile** functions. These functions require a handle to a file to be opened for reading and writing, respectively. **ReadFile** and **WriteFile** read and write a specified number of bytes at the location indicated by the file pointer. The data is read and written exactly as specified; the functions do not format the data.

For Very Large Memory (VLM), **ReadFileVlm** and **WriteFileVlm** may be used similarly to **ReadFile** and **WriteFile**. For more information, see *Very Large Memory (VLM)*.

An application can implement a scatter-gather scheme with **ReadFileScatter** and **WriteFileGather**. A scatter-gather scheme uses the operating system to deliver in one operation multiple discrete chunks of data (such as database records) from a file to separate, noncontiguous buffers in memory. A scatter-gather scheme also writes the data from noncontiguous buffers in one operation.

When the file pointer reaches the end of a file and the application attempts to read from the file, no error occurs, but no bytes are read. Therefore, reading zero bytes without an error means the application has reached the end of the file. Writing zero bytes does nothing.

An application can truncate or extend a file by using the **SetEndOfFile** function. This function sets the end of file to the current position of the file pointer.

When an application writes to a file, the system usually collects the data being written in an internal buffer and writes the data to the disk on a regular basis.

An application can force the operating system to write the contents of the buffer to the disk by using the **FlushFileBuffers** function. Alternatively, an application can specify that write operations are to bypass the internal buffer and write directly to the disk by setting a flag when the file is created or opened by using the **CreateFile** function.

If there is data in the internal buffer when the file is closed, the operating system does not automatically write the contents of the buffer to the disk before closing the file. If the application does not force the operating system to write the buffer to disk before closing the file, the caching algorithm determines when the buffer is written.

Note Accessing the data buffer while a read or write operation is using the buffer may lead to corruption of the data in that buffer. Applications must not read from, write to, reallocate, or free the data buffer that a read or write operation is using until the operation completes.

Locking and Unlocking Files

Although the system allows more than one application to open a file and write to it, applications must not write over each other's work. An application can prevent this problem by temporarily locking a region in a file. The **LockFile** and **LockFileEx** functions lock a specified range of bytes in a file. The range may extend beyond the current end of the file. Locking part of a file prevents all other processes from reading or writing anywhere in the specified area. Attempts to read from or write to a region locked by another process always fail.

The **LockFileEx** function allows an application to specify either a *shared lock* or an *exclusive lock*. An exclusive lock denies all other processes both read and write access to the specified region of the file. A shared lock denies all processes write access to the specified region of the file, including the process that first locks the region. This can be used to create a read-only region in a file.

The application unlocks the region by using the **UnlockFile** or **UnlockFileEx** function. An application should unlock all locked areas before closing a file.

Searching for Files

An application can search the current directory for all file names that match a given pattern by using the **FindFirstFile**, **FindFirstFileEx**, **FindNextFile**, and **FindClose** functions. The pattern must be a valid file name and can include wildcard characters.

The **FindFirstFile** and **FindFirstFileEx** functions create handles that **FindNextFile** uses to search for other files with the same pattern. All functions return information about the file that was found. This information includes the file name, size, attributes, and time.

The **FindFirstFileEx** function also allows an application to search for files based on additional search criteria. The function can limit searches to device names or directory names.

The **FindClose** function destroys handles created by **FindFirstFile** and **FindFirstFileEx**.

An application can search for a single file on a specific path by using the **SearchPath** function.

Monitoring Directories

An application can monitor the contents of a directory and its subdirectories by using the **FindFirstChangeNotification**, **FindNextChangeNotification**, and **FindCloseChangeNotification** functions. Waiting for a change notification is similar to

having a read operation pending against a directory and, if necessary, its subdirectories. When something changes within the directory being watched, the read operation is completed. For example, an application can use these functions to update a directory listing whenever a file name within the monitored directory changes.

An application can specify a set of conditions that trigger a change notification by using the **FindFirstChangeNotification** function. The conditions include changes to file names, directory names, attributes, file size, time of last write, and security. This function also returns a handle that can be waited on by using the wait functions. If the wait condition is satisfied, **FindNextChangeNotification** can be used to provide a notification handle to wait on subsequent changes.

The **FindCloseChangeNotification** function closes the notification handle.

Another way to monitor directory changes is by using the **ReadDirectoryChangesW** function.

Closing and Deleting Files

To use operating system resources efficiently, an application should close files when they are no longer needed by using the **CloseHandle** function. If a file is open when an application terminates, the system closes it automatically.

The **DeleteFile** function can be used to delete a file. The file must, however, be closed before any attempt to delete it will succeed.

Directory Operations

When an application creates a new file, the operating system adds it to the specified directory. Each directory can have any number of files, up to the physical limit of the disk. An application can create new directories and delete existing directories by using the **CreateDirectory**, **CreateDirectoryEx**, and **RemoveDirectory** functions. An application cannot delete a directory unless it is empty.

The directory at the end of the active path is called the current directory; it is the directory in which the active application started, unless explicitly changed. An application can determine which directory is current by using the **GetCurrentDirectory** function. An application can change the current directory by using the **SetCurrentDirectory** function.

Windows NT/2000: You can obtain a handle to a directory by calling the **CreateFile** function with the FILE_FLAG_BACKUP_SEMANTICS flag set, as follows:

```
hDir = CreateFile (
    DirName,
    GENERIC_READ,
    FILE_SHARE_READ|FILE_SHARE_DELETE,
    NULL,
    OPEN_EXISTING,
    FILE_FLAG_BACKUP_SEMANTICS,
    NULL
);
```

You can pass a directory handle to the following functions:

BackupRead
BackupSeek
BackupWrite
GetFileInformationByHandle
GetFileSize
GetFileTime
GetFileType
ReadDirectoryChangesW
SetFileTime

Asynchronous Input and Output

Asynchronous input and output (asynchronous I/O) allows some I/O functions to return immediately, even though an I/O request is still pending. Asynchronous I/O enables an application to continue with other processing and wait for the I/O to be completed at a later time. Asynchronous I/O is also called *overlapped I/O*.

The **ReadFile**, **WriteFile**, **ReadFileVlm**, and **WriteFileVlm** functions enable an application to specify an **OVERLAPPED** structure that indicates where to position the file pointer before the read or write operation. The handle of the file being read from or written to must have been opened with the FILE_FLAG_OVERLAPPED flag. You can also create an event and put the handle in the **OVERLAPPED** structure; the wait functions can then be used to wait for the I/O operation to complete by waiting on the event handle.

An application can also wait on the file handle to synchronize the completion of an I/O operation, but doing so requires extreme caution. Each time an I/O operation is started, the operating system sets the file handle to the nonsignaled state. Each time an I/O operation is completed, the operating system sets the file handle to the signaled state. Therefore, if an application starts two I/O operations and waits on the file handle, there is no way to determine which operation is finished when the handle is set to the signaled state. If an application must perform multiple asynchronous I/O operations on a single file, it should wait on the event handle in the **OVERLAPPED** structure for each I/O operation, rather than on the file handle.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle.

The **ReadFileEx** and **WriteFileEx** functions enable an application to specify a routine to execute (see **FileIOCompletionRoutine**) when the asynchronous I/O request is completed.

For more information, see *Synchronization and Overlapped Input and Output*.

I/O Completion Ports

I/O completion ports are used with asynchronous I/O. The **CreateIoCompletionPort** function associates an I/O completion port with one or more file handles. When an asynchronous I/O operation started on a file handle associated with a completion port is completed, an I/O completion packet is queued to the port. This can be used to combine the synchronization point for multiple file handles into a single object.

A thread uses the **GetQueuedCompletionStatus** function to wait for a completion packet to be queued to the completion port, rather than waiting directly for the asynchronous I/O to complete. Threads that block their execution on a completion port are released in last-in-first-out (LIFO) order. This means that when a completion packet is queued to the completion port, the system releases the last thread to block its execution on the port.

When a thread calls **GetQueuedCompletionStatus**, it is associated with the specified completion port until it exits, specifies a different completion port, or frees the completion port. A thread can be associated with at most one completion port.

The most important property of a completion port is the concurrency value. The concurrency value of a completion port is specified when the completion port is created. This value limits the number of runnable threads associated with the completion port. When the total number of runnable threads associated with the completion port reaches the concurrency value, the system blocks the execution of any subsequent threads that specify the completion port until the number of runnable threads associated with the completion port drops below the concurrency value. The most efficient scenario occurs when there are completion packets waiting in the queue, but no waits can be satisfied because the port has reached its concurrency limit. In this case, when a running thread calls **GetQueuedCompletionStatus**, it will immediately pick up the queued completion packet. No context switches will occur, because the running thread is continually picking up completion packets and the other threads are unable to run.

The best value to pick for the concurrency value is the number of CPUs on the machine. If your transaction required a lengthy computation, a larger concurrency value will allow more threads to run. Each transaction will take longer to complete, but more transactions will be processed at the same time. It is easy to experiment with the concurrency value to achieve the best effect for your application.

The **PostQueuedCompletionStatus** function allows an application to queue its own special-purpose I/O completion packets to the completion port without starting an asynchronous I/O operation. This is useful for notifying worker threads of external events.

The completion port is freed when there are no more references to it. The completion port handle and every file handle associated with the completion port reference the completion port. All the handles must be closed to free the completion port. To close the port handle, call the **CloseHandle** function.

Getting Information About Files

File Attributes

The **GetFileInformationByHandle** function retrieves information about a file and stores it in a structure of type BY_HANDLE_FILE_INFORMATION. This information includes creation time, file size, and attributes.

Eight characteristics called attributes may be associated with a file. The attributes can be one or more of the following values.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file or directory is an archive file. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DEVICE	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY	The handle identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file or directory has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Microsoft® Windows® 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.
FILE_ATTRIBUTE_REPARSE_POINT	The file has an associated reparse point.
FILE_ATTRIBUTE_SPARSE_FILE	The file is a sparse file.
FILE_ATTRIBUTE_SYSTEM	The file or directory is part of the operating system or is used exclusively by the operating system.

FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

An application can retrieve the file attributes by using the **GetFileAttributes** or **GetFileAttributesEx** function. The **CreateFile** and **SetFileAttributes** functions can set many of the attributes. However, applications cannot set all attributes. For more information on how to set these attributes, see **SetFileAttributes**.

File Type

The **GetFileType** function returns the type of a file: disk, character (such as a console), pipe, or unknown. The **GetBinaryType** function determines whether a file is executable, and if so, what type of executable file it is. The **GetFileSize** function returns the size of a file.

File Date and Time

Applications can retrieve and set the date and time a file was created, last modified, or last accessed by using the **GetFileTime** and **SetFileTime** functions. For more information about file times, see *Time*.

File Code Page

The **AreFileApisANSI** function determines whether the file I/O functions are using the ANSI or OEM character set code page. The **SetFileApisToANSI** function causes the functions to use the ANSI code page. The **SetFileApisToOEM** function causes the functions to use the OEM code page.

By default, file I/O functions use ANSI file names. Functions exported by KERNEL32.DLL that accept or return file names are affected by the file code page setting.

Both **SetFileApisToANSI** and **SetFileApisToOEM** set the code page per process, rather than per thread or per computer.

Volume Information

The **GetVolumeInformation** function retrieves information about the file system on a given volume. This information includes the volume name, volume serial number, file system name, file system flags, maximum length of a file name, and so on. The **SetVolumeLabel** function sets the label of a file system volume.

The **GetSystemDirectory** and **GetWindowsDirectory** functions retrieve the paths to the system directory and the Windows directory, respectively.

The **GetDiskFreeSpace** function retrieves organizational information about a volume, including the number of bytes per sector, the number of sectors per cluster, the number of free clusters, and the total number of clusters.

The **GetDriveType** function indicates whether the volume referenced by the specified drive letter is a removable, fixed, CD-ROM, RAM, or network drive.

The **GetLogicalDrives** function identifies the volumes present. The **GetLogicalDriveStrings** function retrieves a null-terminated string for each volume present. Use these strings whenever a root directory is required.

File and Directory Security

Windows NT/Windows 2000 security enables you to control access to file and directory objects stored on a secure file system, such as NTFS. For more information about security, see *Access-Control Model.*

You can specify a security descriptor for a file or directory when you call the **CreateFile**, **CreateDirectory**, or **CreateDirectoryEx** function. To retrieve the security descriptor of a file or directory object, call the **GetNamedSecurityInfo** or **GetSecurityInfo** function. To change the security descriptor of a file or directory object, call the **SetNamedSecurityInfo** or **SetSecurityInfo** function.

When a thread calls the **CreateFile** function to open a handle to a file or directory object, the thread requests a set of generic access rights to the object. The requested access rights determine the operations that the thread can perform with the returned handle. Before returning a handle to the object, **CreateFile** checks the thread's access token and the requested access rights against the DACL in the object's security descriptor.

For file and directory objects, GENERIC_READ access maps to the following standard and specific access rights.

Access right	Description
FILE_READ_ATTRIBUTES	Right to read file attributes.
FILE_READ_DATA	Right to read data from the file. For a directory, the right to list the contents of the directory.
FILE_READ_EA	Right to read extended attributes.
STANDARD_RIGHTS_READ	Includes READ_CONTROL, which is the right to read the information in the object's security descriptor, not including the information in the SACL.
SYNCHRONIZE	Right to specify a file handle in one of the wait functions. However, for asynchronous file I/O operations, you should wait on the event handle in an **OVERLAPPED** structure rather than using the file handle for synchronization.

For file and directory objects, GENERIC_WRITE access maps to the following standard and specific access rights.

Access right	Description
FILE_APPEND_DATA	Right to append data to the file. For a directory, the right to create a subdirectory.
FILE_WRITE_ATTRIBUTES	Right to write file attributes.
FILE_WRITE_DATA	Right to write data to the file. For a directory, the right to create a file in the directory.
FILE_WRITE_EA	Right to write extended attributes.
STANDARD_RIGHTS_WRITE	Includes READ_CONTROL, which is the right to read the information in the object's security descriptor, not including the information in the SACL.
SYNCHRONIZE	Right to specify a file handle in one of the wait functions. However, for asynchronous file I/O operations, you should wait on the event handle in an **OVERLAPPED** structure rather than using the file handle for synchronization.

You cannot use an access-denied ACE to deny only GENERIC_READ or only GENERIC_WRITE access to a file. This is because for file objects, the generic mappings for both GENERIC_READ or GENERIC_WRITE include the SYNCHRONIZE access right. If an ACE denies GENERIC_WRITE access to a trustee, and the trustee requests GENERIC_READ access, the request will fail because the request implicitly includes SYNCHRONIZE access which is implicitly denied by the ACE. And vice versa, too. Instead of using access-denied ACEs, use access-allowed ACEs to explicitly allow the permitted access rights.

You can request the ACCESS_SYSTEM_SECURITY access right to a file or directory object if you want to read or write the object's SACL. For more information, see *Access-Control Lists (ACLs)* and *SACL Access Right*.

File I/O Reference

File I/O Functions

AreFileApisANSI

The **AreFileApisANSI** function determines whether a set of file I/O functions is using the ANSI or OEM character set code page. This function is useful for 8-bit console input and output operations.

```
BOOL AreFileApisANSI(VOID);
```

Parameters

This function has no parameters.

Return Values

If the set of file I/O functions is using the ANSI code page, the return value is nonzero.

If the set of file I/O functions is using the OEM code page, the return value is zero.

Remarks

The **SetFileApisToOEM** function causes a set of file I/O functions to use the OEM code page. The **SetFileApisToANSI** function causes the same set of file I/O functions to use the ANSI code page. Use the **AreFileApisANSI** function to determine which code page the set of file I/O functions is currently using. For a discussion of these functions' usage, please refer to the Remarks sections of **SetFileApisToOEM** and **SetFileApisToANSI**.

The file I/O functions whose code page is ascertained by **AreFileApisANSI** are those functions exported by KERNEL32.DLL that accept or return a file name.

The functions **SetFileApisToOEM** and **SetFileApisToANSI** set the code page for a process, so **AreFileApisANSI** returns a value indicating the code page of an entire process.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **SetFileApisToANSI**, **SetFileApisToOEM**

CancelIo

The **CancelIo** function cancels all pending input and output (I/O) operations that were issued by the calling thread for the specified file handle. The function does not cancel I/O operations issued for the file handle by other threads.

```
BOOL CancelIo(
  HANDLE hFile  // handle to file
);
```

Parameters

hFile

[in] Handle to a file. The function cancels all pending I/O operations for this file handle.

Return Values

If the function succeeds, the return value is nonzero. All pending I/O operations issued by the calling thread for the file handle were successfully canceled.

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

Remarks

If there are any I/O operations in progress for the specified file handle, and they were issued by the calling thread, the **CancelIo** function cancels them.

Note that the I/O operations must have been issued as overlapped I/O. If they were not, the I/O operations would not have returned to allow the thread to call the **CancelIo** function. Calling the **CancelIo** function with a file handle that was not opened with FILE_FLAG_OVERLAPPED does nothing.

All I/O operations that are canceled will complete with the error ERROR_OPERATION_ABORTED. All completion notifications for the I/O operations will occur normally.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **DeviceIoControl**, **LockFileEx**, **ReadDirectoryChangesW**, **ReadFile**, **ReadFileEx**, **WriteFile**, **WriteFileEx**

CopyFile

The **CopyFile** function copies an existing file to a new file.

The **CopyFileEx** function provides two additional capabilities. **CopyFileEx** can call a specified callback function each time a portion of the copy operation is completed, and **CopyFileEx** can be canceled during the copy operation.

```
BOOL CopyFile(
  LPCTSTR lpExistingFileName, // name of an existing file
```

(continued)

(continued)

```
  LPCTSTR lpNewFileName,      // name of new file
  BOOL bFailIfExists          // operation if file exists
);
```

Parameters

lpExistingFileName
[in] Pointer to a null-terminated string that specifies the name of an existing file.

lpNewFileName
[in] Pointer to a null-terminated string that specifies the name of the new file.

bFailIfExists
[in] Specifies how this operation is to proceed if a file of the same name as that specified by *lpNewFileName* already exists. If this parameter is TRUE and the new file already exists, the function fails. If this parameter is FALSE and the new file already exists, the function overwrites the existing file and succeeds.

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

Security attributes for the existing file are not copied to the new file.

File attributes for the existing file are copied to the new file. For example, if an existing file has the FILE_ATTRIBUTE_READONLY file attribute, a copy created through a call to **CopyFile** will also have the FILE_ATTRIBUTE_READONLY file attribute.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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

File I/O Overview, *File I/O Functions*, **CopyFileEx**, **CreateFile**, **MoveFile**

CopyFileEx

The **CopyFileEx** function copies an existing file to a new file. This function preserves extended attributes, OLE structured storage, NTFS alternate data streams, and file attributes. Security attributes for the existing file are not copied to the new file.

```
BOOL CopyFileEx(
  LPCTSTR lpExistingFileName,         // name of existing file
  LPCTSTR lpNewFileName,              // name of new file
  LPPROGRESS_ROUTINE lpProgressRoutine, // callback function
  LPVOID lpData,                      // callback parameter
  LPBOOL pbCancel,                    // cancel status
  DWORD dwCopyFlags                   // copy options
);
```

Parameters

lpExistingFileName
  [in] Pointer to a null-terminated string that specifies the name of an existing file.

lpNewFileName
  [in] Pointer to a null-terminated string that specifies the name of the new file.

lpProgressRoutine
  [in] Specifies the address of a callback function of type LPPROGRESS_ROUTINE that is called each time another portion of the file has been copied. This parameter can be NULL. For more information on the progress callback function, see **CopyProgressRoutine**.

lpData
  [in] Specifies an argument to be passed to the callback function. This parameter can be NULL.

pbCancel
  [in] Pointer to a Boolean variable that can be used to cancel the operation. If this flag is set to TRUE during the copy operation, the operation is canceled.

dwCopyFlags
  [in] Specifies how the file is to be copied. This parameter can be a combination of the following values.

Value	Meaning
COPY_FILE_FAIL_IF_EXISTS	The copy operation fails immediately if the target file already exists.
COPY_FILE_RESTARTABLE	Progress of the copy is tracked in the target file in case the copy fails. The failed copy can be restarted at a later time by specifying the same values for *lpExistingFileName* and *lpNewFileName* as those used in the call that failed.

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 or later.
Windows 95/98: Unsupported.
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

File I/O Overview, *File I/O Functions*, **CreateFile**, **CopyFile**, **CopyProgressRoutine**, **MoveFile**

CopyProgressRoutine

The **CopyProgressRoutine** function is an application-defined callback function used with the **CopyFileEx** or **MoveFileWithProgress** functions. It is called when a portion of a copy or move operation is completed. The **LPPROGRESS_ROUTINE** type defines a pointer to this callback function. **CopyProgresRoutine** is a placeholder for the application-defined function name.

```
DWORD CALLBACK CopyProgressRoutine(
  LARGE_INTEGER TotalFileSize,            // file size
  LARGE_INTEGER TotalBytesTransferred,    // bytes transferred
  LARGE_INTEGER StreamSize,               // bytes in stream
  LARGE_INTEGER StreamBytesTransferred,   // bytes transferred for stream
  DWORD dwStreamNumber,                   // current stream
  DWORD dwCallbackReason,                 // callback reason
  HANDLE hSourceFile,                     // handle to source file
  HANDLE hDestinationFile,                // handle to destination file
  LPVOID lpData                           // from CopyFileEx
);
```

Parameters

TotalFileSize
[in] Specifies the total size of the file, in bytes.

TotalBytesTransferred
[in] Specifies the total number of bytes transferred from the source file to the destination file since the copy operation began.

StreamSize
[in] Specifies the total size of the current file stream, in bytes.

StreamBytesTransferred
[in] Specifies the total number of bytes in the current stream that have been transferred from the source file to the destination file since the copy operation began.

dwStreamNumber
[in] Handle to the current stream. The stream number is 1 the first time **CopyProgressRoutine** is called.

dwCallbackReason
[in] Specifies the reason that **CopyProgressRoutine** was called. This parameter can be one of the following values.

Value	Meaning
CALLBACK_CHUNK_FINISHED	Another part of the data file was copied.
CALLBACK_STREAM_SWITCH	Another stream was created and is about to be copied. This is the callback reason given when the callback routine is first invoked.

hSourceFile
[in] Handle to the source file.

hDestinationFile
[in] Handle to the destination file

lpData
[in] The argument passed to **CopyProgressRoutine** by the **CopyFileEx** or **MoveFileWithProgress** function.

Return Values

The **CopyProgressRoutine** function should return one of the following values.

Value	Meaning
PROGRESS_CANCEL	Cancel the copy operation and delete the destination file.
PROGRESS_STOP	Stop the copy operation. It can be restarted at a later time.
PROGRESS_QUIET	Continue the copy operation, but stop invoking **CopyProgressRoutine** to report progress.

Remarks

An application can use this information to display a progress bar that shows the total number of bytes copied as a percent of the total file size.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

File I/O Overview, *File I/O Functions*, **CopyFileEx**, **MoveFileWithProgress**

CreateDirectory

The **CreateDirectory** function creates a new directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory.

To specify a template directory, use the **CreateDirectoryEx** function.

```
BOOL CreateDirectory(
  LPCTSTR lpPathName,                        // directory name
  LPSECURITY_ATTRIBUTES lpSecurityAttributes  // SD
);
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path of the directory to be created.

There is a default string size limit for paths of 248 characters. This limit is related to how the **CreateDirectory** function parses paths.

Windows NT/2000: An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of **CreateDirectory** and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing; it lets paths longer than MAX_PATH be used with **CreateDirectoryW**. However, each component in the path cannot be more than MAX_PATH characters long. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\bill_g_1\hotstuff\coolapps" is seen as "\\bill_g_1\hotstuff\coolapps".

lpSecurityAttributes

Windows NT/2000: [in] Pointer to a **SECURITY_ATTRIBUTES** structure. The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new directory. If *lpSecurityAttributes* is NULL, the directory gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect.

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

Some file systems, such as NTFS, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

Windows NT/2000: An application can obtain a handle to a directory by calling **CreateFile** with the FILE_FLAG_BACKUP_SEMANTICS flag set. For a code example, see **CreateFile**.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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

File I/O Overview, *File I/O Functions*, **CreateDirectoryEx**, **CreateFile**, **RemoveDirectory**, **SECURITY_ATTRIBUTES**

CreateDirectoryEx

The **CreateDirectoryEx** function creates a new directory with a specified path that retains the attributes of a specified template directory. If the underlying file system supports security on files and directories, the function applies a specified security descriptor to the new directory. The new directory retains the other attributes of the specified template directory.

```
BOOL CreateDirectoryEx(
  LPCTSTR lpTemplateDirectory,                  // template directory
  LPCTSTR lpNewDirectory,                       // directory name
  LPSECURITY_ATTRIBUTES lpSecurityAttributes    // SD
);
```

Parameters

lpTemplateDirectory

[in] Pointer to a null-terminated string that specifies the path of the directory to use as a template when creating the new directory.

lpNewDirectory

[in] Pointer to a null-terminated string that specifies the path of the directory to be created.

lpSecurityAttributes

Windows NT/2000: [in] Pointer to a **SECURITY_ATTRIBUTES** structure. The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the new directory. If *lpSecurityAttributes* is NULL, the directory gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect.

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 **CreateDirectoryEx** function allows you to create directories that inherit stream information from other directories. This function is useful, for example, when dealing with Macintosh directories, which have a resource stream that is needed to properly identify directory contents as an attribute.

Some file systems, such as NTFS, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

Windows NT/2000: You can obtain a handle to a directory by calling the **CreateFile** function with the FILE_FLAG_BACKUP_SEMANTICS flag set. See **CreateFile** for a code example.

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.

File I/O Overview, *File I/O Functions*, **CreateDirectory**, **CreateFile**, **RemoveDirectory**, **SECURITY_ATTRIBUTES**

CreateFile

The **CreateFile** function creates or opens the following objects and returns a handle that can be used to access the object:

- Consoles
- Communications resources
- Directories (open only)
- Disk devices (Windows NT/2000 only)
- Files
- Mailslots
- Pipes

```
HANDLE CreateFile(
  LPCTSTR lpFileName,                          // file name
  DWORD dwDesiredAccess,                       // access mode
  DWORD dwShareMode,                           // share mode
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,  // SD
  DWORD dwCreationDisposition,                 // how to create
  DWORD dwFlagsAndAttributes,                  // file attributes
  HANDLE hTemplateFile                         // handle to template file
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the object to create or open.

If **lpFileName* is a path, there is a default string size limit of MAX_PATH characters. This limit is related to how the **CreateFile** function parses paths.

Windows NT/2000: You can use paths longer than MAX_PATH characters by calling the wide (W) version of **CreateFile** and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing. This lets you use paths that are nearly 32,000 Unicode characters long. However, each component in the path cannot be more than MAX_PATH characters long. You must use fully-qualified paths with this technique. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\tom_1\hotstuff\coolapps" is seen as "\\tom_1\hotstuff\coolapps".

dwDesiredAccess

[in] Specifies the type of access to the object. An application can obtain read access, write access, read/write access, or device query access. This parameter can be any combination of the following values.

Value	Meaning
0	Specifies device query access to the object. An application can query device attributes without accessing the device.
GENERIC_READ	Specifies read access to the object. Data can be read from the file and the file pointer can be moved. Combine with GENERIC_WRITE for read/write access.
GENERIC_WRITE	Specifies write access to the object. Data can be written to the file and the file pointer can be moved. Combine with GENERIC_READ for read/write access.

dwShareMode

[in] Specifies how the object can be shared. If *dwShareMode* is 0, the object cannot be shared. Subsequent open operations on the object will fail, until the handle is closed.

To share the object, use a combination of one or more of the following values.

Value	Meaning
FILE_SHARE_DELETE	**Windows NT/2000:** Subsequent open operations on the object will succeed only if delete access is requested.
FILE_SHARE_READ	Subsequent open operations on the object will succeed only if read access is requested.
FILE_SHARE_WRITE	Subsequent open operations on the object will succeed only if write access is requested.

lpSecurityAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that determines whether the returned handle can be inherited by child processes. If *lpSecurityAttributes* is NULL, the handle cannot be inherited.

Windows NT/2000: The **lpSecurityDescriptor** member of the structure specifies a security descriptor for the object. If *lpSecurityAttributes* is NULL, the object gets a default security descriptor. The target file system must support security on files and directories for this parameter to have an effect on files.

dwCreationDisposition

[in] Specifies which action to take on files that exist, and which action to take when files do not exist. For more information about this parameter, see the Remarks section. This parameter must be one of the following values.

Value	Meaning
CREATE_NEW	Creates a new file. The function fails if the specified file already exists.
CREATE_ALWAYS	Creates a new file. If the file exists, the function overwrites the file and clears the existing attributes.
OPEN_EXISTING	Opens the file. The function fails if the file does not exist. For a discussion of why you should use the OPEN_EXISTING flag if you are using the **CreateFile** function for devices, see Remarks.
OPEN_ALWAYS	Opens the file, if it exists. If the file does not exist, the function creates the file as if *dwCreationDisposition* were CREATE_NEW.
TRUNCATE_EXISTING	Opens the file. Once opened, the file is truncated so that its size is zero bytes. The calling process must open the file with at least GENERIC_WRITE access. The function fails if the file does not exist.

dwFlagsAndAttributes

[in] Specifies the file attributes and flags for the file.

Any combination of the following attributes is acceptable for the *dwFlagsAndAttributes* parameter, except all other file attributes override FILE_ATTRIBUTE_NORMAL.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file should be archived. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories. This flag has no effect if FILE_ATTRIBUTE_SYSTEM is also specified.
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It is not to be included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.

(continued)

(continued)

Attribute	Meaning
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file is read only. Applications can read the file but cannot write to it or delete it.
FILE_ATTRIBUTE_SYSTEM	The file is part of or is used exclusively by the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

Any combination of the following flags is acceptable for the *dwFlagsAndAttributes* parameter.

Flag	Meaning
FILE_FLAG_WRITE_THROUGH	Instructs the system to write through any intermediate cache and go directly to disk. The system can still cache write operations, but cannot lazily flush them.
FILE_FLAG_OVERLAPPED	Instructs the system to initialize the object, so that operations that take a significant amount of time to process return ERROR_IO_PENDING. When the operation is finished, the specified event is set to the signaled state. When you specify FILE_FLAG_OVERLAPPED, the file read and write functions *must* specify an **OVERLAPPED** structure. That is, when FILE_FLAG_OVERLAPPED is specified, an application *must* perform overlapped reading and writing. When FILE_FLAG_OVERLAPPED is specified, the system does not maintain the file pointer. The file position must be passed as part of the *lpOverlapped* parameter (pointing to an **OVERLAPPED** structure) to the file read and write functions. This flag also enables more than one operation to be performed simultaneously with the handle (a simultaneous read and write operation, for example).

Flag	Meaning
FILE_FLAG_NO_BUFFERING	Instructs the system to open the file with no intermediate buffering or caching. When combined with FILE_FLAG_OVERLAPPED, the flag gives maximum asynchronous performance, because the I/O does not rely on the synchronous operations of the memory manager. However, some I/O operations will take longer, because data is not being held in the cache. An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING: • File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. • File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes. • Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to align buffers on integer multiples of the volume sector size is to use **VirtualAlloc** to allocate the buffers. It allocates memory that is aligned on addresses that are integer multiples of the operating system's memory page size. Because both memory page and volume sector sizes are powers of 2, this memory is also aligned on addresses that are integer multiples of a volume's sector size. An application can determine a volume's sector size by calling the **GetDiskFreeSpace** function.
FILE_FLAG_RANDOM_ACCESS	Indicates that the file is accessed randomly. The system can use this as a hint to optimize file caching.
FILE_FLAG_SEQUENTIAL_SCAN	Indicates that the file is to be accessed sequentially from beginning to end. The system can use this as a hint to optimize file caching. If an application moves the file pointer for random access, optimum caching may not occur; however, correct operation is still guaranteed. Specifying this flag can increase performance for applications that read large files using sequential access. Performance gains can be even more

(continued)

(continued)

Flag	Meaning
	noticeable for applications that read large files mostly sequentially, but occasionally skip over small ranges of bytes.
FILE_FLAG_DELETE_ON_CLOSE	Indicates that the operating system is to delete the file immediately after all of its handles have been closed, not just the handle for which you specified FILE_FLAG_DELETE_ON_CLOSE. Subsequent open requests for the file will fail, unless FILE_SHARE_DELETE is used.
FILE_FLAG_BACKUP_SEMANTICS	**Windows NT/2000:** Indicates that the file is being opened or created for a backup or restore operation. The system ensures that the calling process overrides file security checks, provided it has the necessary privileges. The relevant privileges are SE_BACKUP_NAME and SE_RESTORE_NAME. You can also set this flag to obtain a handle to a directory. A directory handle can be passed to some Win32 functions in place of a file handle.
FILE_FLAG_POSIX_SEMANTICS	Indicates that the file is to be accessed according to POSIX rules. This includes allowing multiple files with names, differing only in case, for file systems that support such naming. Use care when using this option because files created with this flag may not be accessible by applications written for MS-DOS or 16-bit Windows.
FILE_FLAG_OPEN_REPARSE_POINT	Specifying this flag inhibits the reparse behavior of NTFS reparse points. When the file is opened, a file handle is returned, whether the filter that controls the reparse point is operational or not. This flag cannot be used with the CREATE_ALWAYS flag.
FILE_FLAG_OPEN_NO_RECALL	Indicates that the file data is requested, but it should continue to reside in remote storage. It should not be transported back to local storage. This flag is intended for use by remote storage systems or the Hierarchical Storage Management system.

If the **CreateFile** function opens the client side of a named pipe, the *dwFlagsAndAttributes* parameter can also contain Security Quality of Service information. For more information, see *Impersonation Levels*. When the calling application specifies the SECURITY_SQOS_PRESENT flag, the *dwFlagsAndAttributes* parameter can contain one or more of the following values.

Value	Meaning
SECURITY_ANONYMOUS	Specifies to impersonate the client at the Anonymous impersonation level.
SECURITY_IDENTIFICATION	Specifies to impersonate the client at the Identification impersonation level.
SECURITY_IMPERSONATION	Specifies to impersonate the client at the Impersonation level.
SECURITY_DELEGATION	Specifies to impersonate the client at the Delegation impersonation level.
SECURITY_CONTEXT_TRACKING	Specifies that the security tracking mode is dynamic. If this flag is not specified, Security Tracking Mode is static.
SECURITY_EFFECTIVE_ONLY	Specifies that only the enabled aspects of the client's security context are available to the server. If you do not specify this flag, all aspects of the client's security context are available. This flag allows the client to limit the groups and privileges that a server can use while impersonating the client.

hTemplateFile

[in] Specifies a handle with GENERIC_READ access to a template file. The template file supplies file attributes and extended attributes for the file being created.

Windows 95: The *hTemplateFile* parameter must be NULL. If you supply a handle, the call fails and **GetLastError** returns ERROR_NOT_SUPPORTED.

Return Values

If the function succeeds, the return value is an open handle to the specified file. If the specified file exists before the function call and *dwCreationDisposition* is CREATE_ALWAYS or OPEN_ALWAYS, a call to **GetLastError** returns ERROR_ALREADY_EXISTS (even though the function has succeeded). If the file does not exist before the call, **GetLastError** returns zero.

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

Remarks

Use the **CloseHandle** function to close an object handle returned by **CreateFile**.

As noted above, specifying zero for *dwDesiredAccess* allows an application to query device attributes without actually accessing the device. This type of querying is useful, for example, if an application wants to determine the size of a floppy disk drive and the formats it supports without having a floppy in the drive.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Files

When creating a new file, the **CreateFile** function performs the following actions:

- Combines the file attributes and flags specified by *dwFlagsAndAttributes* with FILE_ATTRIBUTE_ARCHIVE.
- Sets the file length to zero.
- Copies the extended attributes supplied by the template file to the new file if the *hTemplateFile* parameter is specified.

When opening an existing file, **CreateFile** performs the following actions:

- Combines the file flags specified by *dwFlagsAndAttributes* with existing file attributes. **CreateFile** ignores the file attributes specified by *dwFlagsAndAttributes*.
- Sets the file length according to the value of *dwCreationDisposition*.
- Ignores the *hTemplateFile* parameter.
- Ignores the **lpSecurityDescriptor** member of the **SECURITY_ATTRIBUTES** structure if the *lpSecurityAttributes* parameter is not NULL. The other structure members are used. The **bInheritHandle** member is the only way to indicate whether the file handle can be inherited.

If you are attempting to create a file on a floppy drive that does not have a floppy disk or a CD-ROM drive that does not have a CD, the system displays a message box asking the user to insert a disk or a CD, respectively. To prevent the system from displaying this message box, call the **SetErrorMode** function with SEM_FAILCRITICALERRORS.

Windows NT/2000: Some file systems, such as NTFS, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new file inherits the compression and encryption attributes of its directory.

You cannot use the **CreateFile** function to set a file's compression state. Use the **DeviceIoControl** function to set a file's compression state.

Pipes

If **CreateFile** opens the client end of a named pipe, the function uses any instance of the named pipe that is in the listening state. The opening process can duplicate the handle as many times as required but, once opened, the named pipe instance cannot be opened by another client. The access specified when a pipe is opened must be compatible with the access specified in the *dwOpenMode* parameter of the **CreateNamedPipe** function. For more information about pipes, see *Pipes*.

Mailslots

If **CreateFile** opens the client end of a mailslot, the function returns INVALID_HANDLE_VALUE if the mailslot client attempts to open a local mailslot before

the mailslot server has created it with the **CreateMailSlot** function. For more information about mailslots, see *Mailslots*.

Communications Resources

The **CreateFile** function can create a handle to a communications resource, such as the serial port COM1. For communications resources, the *dwCreationDisposition* parameter must be OPEN_EXISTING, and the *hTemplate* parameter must be NULL. Read, write, or read/write access can be specified, and the handle can be opened for overlapped I/O. For more information about communications, see *Communications*.

Disk Devices

Volume handles may be opened as noncached at the discretion of the file system, even when the noncached option is not specified with **CreateFile**. You should assume that all Microsoft file systems open volume handles as noncached. The restrictions on noncached I/O for files apply to volumes as well.

A file system may or may not require buffer alignment even though the data is noncached. However, if the noncached option is specified when opening a volume, buffer alignment is enforced regardless of the file system on the volume. It is recommended on all file systems that you open volume handles as noncached and follow the noncached I/O restrictions.

Windows NT/2000: You can use the **CreateFile** function to open a disk drive or a partition on a disk drive. The function returns a handle to the disk device; that handle can be used with the **DeviceIOControl** function. The following requirements must be met in order for such a call to succeed:

- The caller must have administrative privileges for the operation to succeed on a hard disk drive.
- The *lpFileName* string should be of the form **\\.\PHYSICALDRIVE*x*** to open the hard disk *x*. Hard disk numbers start at zero. For example:

String	Meaning
\\.\PHYSICALDRIVE2	Obtains a handle to the third physical drive on the user's computer.

For an example showing how to open a physical drive, see *Calling DeviceIoControl on Windows NT/2000*.

- The *lpFileName* string should be **\\.\x:** to open a floppy drive *x* or a partition *x* on a hard disk. For example:

String	Meaning
\\.**A:**	Obtains a handle to drive A on the user's computer.
\\.**C:**	Obtains a handle to drive C on the user's computer.

There is no trailing backslash in a drive name. The string "\\.\c:\" refers to the root directory of drive C.

On Windows 2000, you can also open a volume by referring to its unique volume name. In this case also, there should be no trailing backslash on the unique volume name.

Note that all I/O buffers must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size), even if the disk device is opened without the FILE_FLAG_NO_BUFFERING flag.

Windows 95: This technique does not work for opening a logical drive. In Windows 95, specifying a string in this form causes **CreateFile** to return an error.

- The *dwCreationDisposition* parameter must have the OPEN_EXISTING value.
- When opening a floppy disk or a partition on a hard disk, you must set the FILE_SHARE_WRITE flag in the *dwShareMode* parameter.

Tape Drives

Windows NT/2000: You can open tape drives using a file name of the form **\\.\TAPE***x* where *x* is a number indicating which drive to open, starting with tape drive 0. To open tape drive 0 in C, use the file name "\\\\.\\TAPE0". For more information on manipulating tape drives for backup or other applications, see *Tape Backup*.

Consoles

The **CreateFile** function can create a handle to console input (CONIN$). If the process has an open handle to it as a result of inheritance or duplication, it can also create a handle to the active screen buffer (CONOUT$). The calling process must be attached to an inherited console or one allocated by the **AllocConsole** function. For console handles, set the **CreateFile** parameters as follows.

Parameters	Value
lpFileName	Use the CONIN$ value to specify console input and the CONOUT$ value to specify console output. CONIN$ gets a handle to the console's input buffer, even if the **SetStdHandle** function redirected the standard input handle. To get the standard input handle, use the **GetStdHandle** function. CONOUT$ gets a handle to the active screen buffer, even if **SetStdHandle** redirected the standard output handle. To get the standard output handle, use **GetStdHandle**.
dwDesiredAccess	GENERIC_READ | GENERIC_WRITE is preferred, but either one can limit access.
dwShareMode	If the calling process inherited the console or if a child process should be able to access the console, this parameter must be FILE_SHARE_READ | FILE_SHARE_WRITE.

lpSecurityAttributes	If you want the console to be inherited, the **bInheritHandle** member of the **SECURITY_ATTRIBUTES** structure must be TRUE.
dwCreationDisposition	You should specify OPEN_EXISTING when using **CreateFile** to open the console.
dwFlagsAndAttributes	Ignored.
hTemplateFile	Ignored.

The following list shows the effects of various settings of *fwdAccess* and *lpFileName*.

lpFileName	*fwdAccess*	**Result**
CON	GENERIC_READ	Opens console for input.
CON	GENERIC_WRITE	Opens console for output.
CON	GENERIC_READ GENERIC_WRITE	**Windows 95:** Causes **CreateFile** to fail; **GetLastError** returns ERROR_PATH_NOT_FOUND. **Windows NT/2000:** Causes **CreateFile** to fail; **GetLastError** returns ERROR_FILE_NOT_FOUND.

Directories

An application cannot create a directory with **CreateFile**; it must call **CreateDirectory** or **CreateDirectoryEx** to create a directory.

Windows NT/2000: You can obtain a handle to a directory by setting the FILE_FLAG_BACKUP_SEMANTICS flag. A directory handle can be passed to some Win32 functions in place of a file handle.

Some file systems, such as NTFS, support compression or encryption for individual files and directories. On volumes formatted for such a file system, a new directory inherits the compression and encryption attributes of its parent directory.

You cannot use the **CreateFile** function to set a directory's compression state. Use the **DeviceIoControl** function to set a directory's compression 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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **AllocConsole**, **CloseHandle**, **ConnectNamedPipe**, **CreateDirectory**, **CreateDirectoryEx**, **CreateNamedPipe**, **DeviceIOControl**, **GetDiskFreeSpace**, **GetOverlappedResult**, **GetStdHandle**, **OpenFile**, **OVERLAPPED**, **ReadFile**, **SECURITY_ATTRIBUTES**, **SetErrorMode**, **SetStdHandle** **TransactNamedPipe**, *Unique Volume Names*, **VirtualAlloc**, **WriteFile**

CreateIoCompletionPort

The **CreateIoCompletionPort** function can associate an instance of an opened file with a newly created or an existing input/output (I/O) completion port, or it can create an I/O completion port without associating it with a file.

Associating an instance of an opened file with an I/O completion port lets an application receive notification of the completion of asynchronous I/O operations involving that file.

```
HANDLE CreateIoCompletionPort (
  HANDLE FileHandle,              // handle to file
  HANDLE ExistingCompletionPort,  // handle to I/O completion port
  ULONG_PTR CompletionKey,        // completion key
  DWORD NumberOfConcurrentThreads // number of threads to execute concurrently
);
```

Parameters

FileHandle

[in] Handle to a file opened for overlapped I/O completion. You must specify the FILE_FLAG_OVERLAPPED flag when using the **CreateFile** function to obtain the handle.

If *FileHandle* specifies INVALID_HANDLE_VALUE, **CreateIoCompletionPort** creates an I/O completion port without associating it with a file. In this case, the *ExistingCompletionPort* parameter must be NULL and the *CompletionKey* parameter is ignored.

ExistingCompletionPort

[in] Handle to the I/O completion port.

If this parameter specifies an existing completion port, the function associates it with the file specified by the *FileHandle* parameter. The function returns the handle of the existing completion port; it does not create a new I/O completion port.

If this parameter is NULL, the function creates a new I/O completion port and associates it with the file specified by *FileHandle*. The function returns the handle to the new I/O completion port.

CompletionKey

[in] Per-file completion key that is included in every I/O completion packet for the specified file.

NumberOfConcurrentThreads

[in] Maximum number of threads that the operating system allows to concurrently process I/O completion packets for the I/O completion port. If this parameter is zero, the system allows as many concurrently running threads as there are processors in the system.

Although any number of threads can call the **GetQueuedCompletionStatus** function to wait for an I/O completion port, each thread is associated with only one completion port at a time. That port is the port that was last checked by the thread.

When a packet is queued to a port, the system first checks how many threads associated with the port are running. If the number of threads running is less than the value of *NumberOfConcurrentThreads*, then one of the waiting threads is allowed to process the packet. When a running thread completes its processing, it calls **GetQueuedCompletionStatus** again, at which point the system can allow another waiting thread to process a packet.

The system also allows a waiting thread to process a packet if a running thread enters any wait state. When the thread in the wait state begins running again, there may be a brief period when the number of active threads exceeds the *NumberOfConcurrentThreads* value. However, the system quickly reduces the number by not allowing any new active threads until the number of active threads falls below the specified value.

Return Values

If the function succeeds, the return value is the handle to the I/O completion port that is associated with the specified file.

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

Remarks

The I/O system can be instructed to send I/O completion notification packets to I/O completion ports, where they are queued. The **CreateIoCompletionPort** function provides this functionality.

After an instance of an open file is associated with an I/O completion port, it cannot be used in the **ReadFileEx** or **WriteFileEx** function. It is best not to share such an associated file through either handle inheritance or a call to the **DuplicateHandle** function. Operations performed with such duplicate handles generate completion notifications.

When you perform an I/O operation with a file handle that has an associated I/O completion port, the I/O system sends a completion notification packet to the completion port when the I/O operation completes. The I/O completion port places the completion packet in a first-in-first-out queue. Use the **GetQueuedCompletionStatus** function to retrieve these queued I/O completion packets.

Threads in the same process can use the **PostQueuedCompletionStatus** function to place I/O completion notification packets in a completion port's queue. By doing so, you can use the port to receive communications from other threads of the process, in addition to receiving I/O completion notification packets from the I/O system.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **DuplicateHandle**, **GetQueuedCompletionStatus**, **PostQueuedCompletionStatus**, **ReadFileEx**, **WriteFileEx**

DefineDosDevice

The **DefineDosDevice** function lets an application define, redefine, or delete MS-DOS device names.

```
BOOL DefineDosDevice(
  DWORD dwFlags,         // options
  LPCTSTR lpDeviceName,  // device name
  LPCTSTR lpTargetPath   // path string
);
```

Parameters

dwFlags

[in] Specifies several controllable aspects of the **DefineDosDevice** function. This parameter can be one or more of the following values.

Value	Meaning
DDD_RAW_TARGET_PATH	If this value is specified, the function does not convert the *lpTargetPath* string from an MS-DOS path to a path, but takes it as is.

DDD_REMOVE_DEFINITION	If this value is specified, the function removes the specified definition for the specified device. To determine which definition to remove, the function walks the list of mappings for the device, looking for a match of *lpTargetPath* against a prefix of each mapping associated with this device. The first mapping that matches is the one removed, and then the function returns. If *lpTargetPath* is NULL or a pointer to a NULL string, the function will remove the first mapping associated with the device and pop the most recent one pushed. If there is nothing left to pop, the device name will be removed. If this value is NOT specified, the string pointed to by the *lpTargetPath* parameter will become the new mapping for this device.
DDD_EXACT_MATCH_ON_REMOVE	If this value is specified along with DDD_REMOVE_DEFINITION, the function will use an exact match to determine which mapping to remove. Use this value to insure that you do not delete something that you did not define.

lpDeviceName
[in] Pointer to an MS-DOS device name string specifying the device the function is defining, redefining, or deleting. The device name string must not have a trailing colon, unless a drive letter (C or D, for example) is being defined, redefined, or deleted. In no case is a trailing backslash allowed.

lpTargetPath
[in] Pointer to a path string that will implement this device. The string is an MS-DOS path string unless the DDD_RAW_TARGET_PATH flag is specified, in which case this string is a path 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

MS-DOS device names are stored as symbolic links in the object name space. The code that converts an MS-DOS path into a corresponding path uses these symbolic links to map MS-DOS devices and drive letters. The **DefineDosDevice** function provides a mechanism whereby an application can modify the symbolic links used to implement the MS-DOS device name space.

To retrieve the current mapping for a particular MS-DOS device name or to obtain a list of all MS-DOS devices known to the system, use the **QueryDosDevice** function.

MS-DOS Device names are global. After it is defined, an MS-DOS device name remains visible to all processes until either it is explicitly removed or the system reboots.

Windows 2000: To define a drive letter assignment that is persistent across boots and not a network share, use the **SetVolumeMountPoint** function. If the volume to be mounted already has a drive letter assigned to it, use the **DeleteVolumeMountPoint** function to remove the assignment.

Note Drive letters and device names defined at system boot time are protected from redefinition and deletion unless the user is an administrator.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
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

File I/O Overview, *File I/O Functions*, **DeleteVolumeMountPoint, QueryDosDevice, SetVolumeMountPoint**

DeleteFile

The **DeleteFile** function deletes an existing file.

```
BOOL DeleteFile(
  LPCTSTR lpFileName   // file name
);
```

Parameters

lpFileName
[in] Pointer to a null-terminated string that specifies the file to be deleted.

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 an application attempts to delete a file that does not exist, the **DeleteFile** function fails.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

Windows 95: The **DeleteFile** function deletes a file even if it is open for normal I/O or as a memory-mapped file. To prevent loss of data, close files before attempting to delete them.

Windows NT/2000: The **DeleteFile** function fails if an application attempts to delete a file that is open for normal I/O or as a memory-mapped file.

To close an open file, use the **CloseHandle** function.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development.*

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **CloseHandle**, **CreateFile**

FileIOCompletionRoutine

The **FileIOCompletionRoutine** function is an application-defined callback function used with the **ReadFileEx** or **WriteFileEx** function. It is called when the asynchronous input and output (I/O) operation is completed or canceled and the calling thread is in an alertable state (using the **SleepEx**, **MsgWaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, or **WaitForMultipleObjectsEx** function with the *fAlertable* flag set to TRUE).

The **LPOVERLAPPED_COMPLETION_ROUTINE** type defines a pointer to this callback function. **FileIOCompletionRoutine** is a placeholder for the application-defined function name.

```
VOID CALLBACK FileIOCompletionRoutine(
  DWORD dwErrorCode,                // completion code
  DWORD dwNumberOfBytesTransfered,  // number of bytes transferred
  LPOVERLAPPED lpOverlapped         // I/O information buffer
);
```

Parameters

dwErrorCode

[in] Specifies the I/O completion status. This parameter can be one of the following values.

Value	Meaning
0	The I/O was successful.
ERROR_HANDLE_EOF	The **ReadFileEx** function tried to read past the end of the file.

dwNumberOfBytesTransfered

[in] Specifies the number of bytes transferred. If an error occurs, this parameter is zero.

lpOverlapped

[in] Pointer to the **OVERLAPPED** structure specified by the asynchronous I/O function.

The system does not use the **hEvent** member of the **OVERLAPPED** structure; the calling application may use this member to pass information to the completion routine. The system does not use the **OVERLAPPED** structure after the completion routine is called, so the completion routine can deallocate the memory used by the overlapped structure.

Return Values

This function does not return a value.

Remarks

The **FileIOCompletionRoutine** function is a placeholder for an application-defined or library-defined function name.

Returning from this function allows another pending I/O completion routine to be called. All waiting completion routines are called before the alertable thread's wait is completed with a return code of WAIT_IO_COMPLETION. The system may call the waiting completion routines in any order. They may or may not be called in the order the I/O functions are completed.

Each time the system calls a completion routine, it uses some of the application's stack. If the completion routine does additional asynchronous I/O and alertable waits, the stack may grow.

For more information, see *Asynchronous Procedure Calls*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

See Also

File I/O Overview, *File I/O Functions*, **OVERLAPPED**, **ReadFileEx**, **SleepEx**, **WaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, **WriteFileEx**

FindClose

The **FindClose** function closes the specified search handle. The **FindFirstFile** and **FindNextFile** functions use the search handle to locate files with names that match a given name.

```
BOOL FindClose(
  HANDLE hFindFile   // file search handle
);
```

Parameters

hFindFile

[in/out] File search handle. This handle must have been previously opened by the **FindFirstFile** 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

After the **FindClose** function is called, the handle specified by the *hFindFile* parameter cannot be used in subsequent calls to either the **FindNextFile** or **FindClose** function.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **FindFirstFile**, **FindNextFile**

FindCloseChangeNotification

The **FindCloseChangeNotification** function stops change notification handle monitoring.

```
BOOL FindCloseChangeNotification(
  HANDLE hChangeHandle   // handle to change notification
);
```

Parameters

hChangeHandle
[in] Handle to a change notification handle created by the **FindFirstChangeNotification** 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

After the **FindCloseChangeNotification** function is called, the handle specified by the *hChangeHandle* parameter cannot be used in subsequent calls to either the **FindNextChangeNotification** or **FindCloseChangeNotification** function.

Change notifications can also be used in the wait 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **FindFirstChangeNotification**, **FindNextChangeNotification**

FindFirstChangeNotification

The **FindFirstChangeNotification** function creates a change notification handle and sets up initial change notification filter conditions. A wait on a notification handle succeeds when a change matching the filter conditions occurs in the specified directory or subtree.

```
HANDLE FindFirstChangeNotification(
  LPCTSTR lpPathName,    // directory name
  BOOL bWatchSubtree,    // monitoring option
  DWORD dwNotifyFilter   // filter conditions
);
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path of the directory to watch.

bWatchSubtree

[in] Specifies whether the function will monitor the directory or the directory tree. If this parameter is TRUE, the function monitors the directory tree rooted at the specified directory; if it is FALSE, it monitors only the specified directory.

dwNotifyFilter

[in] Specifies the filter conditions that satisfy a change notification wait. This parameter can be one or more of the following values.

Value	Meaning
FILE_NOTIFY_CHANGE_FILE_NAME	Any file name change in the watched directory or subtree causes a change notification wait operation to return. Changes include renaming, creating, or deleting a file name.
FILE_NOTIFY_CHANGE_DIR_NAME	Any directory-name change in the watched directory or subtree causes a change notification wait operation to return. Changes include creating or deleting a directory.
FILE_NOTIFY_CHANGE_ATTRIBUTES	Any attribute change in the watched directory or subtree causes a change notification wait operation to return.

(continued)

(continued)

Value	Meaning
FILE_NOTIFY_CHANGE_SIZE	Any file-size change in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change in file size only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_WRITE	Any change to the last write-time of files in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_SECURITY	Any security-descriptor change in the watched directory or subtree causes a change notification wait operation to return.

Return Values

If the function succeeds, the return value is a handle to a find change notification object.

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

Remarks

The wait functions can monitor the specified directory or subtree by using the handle returned by the **FindFirstChangeNotification** function. A wait is satisfied when one of the filter conditions occurs in the monitored directory or subtree.

After the wait has been satisfied, the application can respond to this condition and continue monitoring the directory by calling the **FindNextChangeNotification** function and the appropriate wait function. When the handle is no longer needed, it can be closed by using the **FindCloseChangeNotification** 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.

File I/O Overview, *File I/O Functions*, **FindCloseChangeNotification**, **FindNextChangeNotification**

FindFirstFile

The **FindFirstFile** function searches a directory for a file whose name matches the specified file name. **FindFirstFile** examines subdirectory names as well as file names.

To specify additional attributes to be used in the search, use the **FindFirstFileEx** function.

```
HANDLE FindFirstFile(
  LPCTSTR lpFileName,               // file name
  LPWIN32_FIND_DATA lpFindFileData  // data buffer
);
```

Parameters

lpFileName

Windows 95: [in] Pointer to a null-terminated string that specifies a valid directory or path and file name, which can contain wildcard characters (* and ?). This string must not exceed MAX_PATH characters.

Windows NT/2000: [in] Pointer to a null-terminated string that specifies a valid directory or path and file name, which can contain wildcard characters (* and ?).

There is a default string size limit for paths of MAX_PATH characters. This limit is related to how the **FindFirstFile** function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of **FindFirstFile** and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing; it lets paths longer than MAX_PATH be used with **FindFirstFileW**. However, each component in the path cannot be more than MAX_PATH characters long. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\bill_g_1\hotstuff\coolapps" is seen as "\\bill_g_1\hotstuff\coolapps".

lpFindFileData

[out] Pointer to the **WIN32_FIND_DATA** structure that receives information about the found file or subdirectory.

Return Values

If the function succeeds, the return value is a search handle used in a subsequent call to **FindNextFile** or **FindClose**.

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

Remarks

The **FindFirstFile** function opens a search handle and returns information about the first file whose name matches the specified pattern. After the search handle has been established, use the **FindNextFile** function to search for other files that match the same pattern. When the search handle is no longer needed, close it by using the **FindClose** function.

This function searches for files by name only; it cannot be used for attribute-based searches.

You cannot use root directories as the *lpFileName* input string for **FindFirstFile**, with or without a trailing backslash. To examine files in a root directory, use something like "C:*" and step through the directory with **FindNextFile**. To get the attributes of a root directory, use **GetFileAttributes**. Prepending the string "\\?\" does not allow access to the root directory.

Similarly, on network shares, you can use an *lpFileName* of the form "\\server\service*" but you cannot use an *lpFileName* that points to the share itself, such as "\\server\service".

To examine any directory other than a root directory, use an appropriate path to that directory, with no trailing backslash. For example, an argument of "C:\windows" will return information about the directory "C:\windows", not about any directory or file in "C:\windows". An attempt to open a search with a trailing backslash will always fail.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, *File I/O Functions*, **FindClose**, **FindFirstFileEx**, **FindNextFile**, **GetFileAttributes**, **SetFileAttributes**, **WIN32_FIND_DATA**

FindFirstFileEx

The **FindFirstFileEx** function searches a directory for a file whose name and attributes match those specified in the function call.

```
HANDLE FindFirstFileEx(
  LPCTSTR lpFileName,              // file name
  FINDEX_INFO_LEVELS fInfoLevelId, // information level
```

```
  LPVOID lpFindFileData,            // information buffer
  FINDEX_SEARCH_OPS fSearchOp,      // filtering type
  LPVOID lpSearchFilter,            // search criteria
  DWORD dwAdditionalFlags           // additional search control
);
```

Parameters

lpFileName

Windows 95: [in] Pointer to a null-terminated string that specifies a valid directory or path and file name, which can contain wildcard characters (* and ?). This string must not exceed MAX_PATH characters.

Windows NT/2000: [in] Pointer to a null-terminated string that specifies a valid directory or path and file name, which can contain wildcard characters (* and ?).

There is a default string size limit for paths of MAX_PATH characters. This limit is related to how the **FindFirstFileEx** function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of **FindFirstFileEx** and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing; it lets paths longer than MAX_PATH be used with **FindFirstFileExW**. However, each component in the path cannot be more than MAX_PATH characters long. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\bill_g_1\hotstuff\coolapps" is seen as "\\bill_g_1\hotstuff\coolapps".

fInfoLevelId

[in] Specifies a **FINDEX_INFO_LEVELS** enumeration type that gives the information level of the returned data.

lpFindFileData

[out] Pointer to the buffer that receives the file data. The pointer type is determined by the level of information specified in the *fInfoLevelId* parameter.

fSearchOp

[in] Specifies a **FINDEX_SEARCH_OPS** enumeration type that gives the type of filtering to perform beyond wildcard matching.

lpSearchFilter

[in] If the specified *fSearchOp* needs structured search information, *lpSearchFilter* points to the search criteria. At this time, none of the supported *fSearchOp* values require extended search information. Therefore, this pointer must be NULL.

dwAdditionalFlags

[in] Specifies additional optionsfor controlling the search. You can use FIND_FIRST_EX_CASE_SENSITIVE for case-sensitive searches. The default search is case insensitive. At this time, no other flags are defined.

Return Values

If the function succeeds, the return value is a search handle that can be used in a subsequent call to the **FindNextFile** or **FindClose** functions.

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

Remarks

The **FindFirstFileEx** function is provided to open a search handle and return information about the first file whose name matches the specified pattern and attributes.

If the underlying file system does not support the specified type of filtering, other than directory filtering, **FindFirstFileEx** fails with the error ERROR_NOT_SUPPORTED. The application has to use **FINDEX_SEARCH_OPS** type FileExSearchNameMatch and perform its own filtering.

After the search handle has been established, use it in the **FindNextFile** function to search for other files that match the same pattern with the same filtering being performed. When the search handle is no longer needed, it should be closed using the **FindClose** function.

You cannot use root directories as the *lpFileName* input string for **FindFirstFileEx**, with or without a trailing backslash. To examine files in a root directory, use something like "C:*" and step through the directory with **FindNextFile**. To get the attributes of a root directory, use **GetFileAttributes**. Prepending the string "\\?\" does not allow access to the root directory.

Similarly, on network shares, you can use an *lpFileName* of the form "\\server\service*" but you cannot use an *lpFileName* that points to the share itself, such as "\\server\service".

To examine any directory other than a root directory, use an appropriate path to that directory, with no trailing backslash. For example, an argument of "C:\windows" will return information about the directory "C:\windows", not about any directory or file in "C:\windows". An attempt to open a search with a trailing backslash will always fail.

The call

```
FindFirstFileEx( lpFileName,
                 FindExInfoStandard,
                 lpFindData,
                 FindExSearchNameMatch,
                 NULL,
                 0 );
```

is equivalent to the call

```
FindFirstFile( lpFileName, lpFindData );
```

The following code shows a minimal use of **FindFirstFileEx**. This program is the equivalent of the example shown in **FindFirstFile**.

```
#define _WIN32_WINNT 0x0400

#include "windows.h"
```

```
int
main(int argc, char *argv[])
{
  WIN32_FIND_DATA FindFileData;
  HANDLE hFind;

  printf ("Target file is %s.\n", argv[1]);

  hFind = FindFirstFileEx(argv[1], FindExInfoStandard, &FindFileData,
              FindExSearchNameMatch, NULL, 0 );

  if (hFind == INVALID_HANDLE_VALUE) {
    printf ("Invalid File Handle. Get Last Error reports %d\n", GetLastError ());
  } else {
    printf ("The first file found is %s\n", FindFileData.cFileName);
    FindClose(hFind);
  }

  return (0);
```

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
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

File I/O Overview, *File I/O Functions*, **FINDEX_INFO_LEVELS**, **FINDEX_SEARCH_OPS**, **FindFirstFile**, **FindNextFile**, **FindClose**, **GetFileAttributes**

FindNextChangeNotification

The **FindNextChangeNotification** function requests that the operating system signal a change notification handle the next time it detects an appropriate change.

```
BOOL FindNextChangeNotification(
  HANDLE hChangeHandle   // handle to change notification
);
```

Parameters

hChangeHandle
[in] Handle to a change notification handle created by the **FindFirstChangeNotification** 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

After the **FindNextChangeNotification** function returns successfully, the application can wait for notification that a change has occurred by using the wait functions.

If a change occurs after a call to **FindFirstChangeNotification** but before a call to **FindNextChangeNotification**, the operating system records the change. When **FindNextChangeNotification** is executed, the recorded change immediately satisfies a wait for the change notification.

FindNextChangeNotification should not be used more than once on the same handle without using one of the wait functions. An application may miss a change notification if it uses **FindNextChangeNotification** when there is a change request outstanding.

When *hChangeHandle* is no longer needed, close it by using the **FindCloseChangeNotification** 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.

File I/O Overview, *File I/O Functions*, **FindCloseChangeNotification**, **FindFirstChangeNotification**

FindNextFile

The **FindNextFile** function continues a file search from a previous call to the **FindFirstFile** function.

```
BOOL FindNextFile(
```

```
  HANDLE hFindFile,                // search handle
  LPWIN32_FIND_DATA lpFindFileData // data buffer
);
```

Parameters

hFindFile

[in] Search handle returned by a previous call to the **FindFirstFile** function.

lpFindFileData

[out] Pointer to the **WIN32_FIND_DATA** structure that receives information about the found file or subdirectory. The structure can be used in subsequent calls to **FindNextFile** to refer to the found file or directory.

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**. If no matching files can be found, the **GetLastError** function returns ERROR_NO_MORE_FILES.

Remarks

The **FindNextFile** function searches for files by name only; it cannot be used for attribute-based searches.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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

File I/O Overview, *File I/O Functions*, **FindClose**, **FindFirstFile**, **GetFileAttributes**, **SetFileAttributes**, **WIN32_FIND_DATA**

FlushFileBuffers

The **FlushFileBuffers** function clears the buffers for the specified file and causes all buffered data to be written to the file.

```
BOOL FlushFileBuffers(
  HANDLE hFile  // handle to file
);
```

Parameters

hFile

[in] Handle to an open file. The function flushes this file's buffers. The file handle must have GENERIC_WRITE access to the file.

If *hFile* is a handle to a communications device, the function only flushes the transmit buffer.

If *hFile* is a handle to the server end of a named pipe, the function does not return until the client has read all buffered data from the pipe.

Windows NT/2000: The function fails if *hFile* is a handle to console output. That is because console output is not buffered. The function returns FALSE, and **GetLastError** returns ERROR_INVALID_HANDLE.

Windows 95: The function does nothing if *hFile* is a handle to console output. That is because console output is not buffered. The function returns TRUE, but it does nothing.

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 **WriteFile** and **WriteFileEx** functions typically write data to an internal buffer that the operating system writes to disk on a regular basis. The **FlushFileBuffers** function writes all of the buffered information for the specified file to disk.

You can pass the same file handle used with the **_lread**, **_lwrite**, **_lcreat**, and related functions to **FlushFileBuffers**.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **_lread**, **_lwrite**, **_lcreat**, **WriteFile**, **WriteFileEx**

GetBinaryType

The **GetBinaryType** function determines whether a file is executable, and if so, what type of executable file it is. That last property determines which subsystem an executable file runs under.

```
BOOL GetBinaryType (
  LPCTSTR lpApplicationName,    // full file path
  LPDWORD lpBinaryType          // binary type information
);
```

Parameters

lpApplicationName

[in] Pointer to a null-terminated string that contains the full path of the file whose binary type the function shall determine.

lpBinaryType

[out] Pointer to a variable to receive information about the executable type of the file specified by *lpApplicationName*. The function adjusts a set of bit flags in this variable. The following bit flag constants are defined.

Value	Description
SCS_32BIT_BINARY	A Win32-based application
SCS_DOS_BINARY	An MS-DOS-based application
SCS_OS216_BINARY	A 16-bit OS/2-based application
SCS_PIF_BINARY	A PIF file that executes an MS-DOS-based application
SCS_POSIX_BINARY	A POSIX-based application
SCS_WOW_BINARY	A 16-bit Windows-based application

Return Values

If the file is executable, the return value is nonzero. The function sets the variable pointed to by *lpBinaryType* to indicate the file's executable type.

If the function is not executable, or if the function fails, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

As an alternative, you can obtain the same information by calling the **SHGetFileInfo** function, passing the SHGFI_EXETYPE flag in the *uFlags* parameter.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.

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

File I/O Overview, *File I/O Functions*

GetCurrentDirectory

The **GetCurrentDirectory** function retrieves the current directory for the current process.

```
DWORD GetCurrentDirectory(
  DWORD nBufferLength,  // size of directory buffer
  LPTSTR lpBuffer       // directory buffer
);
```

Parameters

nBufferLength

[in] Specifies the length, in characters, of the buffer for the current directory string. The buffer length must include room for a terminating null character.

lpBuffer

[out] Pointer to the buffer that receives the current directory string. This null-terminated string specifies the absolute path to the current directory.

Return Values

If the function succeeds, the return value specifies the number of characters written 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**.

If the buffer pointed to by *lpBuffer* is not large enough, the return value specifies the required size of the buffer, including the number of bytes necessary for a terminating null character.

Requirements

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.

File I/O Overview, *File I/O Functions*, **CreateDirectory**, **GetSystemDirectory**, **GetWindowsDirectory**, **RemoveDirectory**, **SetCurrentDirectory**

GetDiskFreeSpace

The **GetDiskFreeSpace** function retrieves information about the specified disk, including the amount of free space on the disk.

This function has been superseded by the **GetDiskFreeSpaceEx** function. New Win32-based applications should use **GetDiskFreeSpaceEx**.

```
BOOL GetDiskFreeSpace(
  LPCTSTR lpRootPathName,          // root path
  LPDWORD lpSectorsPerCluster,     // sectors per cluster
  LPDWORD lpBytesPerSector,        // bytes per sector
  LPDWORD lpNumberOfFreeClusters,  // free clusters
  LPDWORD lpTotalNumberOfClusters  // total clusters
);
```

Parameters

lpRootPathName

[in] Pointer to a null-terminated string that specifies the root directory of the disk to return information about. If *lpRootPathName* is NULL, the function uses the root of the current directory. If this parameter is a UNC name, you must follow it with a trailing backslash. For example, you would specify \\MyServer\MyShare as \\MyServer\MyShare\. However, a drive specification such as "C:" cannot have a trailing backslash.

Windows 95: The initial release of Windows 95 does not support UNC paths for the *lpszRootPathName* parameter. To query the free disk space using a UNC path, temporarily map the UNC path to a drive letter, query the free disk space on the drive, then remove the temporary mapping. **Windows 95 OSR2 and later:** UNC paths are supported.

lpSectorsPerCluster

[out] Pointer to a variable for the number of sectors per cluster.

lpBytesPerSector

[out] Pointer to a variable for the number of bytes per sector.

lpNumberOfFreeClusters

[out] Pointer to a variable for the total number of free clusters on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user disk quotas are in use, this value may be less than the total number of free clusters on the disk.

lpTotalNumberOfClusters

[out] Pointer to a variable for the total number of clusters on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user disk quotas are in use, this value may be less than the total number of clusters on the disk.

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 **GetDiskFreeSpaceEx** function lets you avoid the arithmetic required by the **GetDiskFreeSpace** function.

Windows 95: The **GetDiskFreeSpace** function returns incorrect values for volumes that are larger than 2 gigabytes. The function caps the values stored into **lpNumberOfFreeClusters* and **lpTotalNumberOfClusters* so as to never report volume sizes that are greater than 2 gigabytes.

Even on volumes that are smaller than 2 gigabytes, the values stored into **lpSectorsPerCluster*, **lpNumberOfFreeClusters*, and **lpTotalNumberOfClusters* values may be incorrect. That is because the operating system manipulates the values so that computations with them yield the correct volume size.

Windows 95 OSR2 and Windows 98: The **GetDiskFreeSpaceEx** function is available on beginning with Windows 95 OEM Service Release 2 (OSR2). The **GetDiskFreeSpaceEx** function returns correct values for all volumes, including those that are greater than 2 gigabytes.

Requirements

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

File I/O Overview, *File I/O Functions*, **GetDiskFreeSpaceEx**, **GetDriveType**

GetDiskFreeSpaceEx

The **GetDiskFreeSpaceEx** function obtains information about the amount of space available on a disk volume: the total amount of space, the total amount of free space, and the total amount of free space available to the user associated with the calling thread.

```
BOOL GetDiskFreeSpaceEx(
  LPCTSTR lpDirectoryName,                 // directory name
  PULARGE_INTEGER lpFreeBytesAvailable,    // bytes available to caller
  PULARGE_INTEGER lpTotalNumberOfBytes,    // bytes on disk
  PULARGE_INTEGER lpTotalNumberOfFreeBytes // free bytes on disk
);
```

Parameters

lpDirectoryName

[in] Pointer to a null-terminated string that specifies a directory on the disk of interest. This string can be a UNC name. If this parameter is a UNC name, you must follow it with an additional backslash. For example, you would specify \\MyServer\MyShare as \\MyServer\MyShare\.

If *lpDirectoryName* is NULL, the **GetDiskFreeSpaceEx** function obtains information about the disk that contains the current directory.

Note that *lpDirectoryName* does not have to specify the root directory on a disk. The function accepts any directory on the disk.

lpFreeBytesAvailable

[out] Pointer to a variable that receives the total number of free bytes on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user quotas are in use, this value may be less than the total number of free bytes on the disk.

lpTotalNumberOfBytes

[out] Pointer to a variable that receives the total number of bytes on the disk that are available to the user associated with the calling thread.

Windows 2000: If per-user quotas are in use, this value may be less than the total number of bytes on the disk.

lpTotalNumberOfFreeBytes

[out] Pointer to a variable that receives the total number of free bytes on the disk.

This parameter can be NULL.

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 the values obtained by this function are of type **ULARGE_INTEGER**. Be careful not to truncate these values to 32 bits.

Windows 95 OSR2 and Windows 98: The **GetDiskFreeSpaceEx** function is available beginning with Windows 95 OEM Service Release 2 (OSR2).

To determine whether **GetDiskFreeSpaceEx** is available, call **GetModuleHandle** to get the handle to Kernel32.dll. Then you can call **GetProcAddress**.

It is not necessary to call **LoadLibrary** on Kernel32.dll because it is already loaded into every Win32 process's address space.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 OSR2 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

File I/O Overview, *File I/O Functions*, **GetDiskFreeSpace**, **GetModuleHandle**, **GetProcAddress**

GetDriveType

The **GetDriveType** function determines whether a disk drive is a removable, fixed, CD-ROM, RAM disk, or network drive.

```
UINT GetDriveType(
  LPCTSTR lpRootPathName   // root directory
);
```

Parameters

lpRootPathName

[in] Pointer to a null-terminated string that specifies the root directory of the disk to return information about. A trailing backslash is required. If *lpRootPathName* is NULL, the function uses the root of the current directory.

Return Values

The return value specifies the type of drive. It can be one of the following values.

Value	Meaning
DRIVE_UNKNOWN	The drive type cannot be determined.
DRIVE_NO_ROOT_DIR	The root path is invalid. For example, no volume is mounted at the path.
DRIVE_REMOVABLE	The disk can be removed from the drive.
DRIVE_FIXED	The disk cannot be removed from the drive.
DRIVE_REMOTE	The drive is a remote (network) drive.
DRIVE_CDROM	The drive is a CD-ROM drive.
DRIVE_RAMDISK	The drive is a RAM disk.

Requirements

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.

File I/O Overview, *File I/O Functions*, **GetDiskFreeSpace**

GetFileAttributes

The **GetFileAttributes** function returns attributes for a specified file or directory.

This function returns a set of FAT-style attribute information. The **GetFileAttributesEx** function can obtain other sets of file or directory attribute information.

```
DWORD GetFileAttributes(
  LPCTSTR lpFileName   // name of file or directory
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the name of a file or directory.

Windows NT/2000: There is a default string size limit for paths of MAX_PATH characters. This limit is related to how the **GetFileAttributes** function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of **GetFileAttributes** and prepending "\\?\" to the path. The "\\?\" tells the function to turn off path parsing; it lets paths longer than

MAX_PATH be used with **GetFileAttributesW**. However, each component in the path cannot be more than MAX_PATH characters long. This also works with UNC names.

The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\bill_g_1\hotstuff\coolapps" is seen as "\\bill_g_1\hotstuff\coolapps".

Windows 95: The *lpFileName* string must not exceed MAX_PATH characters. Windows 95 does not support the "\\?\" prefix.

Return Values

If the function succeeds, the return value contains the attributes of the specified file or directory.

If the function fails, the return value is –1. To get extended error information, call **GetLastError**.

The attributes can be one or more of the following values.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file or directory is an archive file or directory. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DEVICE	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY	The handle identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data streams in the file are encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file or directory has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.

FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.
FILE_ATTRIBUTE_REPARSE_POINT	The file has an associated reparse point.
FILE_ATTRIBUTE_SPARSE_FILE	The file is a sparse file.
FILE_ATTRIBUTE_SYSTEM	The file or directory is part of, or is used exclusively by, the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

Remarks

When **GetFileAttributes** is called on a directory containing a volume mount point, the file attributes returned are those of the directory where the volume mount point is set, not those of the root directory in the target mounted volume. To obtain the file attributes of the mounted volume, call **GetVolumeNameForVolumeMountPoint** to obtain the name of the target volume. Then use the resulting name in a call to **GetFileAttributes**. The results will be the attributes of the root directory on the target volume.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **DeviceIoControl**, **FindFirstFile**, **FindNextFile**, **GetFileAttributesEx**, **SetFileAttributes**

GetFileAttributesEx

The **GetFileAttributesEx** function obtains attribute information about a specified file or directory.

```
BOOL GetFileAttributesEx(
  LPCTSTR lpFileName,                    // file or directory name
  GET_FILEEX_INFO_LEVELS fInfoLevelId,   // attribute
  LPVOID lpFileInformation               // attribute information
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies a file or directory.

By default, this string is limited to MAX_PATH characters. The limit is related to how the **GetFileAttributesEx** function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of **GetFileAttributesEx** and prepending "\\?\" to the path. However, each component in the path cannot be more than MAX_PATH characters long. The "\\?\" tells the function to turn off path parsing. This technique also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\peanuts\hotstuff\coolapps" is seen as "\\peanuts\hotstuff\coolapps".

fInfoLevelId

[in] Specifies a **GET_FILEEX_INFO_LEVELS** enumeration type that gives the set of attribute information to obtain.

lpFileInformation

[out] Pointer to a buffer that receives the attribute information. The type of attribute information stored into this buffer is determined by the value of *fInfoLevelId*.

Return Values

If the function succeeds, the return value is a nonzero value.

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

Remarks

The **GetFileAttributes** function returns a set of FAT-style attribute information. **GetFileAttributesEx** can obtain other sets of file or directory attribute information.

Currently, **GetFileAttributeEx** obtains a set of standard attributes that is a superset of the FAT-style attribute information.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98.
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

File I/O Overview, *File I/O Functions*, **GetFileAttributes**, **GET_FILEEX_INFO_LEVELS**, **SetFileAttributes**

GetFileInformationByHandle

The **GetFileInformationByHandle** function retrieves information about a specified file.

```
BOOL GetFileInformationByHandle(
  HANDLE hFile,                                    // handle to file
  LPBY_HANDLE_FILE_INFORMATION lpFileInformation   // buffer
);
```

Parameters

hFile

[in] Handle to the file for which to obtain information.

This handle should not be a pipe handle. The **GetFileInformationByHandle** function does not work with pipe handles.

lpFileInformation

[out] Pointer to a **BY_HANDLE_FILE_INFORMATION** structure that receives the file information. The structure can be used in subsequent calls to **GetFileInformationByHandle** to refer to the information about the file.

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

Depending on the underlying network components of the operating system and the type of server connected to, the **GetFileInformationByHandle** function may fail, return partial information, or full information for the given file. In general, you should not use **GetFileInformationByHandle** unless your application is intended to be run on a limited set of operating system configurations.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **BY_HANDLE_FILE_INFORMATION**

GetFileSize

The **GetFileSize** function retrieves the size, in bytes, of a specified file.

This function stores the file size in a **DWORD** value. To retrieve a file size that is larger than a **DWORD** value, use the **GetFileSizeEx** function.

```
DWORD GetFileSize(
  HANDLE hFile,             // handle to file
  LPDWORD lpFileSizeHigh    // high-order word of file size
);
```

Parameters

hFile
[in] Handle to the file whose size is to be returned. This handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

lpFileSizeHigh
[out] Pointer to the variable where the high-order word of the file size is returned. This parameter can be NULL if the application does not require the high-order word.

Return Values

If the function succeeds, the return value is the low-order doubleword of the file size, and, if *lpFileSizeHigh* is non-NULL, the function puts the high-order doubleword of the file size into the variable pointed to by that parameter.

If the function fails and *lpFileSizeHigh* is NULL, the return value is –1. To get extended error information, call **GetLastError**.

If the function fails and *lpFileSizeHigh* is non-NULL, the return value is –1 and **GetLastError** will return a value other than NO_ERROR.

Remarks

You cannot use the **GetFileSize** function with a handle of a nonseeking device such as a pipe or a communications device. To determine the file type for *hFile*, use the **GetFileType** function.

The **GetFileSize** function obtains the uncompressed size of a file. Use the **GetCompressedFileSize** function to obtain the compressed size of a file.

Note that if the return value is –1 and *lpFileSizeHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded or failed.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **GetCompressedFileSize**, **GetFileSizeEx**, **GetFileType**

GetFileSizeEx

The **GetFileSizeEx** function retrieves the size, in bytes, of a specified file.

```
BOOL GetFileSizeEx(
  HANDLE hFile,               // handle to file
  PLARGE_INTEGER lpFileSize   // file size
);
```

Parameters

hFile
: [in] Handle to the file whose size is to be returned. The handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

lpFileSize
: [out] Pointer to a **LARGE_INTEGER** structure that receives the file size.

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: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **LARGE_INTEGER**

GetFileType

The **GetFileType** function returns the type of the specified file.

```
DWORD GetFileType(
  HANDLE hFile    // handle to file
);
```

Parameters

hFile
[in] Handle to an open file.

Return Values

The return value is one of the following values.

Value	Meaning
FILE_TYPE_UNKNOWN	The type of the specified file is unknown.
FILE_TYPE_DISK	The specified file is a disk file.
FILE_TYPE_CHAR	The specified file is a character file, typically an LPT device or a console.
FILE_TYPE_PIPE	The specified file is either a named or anonymous pipe.

Requirements

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.

File I/O Overview, *File I/O Functions*, **GetFileSize**, **GetFileTime**

GetFullPathName

The **GetFullPathName** function retrieves the full path and file name of a specified file.

```
DWORD GetFullPathName(
  LPCTSTR lpFileName,     // file name
  DWORD nBufferLength,    // size of path buffer
  LPTSTR lpBuffer,        // path buffer
  LPTSTR *lpFilePart      // address of file name in path
);
```

Parameters

lpFileName
[in] Pointer to a null-terminated string that specifies a valid file name. This string can use either short (the 8.3 form) or long file names.

nBufferLength
[in] Specifies the size, in characters, of the buffer for the drive and path.

lpBuffer
[out] Pointer to a buffer that contains the null-terminated string for the name of the drive and path.

lpFilePart
[out] Pointer to a buffer that receives the address (in *lpBuffer*) of the final file name component in the path.

Return Values

If the **GetFullPathName** function succeeds, the return value is the length, in characters, of the string copied to *lpBuffer*, not including the terminating null character.

If the *lpBuffer* buffer is too small, the return value is the size of the buffer, in characters, required to hold the path.

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

Remarks

The **GetFullPathName** function merges the name of the current drive and directory with the specified file name to determine the full path and file name of the specified file. It also calculates the address of the file name portion of the full path and file name. This function does not verify that the resulting path and file name are valid or that they refer to an existing file on the associated volume.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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

File I/O Overview, *File I/O Functions*, **GetShortPathName**, **GetTempPath**, **SearchPath**

GetLogicalDrives

The **GetLogicalDrives** function returns a bitmask representing the currently available disk drives.

```
DWORD GetLogicalDrives(VOID);
```

Parameters

This function has no parameters.

Return Values

If the function succeeds, the return value is a bitmask representing the currently available disk drives. Bit position 0 (the least-significant bit) is drive A, bit position 1 is drive B, bit position 2 is drive C, and so on.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **GetLogicalDriveStrings**

GetLogicalDriveStrings

The **GetLogicalDriveStrings** function fills a buffer with strings that specify valid drives in the system.

```
DWORD GetLogicalDriveStrings(
  DWORD nBufferLength,  // size of buffer
  LPTSTR lpBuffer       // drive strings buffer
);
```

Parameters

nBufferLength
[in] Specifies the maximum size, in characters, of the buffer pointed to by *lpBuffer*. This size does not include the terminating null character.

lpBuffer
[out] Pointer to a buffer that receives a series of null-terminated strings, one for each valid drive in the system, that end with a second null character. The following example shows the buffer contents with <null> representing the terminating null character.

```
c:\<null>d:\<null><null>
```

Return Values

If the function succeeds, the return value is the length, in characters, of the strings copied to the buffer, not including the terminating null character. Note that an ANSI-ASCII null character uses one byte, but a Unicode null character uses two bytes.

If the buffer is not large enough, the return value is greater than *nBufferLength*. It is the size of the buffer required to hold the drive strings.

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

Remarks

Each string in the buffer may be used wherever a root directory is required, such as for the **GetDriveType** and **GetDiskFreeSpace** 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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, *File I/O Functions*, **GetDriveType**, **GetDiskFreeSpace**, **GetLogicalDrives**

GetLongPathName

The **GetLongPathName** function converts the specified path to its long form. If no long path is found, this function simply returns the specified name.

```
DWORD GetLongPathName(
  LPCTSTR lpszShortPath, // file name
  LPTSTR lpszLongPath,   // path buffer
  DWORD cchBuffer        // size of path buffer
);
```

Parameters

lpszShortPath
[in] Pointer to a null-terminated path to be converted.

lpszLongPath
[out] Pointer to the buffer to receive the long path. You can use the same buffer you used for the *lpszShortPath* parameter.

cchBuffer
[in] Specifies the size of the buffer, in characters.

Return Values

If the function succeeds, the return value is the length of the string copied to the *lpszLongPath* parameter, in characters. This length does not include the terminating null character.

If *lpszLongPath* is too small, the function returns the size of the buffer required to hold the long path, in characters.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **GetShortPathName**

GetQueuedCompletionStatus

The **GetQueuedCompletionStatus** function attempts to dequeue an I/O completion packet from a specified I/O completion port. If there is no completion packet queued, the function waits for a pending I/O operation associated with the completion port to complete.

```
BOOL GetQueuedCompletionStatus(
  HANDLE CompletionPort,        // handle to completion port
  LPDWORD lpNumberOfBytes,      // bytes transferred
  PULONG_PTR lpCompletionKey,   // file completion key
  LPOVERLAPPED *lpOverlapped,   // buffer
  DWORD dwMilliseconds          // optional timeout value
);
```

Parameters

CompletionPort
[in] Handle to the completion port of interest. To create a completion port, use the **CreateIoCompletionPort** function.

lpNumberOfBytes
[out] Pointer to a variable that receives the number of bytes transferred during an I/O operation that has completed.

lpCompletionKey
[out] Pointer to a variable that receives the completion key value associated with the file handle whose I/O operation has completed. A completion key is a per-file key that is specified in a call to **CreateIoCompletionPort**.

lpOverlapped
[out] Pointer to a variable that receives the address of the **OVERLAPPED** structure that was specified when the completed I/O operation was started.

The following functions can be used to start I/O operations that complete using completion ports. You must pass the function an **OVERLAPPED** structure and a file

handle associated with an completion port (by a call to **CreateIoCompletionPort**) to invoke the I/O completion port mechanism:

ConnectNamedPipe

DeviceIoControl

LockFileEx

ReadDirectoryChangesW

ReadFile

ReadFileVlm

TransactNamedPipe

WaitCommEvent

WriteFile

WriteFileVlm

Even if you have passed the function a file handle associated with a completion port and a valid **OVERLAPPED** structure, an application can prevent completion port notification. This is done by specifying a valid event handle for the **hEvent** member of the **OVERLAPPED** structure, and setting its low-order bit. A valid event handle whose low-order bit is set keeps I/O completion from being queued to the completion port.

dwMilliseconds

[in] Specifies the number of milliseconds that the caller is willing to wait for an completion packet to appear at the completion port. If a completion packet doesn't appear within the specified time, the function times out, returns FALSE, and sets **lpOverlapped* to NULL.

If *dwMilliseconds* is INFINITE, the function will never time out. If *dwMilliseconds* is zero and there is no I/O operation to dequeue, the function will time out immediately.

Return Values

If the function dequeues a completion packet for a successful I/O operation from the completion port, the return value is nonzero. The function stores information in the variables pointed to by the *lpNumberOfBytesTransferred*, *lpCompletionKey*, and *lpOverlapped* parameters.

If **lpOverlapped* is NULL and the function does not dequeue a completion packet from the completion port, the return value is zero. The function does not store information in the variables pointed to by the *lpNumberOfBytesTransferred* and *lpCompletionKey* parameters. To get extended error information, call **GetLastError**. If the function did not dequeue a completion packet because the wait timed out, **GetLastError** returns WAIT_TIMEOUT.

If **lpOverlapped* is not NULL and the function dequeues a completion packet for a failed I/O operation from the completion port, the return value is zero. The function stores information in the variables pointed to by *lpNumberOfBytesTransferred*, *lpCompletionKey*, and *lpOverlapped*. To get extended error information, call **GetLastError**.

Remarks

This function associates a thread with the specified completion port. A thread can be associated with at most one completion port.

The I/O system can be instructed to send completion notification packets to completion ports, where they are queued. The **CreateIoCompletionPort** function provides a mechanism for this.

When you perform an input/output operation with a file handle that has an associated input/output completion port, the I/O system sends a completion notification packet to the completion port when the I/O operation completes. The completion port places the completion packet in a first-in-first-out queue. The **GetQueuedCompletionStatus** function retrieves these queued completion packets.

A server application may have several threads calling **GetQueuedCompletionStatus** for the same completion port. As input operations complete, the operating system queues completion packets to the completion port. If threads are actively waiting in a call to this function, queued requests complete their call. For more information, see *I/O Completion Ports*.

You can call the **PostQueuedCompletionStatus** function to post an completion packet to an completion port. The completion packet will satisfy an outstanding call to the **GetQueuedCompletionStatus** function.

Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **ConnectNamedPipe**, **CreateIoCompletionPort**, **DeviceIoControl**, **LockFileEx**, **OVERLAPPED**, **ReadFile**, **PostQueuedCompletionStatus**, **TransactNamedPipe**, **WaitCommEvent**, **WriteFile**

GetShortPathName

The **GetShortPathName** function obtains the short path form of a specified input path.

```
DWORD GetShortPathName(
  LPCTSTR lpszLongPath,   // null-terminated path string
  LPTSTR lpszShortPath,   // short form buffer
  DWORD cchBuffer         // size of short form buffer
);
```

Parameters

lpszLongPath

[in] Pointer to a null-terminated path string. The function obtains the short form of this path.

lpszShortPath

[out] Pointer to a buffer to receive the null-terminated short form of the path specified by *lpszLongPath*.

cchBuffer

[in] Specifies the size, in characters, of the buffer pointed to by *lpszShortPath*.

Return Values

If the function succeeds, the return value is the length, in characters, of the string copied to *lpszShortPath*, not including the terminating null character.

If the function fails due to the *lpszShortPath* buffer being too small to contain the short path string, the return value is the size, in characters, of the short path string. You need to call the function with a short path buffer that is at least as large as the short path string.

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

Remarks

When an application calls this function and specifies a path on a volume that does not support 8.3 aliases, the function fails with ERROR_INVALID_PARAMETER if the path is longer than 67 bytes.

The path specified by *lpszLongPath* does not have to be a full or a long path. The short form may be longer than the specified path.

If the specified path is already in its short form, there is no need for any conversion, and the function simply copies the specified path to the buffer for the short path.

You can set *lpszShortPath* to the same value as *lpszLongPath*; in other words, you can set the buffer for the short path to the address of the input path string.

You can obtain the long name of a file from the short name by calling the **FindFirstFile** 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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **GetFullPathName**, **FindFirstFile**

GetTempFileName

The **GetTempFileName** function creates a name for a temporary file. The file name is the concatenation of specified path and prefix strings, a hexadecimal string formed from a specified integer, and the .tmp extension.

The specified integer can be nonzero, in which case, the function creates the file name but does not create the file. If you specify zero for the integer, the function creates a unique file name and creates the file in the specified directory.

```
UINT GetTempFileName(
  LPCTSTR lpPathName,       // directory name
  LPCTSTR lpPrefixString,   // file name prefix
  UINT uUnique,             // integer
  LPTSTR lpTempFileName     // file name buffer
);
```

Parameters

lpPathName
[in] Pointer to a null-terminated string that specifies the directory path for the file name. This string must consist of characters in the ANSI character set. Applications typically specify a period (.) or the result of the **GetTempPath** function for this parameter. If this parameter is NULL, the function fails.

lpPrefixString
[in] Pointer to a null-terminated prefix string. The function uses the first three characters of this string as the prefix of the file name. This string must consist of characters in the ANSI character set.

uUnique
[in] Specifies an unsigned integer that the function converts to a hexadecimal string for use in creating the temporary file name.

If *uUnique* is nonzero, the function appends the hexadecimal string to *lpPrefixString* to form the temporary file name. In this case, the function does not create the specified file, and does not test whether the file name is unique.

If *uUnique* is zero, the function uses a hexadecimal string derived from the current system time. In this case, the function uses different values until it finds a unique file name, and then it creates the file in the *lpPathName* directory.

lpTempFileName
[out] Pointer to the buffer that receives the temporary file name. This null-terminated string consists of characters in the ANSI character set. This buffer should be at least the length, in bytes, specified by MAX_PATH to accommodate the path.

Return Values

If the function succeeds, the return value specifies the unique numeric value used in the temporary file name. If the *uUnique* parameter is nonzero, the return value specifies that same number.

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

Remarks

The **GetTempFileName** function creates a temporary file name of the following form:

```
path\preuuuu.TMP
```

The following table describes the file name syntax.

Component	Meaning
path	Path specified by the *lpPathName* parameter
pre	First three letters of the *lpPrefixString* string
uuuu	Hexadecimal value of *uUnique*

When the system shuts down, temporary files whose names have been created by this function are not automatically deleted.

To avoid problems resulting when converting an ANSI string, an application should call the **CreateFile** function to create a temporary file.

If the *uUnique* parameter is zero, **GetTempFileName** attempts to form a unique number based on the current system time. If a file with the resulting file name exists, the number is increased by one and the test for existence is repeated. Testing continues until a unique file name is found. **GetTempFileName** then creates a file by that name and closes it. When *uUnique* is nonzero, no attempt is made to create and open the file.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

! Requirements

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

File I/O Overview, *File I/O Functions*, **CreateFile**, **GetTempPath**

GetTempPath

The **GetTempPath** function retrieves the path of the directory designated for temporary files.

```
DWORD GetTempPath(
  DWORD nBufferLength,  // size of buffer
  LPTSTR lpBuffer       // path buffer
);
```

Parameters

nBufferLength

[in] Specifies the size, in characters, of the string buffer identified by *lpBuffer*.

lpBuffer

[out] Pointer to a string buffer that receives the null-terminated string specifying the temporary file path. The returned string ends with a backslash, for example, C:\TEMP\.

Return Values

If the function succeeds, the return value is the length, in characters, of the string copied to *lpBuffer*, not including the terminating null character. If the return value is greater than *nBufferLength*, the return value is the size of the buffer required to hold the path.

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

Remarks

Windows 95/98: The **GetTempPath** function gets the temporary file path as follows:

1. The path specified by the TMP environment variable.
2. The path specified by the TEMP environment variable, if TMP is not defined or if TMP specifies a directory that does not exist.
3. The current directory, if both TMP and TEMP are not defined or specify nonexistent directories.

Windows NT/2000: The **GetTempPath** function does not verify that the directory specified by the TMP or TEMP environment variables exists. The function gets the temporary file path as follows:

1. The path specified by the TMP environment variable.
2. The path specified by the TEMP environment variable, if TMP is not defined.
3. The Windows directory, if both TMP and TEMP are not defined.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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

File I/O Overview, *File I/O Functions*, **GetTempFileName**

Int32x32To64

The **Int32x32To64** function multiplies two signed 32-bit integers, returning a signed 64-bit integer result. The function performs optimally on all Win32 platforms.

```
LONGLONG Int32x32To64(
  LONG Multiplier,     // first signed 32-bit integer
  LONG Multiplicand    // second signed 32-bit integer
);
```

Parameters

Multiplier
[in] Specifies the first signed 32-bit integer for the multiplication.

Multiplicand
[in] Specifies the second signed 32-bit integer for the multiplication.

Return Values

The return value is the signed 64-bit integer result of the multiplication.

Remarks

This function is implemented on all platforms by optimal inline code: a single multiply instruction that returns a 64-bit result.

Please note that the function's return value is a 64-bit value, not a **LARGE_INTEGER** 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 winnt.h; include windows.h.

File I/O Overview, *File I/O Functions*, **UInt32x32To64**

Int64ShllMod32

The **Int64ShllMod32** function performs a left logical shift operation on an unsigned 64-bit integer value. The function provides improved shifting code for left logical shifts where the shift count is in the range 0–31.

```
ULONGLONG Int64ShllMod32(
  ULONGLONG Value,  // unsigned 64-bit integer
  DWORD ShiftCount  // shift count
);
```

Parameters

Value

[in] Specifies the unsigned 64-bit integer to be shifted.

ShiftCount

[in] Specifies a shift count in the range 0–31.

Return Values

The return value is the unsigned 64-bit integer result of the left logical shift operation.

Remarks

The shift count is the number of bit positions that the value's bits move.

In a left logical shift operation on an unsigned value, the value's bits move to the left, and vacated bits on the right side of the value are set to zero.

A compiler can generate optimal code for a left logical shift operation when the shift count is a constant. However, if the shift count is a variable whose range of values is unknown, the compiler must assume the worst case, leading to non-optimal code: code that calls a subroutine, or code that is inline but branches. By restricting the shift count to the range 0–31, the **Int64ShllMod32** function lets the compiler generate optimal or near-optimal code.

Please note that the **Int64ShllMod32** function's *Value* parameter and return value are 64-bit values, not **LARGE_INTEGER** structures.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

File I/O Overview, *File I/O Functions*, **Int64ShraMod32**, **Int64ShrlMod32**

Int64ShraMod32

The **Int64ShraMod32** function performs a right arithmetic shift operation on a signed 64-bit integer value. The function provides improved shifting code for right arithmetic shifts where the shift count is in the range 0–31.

```
LONGLONG Int64ShraMod32(
  LONGLONG Value,     // signed 64-bit integer
  DWORD ShiftCount    // shift count
);
```

Parameters

Value
[in] Specifies the signed 64-bit integer to be shifted.

ShiftCount
[in] Specifies a shift count in the range 0–31.

Return Values

The return value is the signed 64-bit integer result of the right arithmetic shift operation.

Remarks

The shift count is the number of bit positions that the value's bits move.

In a right arithmetic shift operation on a signed value, the value's bits move to the right, and vacated bits on the left side of the value are set to the value of the sign bit.

A compiler can generate optimal code for a right arithmetic shift operation when the shift count is a constant. However, if the shift count is a variable whose range of values is unknown, the compiler must assume the worst case, leading to non-optimal code: code that calls a subroutine, or code that is inline but branches. By restricting the shift count to the range 0–31, the **Int64ShraMod32** function lets the compiler generate optimal or near-optimal code.

Please note that the **Int64ShraMod32** function's *Value* parameter and return value are 64-bit values, not **LARGE_INTEGER** structures.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

File I/O Overview, *File I/O Functions*, **Int64ShllMod32**, **Int64ShrlMod32**

Int64ShrlMod32

The **Int64ShrlMod32** function performs a right logical shift operation on an unsigned 64-bit integer value. The function provides improved shifting code for right logical shifts where the shift count is in the range 0–31.

```
ULONGLONG Int64ShrlMod32(
  ULONGLONG Value,  // unsigned 64-bit integer
  DWORD ShiftCount  // shift count
);
```

Parameters

Value

[in] Specifies the unsigned 64-bit integer to be shifted.

ShiftCount

[in] Specifies a shift count in the range 0–31.

Return Values

The return value is the unsigned 64-bit integer result of the right logical shift operation.

Remarks

The shift count is the number of bit positions that the value's bits move.

In a right logical shift operation on an unsigned value, the value's bits move to the right, and vacated bits on the left side of the value are set to zero.

A compiler can generate optimal code for a right logical shift operation when the shift count is a constant. However, if the shift count is a variable whose range of values is unknown, the compiler must assume the worst case, leading to non-optimal code: code that calls a subroutine, or code that is inline but branches. By restricting the shift count to the range 0–31, the **Int64ShrlMod32** function lets the compiler generate optimal or near-optimal code.

Note The **Int64ShrlMod32** function's *Value* parameter and return value are 64-bit values, not **LARGE_INTEGER** structures.

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

Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

File I/O Overview, *File I/O Functions*, **Int64ShllMod32**, **Int64ShraMod32**

LockFile

The **LockFile** function locks a region in an open file. Locking a region prevents other processes from accessing the region.

To specify additional options, use the **LockFileEx** function.

```
BOOL LockFile(
  HANDLE hFile,                    // handle to file
  DWORD dwFileOffsetLow,           // low-order word of offset
  DWORD dwFileOffsetHigh,          // high-order word of offset
  DWORD nNumberOfBytesToLockLow,   // low-order word of length
  DWORD nNumberOfBytesToLockHigh   // high-order word of length
);
```

Parameters

hFile
[in] Handle to the file with a region to be locked. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file (or both).

dwFileOffsetLow
[in] Specifies the low-order word of the starting byte offset in the file where the lock should begin.

dwFileOffsetHigh
[in] Specifies the high-order word of the starting byte offset in the file where the lock should begin.

Windows 95/98: *dwFileOffsetHigh* must be 0, the sign extension of the value of *dwFileOffsetLow*. Any other value will be rejected.

nNumberOfBytesToLockLow
[in] Specifies the low-order word of the length of the byte range to be locked.

nNumberOfBytesToLockHigh
[in] Specifies the high-order word of the length of the byte range to be locked.

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

Locking a region of a file gives the locking process exclusive access to the specified region. File locks are not inherited by processes created by the locking process.

Locking a region of a file denies all other processes both read and write access to the specified region. Locking a region that goes beyond the current end-of-file position is not an error.

Locks may not overlap an existing locked region of the file.

The **UnlockFile** function unlocks a file region locked by **LockFile**.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **LockFileEx**, **UnlockFile**

LockFileEx

The **LockFileEx** function locks a byte range within an open file for shared or exclusive access.

```
BOOL LockFileEx(
  HANDLE hFile,                    // handle to file
  DWORD dwFlags,                   // lock options
  DWORD dwReserved,                // reserved
  DWORD nNumberOfBytesToLockLow,   // low-order word of length
  DWORD nNumberOfBytesToLockHigh,  // high-order word of length
  LPOVERLAPPED lpOverlapped        // starting offset
);
```

Parameters

hFile

[in] Handle to an open handle to a file that is to have a range of bytes locked for shared or exclusive access. The handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

dwFlags

[in] Specifies flags that modify the behavior of this function. This parameter may be one or more of the following values:

Value	Meaning
LOCKFILE_FAIL_IMMEDIATELY	If this value is specified, the function returns immediately if it is unable to acquire the requested lock. Otherwise, it waits.
LOCKFILE_EXCLUSIVE_LOCK	If this value is specified, the function requests an exclusive lock. Otherwise, it requests a shared lock.

dwReserved
Reserved parameter; must be set to zero.

nNumberOfBytesToLockLow
[in] Specifies the low-order 32 bits of the length of the byte range to lock.

nNumberOfBytesToLockHigh
[in] Specifies the high-order 32 bits of the length of the byte range to lock.

lpOverlapped
[in] Pointer to an **OVERLAPPED** structure that the function uses with the locking request. This structure, which is required, contains the file offset of the beginning of the lock range. Be sure to initialize the **hEvent** member to a valid handle or zero.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Locking a region of a file is used to acquire shared or exclusive access to the specified region of the file. File locks are not inherited by a new process during process creation.

Locking a portion of a file for exclusive access denies all other processes both read and write access to the specified region of the file. Locking a region that goes beyond the current end-of-file position is not an error.

Locking a portion of a file for shared access denies all processes write access to the specified region of the file, including the process that first locks the region. All processes can read the locked region.

If an exclusive lock is requested for a range of a file that already has a shared or exclusive lock, this call waits until the lock is granted, unless the LOCKFILE_FAIL_IMMEDIATELY flag is specified.

Locks may not overlap an existing locked region of the file.

! Requirements

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

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **CreateFile**, **LockFile**, **OVERLAPPED**, **UnlockFile**, **UnlockFileEx**

MoveFile

The **MoveFile** function renames an existing file or a directory (including all its children).

To specify how to move the file, use the **MoveFileEx** function.

```
BOOL MoveFile(
  LPCTSTR lpExistingFileName, // file name
  LPCTSTR lpNewFileName       // new file name
);
```

Parameters

lpExistingFileName
[in] Pointer to a null-terminated string that names an existing file or directory.

lpNewFileName
[in] Pointer to a null-terminated string that specifies the new name of a file or directory. The new name must not already exist. A new file may be on a different file system or drive. A new directory must be on the same drive.

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 **MoveFile** function will move (rename) either a file or a directory (including all its children) either in the same directory or across directories. The one caveat is that the **MoveFile** function will fail on directory moves when the destination is on a different volume.

Windows 2000: The **MoveFile** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **CopyFile**, **MoveFileEx**, **MoveFileWithProgress**

MoveFileEx

The **MoveFileEx** function renames an existing file or directory.

The **MoveFileWithProgress** function is equivalent to the **MoveFileEx** function, except that **MoveFileWithProgress** allows you to provide a callback function that receives progress notifications.

```
BOOL MoveFileEx(
  LPCTSTR lpExistingFileName,    // file name
  LPCTSTR lpNewFileName,         // new file name
  DWORD dwFlags                  // move options
);
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that names an existing file or directory on the local machine.

If *dwflags* specifies MOVEFILE_DELAY_UNTIL_REBOOT, the file cannot have the read-only attribute.

lpNewFileName

[in] Pointer to a null-terminated string that specifies the new name of *lpExistingFileName* on the local machine.

When moving a file, the destination can be on a different file system or drive. If the destination is on another drive, you must set the MOVEFILE_COPY_ALLOWED flag in *dwFlags*.

When moving a directory, the destination must be on the same drive.

If *dwFlags* specifies MOVEFILE_DELAY_UNTIL_REBOOT, *lpNewFileName* can be NULL. In this case, **MoveFileEx** registers the *lpExistingFileName* file to be deleted when the system reboots. If *lpExistingFileName* refers to a directory, the system removes the directory at reboot only if the directory is empty.

dwFlags

[in] Specifies how to move the file. This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED	If the file is to be moved to a different volume, the function simulates the move by using the **CopyFile** and **DeleteFile** functions. This flag cannot be used with the MOVEFILE_DELAY_UNTIL_REBOOT flag.
MOVEFILE_DELAY_UNTIL_REBOOT	The function does not move the file until the operating system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups. This flag can be used only if the process is in the context of a user who belongs to the administrator group or the LocalSystem account. This flag cannot be used with the MOVEFILE_COPY_ALLOWED flag.
MOVEFILE_REPLACE_EXISTING	If a file of the name specified by *lpNewFileName* already exists, the function replaces its contents with those specified by *lpExistingFileName*.
MOVEFILE_WRITE_THROUGH	The function does not return until the file has actually been moved on the disk. Setting this flag guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation. This flag has no effect if the MOVEFILE_DELAY_UNTIL_REBOOT flag is set.

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 *dwFlags* parameter specifies MOVEFILE_DELAY_UNTIL_REBOOT, **MoveFileEx** stores the locations of the files to be renamed at reboot in the following registry value:

HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager\PendingFileRenameOperations

The function fails if it cannot access the registry.

The **PendingFileRenameOperations** value is of type REG_MULTI_SZ. Each rename operation stores a pair of NULL-terminated strings. The system uses these registry entries to complete the operations at reboot in the same order that they were issued. For example, the following code fragment creates registry entries that delete *szDstFile* and rename *szSrcFile* to be *szDstFile* at reboot:

```
MoveFileEx(szDstFile, NULL, MOVEFILE_DELAY_UNTIL_REBOOT);
MoveFileEx(szSrcFile, szDstFile, MOVEFILE_DELAY_UNTIL_REBOOT);
```

The system stores the following entries in **PendingFileRenameOperations**:

```
szDstFile\0\0
szSrcFile\0szDstFile\0\0
```

Because the actual move and deletion operations specified with the MOVEFILE_DELAY_UNTIL_REBOOT flag take place after the calling application has ceased running, the return value cannot reflect success or failure in moving or deleting the file. Rather, it reflects success or failure in placing the appropriate entries into the registry.

The system deletes a directory tagged for deletion with the MOVEFILE_DELAY_UNTIL_REBOOT flag only if it is empty. To ensure deletion of directories, move or delete all files from the directory before attempting to delete it. Files may be in the directory at boot time, but they must be deleted or moved before the system can delete the directory.

The move and deletion operations are carried out at boot time in the same order they are specified in the calling application. To delete a directory that has files in it at boot time, first delete the files.

Windows 2000: The **MoveFileEx** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

Windows 95/98: The **MoveFileEx** function is not supported. To rename or delete a file at reboot, use the following procedure.

To rename or delete a file on Windows 95/98:

1. Check for the existence of the WININIT.INI file in the Windows directory.
2. If WININIT.INI exists, open it and add new entries to the existing [rename] section. If the file does not exist, create the file and create a [rename] section.
3. Add lines of the following format to the [rename] section:

   ```
   DestinationFileName=SourceFileName
   ```

Both *DestinationFileName* and *SourceFileName* must be short file names. To delete a file, use NUL as the value for *DestinationFileName*.

The system processes WININIT.INI during system boot. After WININIT.INI has been processed, the system names it WININIT.BAK.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
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

File I/O Overview, *File I/O Functions*, **CopyFile**, **DeleteFile**, **GetWindowsDirectory**, **MoveFileWithProgress**, **WritePrivateProfileString**

MoveFileWithProgress

The **MoveFileWithProgress** function moves a file or directory. **MoveFileWithProgress** is equivalent to the **MoveFileEx** function, except that **MoveFileWithProgress** allows you to provide a callback function that receives progress notifications.

```
BOOL MoveFileWithProgress(
  LPCTSTR lpExistingFileName,              // file name
  LPCTSTR lpNewFileName,                   // new file name
  LPPROGRESS_ROUTINE lpProgressRoutine,    // callback function
  LPVOID lpData,                           // parameter for callback
  DWORD dwFlags                            // move options
);
```

Parameters

lpExistingFileName

[in] Pointer to a null-terminated string that names an existing file or directory on the local machine.

lpNewFileName

[in] Pointer to a null-terminated string containing the new name of the file or directory.

When moving a file, *lpNewFileName* can be on a different file system or drive. If *lpNewFileName* is on another drive, you must set the MOVEFILE_COPY_ALLOWED flag in *dwFlags*.

When moving a directory, *lpExistingFileName* and *lpNewFileName* must be on the same drive.

If *dwFlags* specifies MOVEFILE_DELAY_UNTIL_REBOOT, *lpNewFileName* can be NULL. In this case, **MoveFileEx** registers *lpExistingFileName* to be deleted when the system reboots. The function fails if it cannot access the registry to store the information about the delete operation. If *lpExistingFileName* refers to a directory, the system removes the directory at reboot only if the directory is empty.

lpProgressRoutine

[in] Pointer to a **CopyProgressRoutine** callback function that is called each time another portion of the file has been moved. The callback function can be useful if you provide a user interface that displays the progress of the operation. This parameter can be NULL.

lpData

[in] Specifies an argument that **MoveFileWithProgress** passes to the **CopyProgressRoutine** callback function. This parameter can be NULL.

dwFlags

[in] Specifies how to move the file. This parameter can be one or more of the following values.

Value	Meaning
MOVEFILE_COPY_ALLOWED	If the file is to be moved to a different volume, the function simulates the move by using the **CopyFile** and **DeleteFile** functions.
MOVEFILE_CREATE_HARDLINK	Reserved for future use.
MOVEFILE_DELAY_UNTIL_REBOOT	The system does not move the file until the operating system is restarted. The system moves the file immediately after AUTOCHK is executed, but before creating any paging files. Consequently, this parameter enables the function to delete paging files from previous startups. This flag can only be used if the process is in the context of a user who belongs to the administrator group or the LocalSystem account. This flag cannot be used with the MOVEFILE_COPY_ALLOWED flag.

MOVEFILE_FAIL_IF_NOT_TRACKABLE	**MoveFileWithProgress** fails if the source file is a link source, but the file cannot be tracked after the move. This situation can occur if the destination is a volume formatted with the FAT file system.
MOVEFILE_REPLACE_EXISTING	If a file named *lpNewFileName* exists, the function replaces its contents with the contents of the *lpExistingFileName* file.
MOVEFILE_WRITE_THROUGH	**MoveFileWithProgress** does not return until the file has actually been moved on the disk. Setting this flag guarantees that a move performed as a copy and delete operation is flushed to disk before the function returns. The flush occurs at the end of the copy operation. This flag has no effect if the MOVEFILE_DELAY_UNTIL_REBOOT flag is set.

Return Value

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 **MoveFileWithProgress** function coordinates its operation with the link tracking service, so link sources can be tracked as they are moved.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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.

File I/O Overview, *File I/O Functions*, **CopyFileEx**, **CopyProgressRoutine**, **MoveFileEx**

MulDiv

The **MulDiv** function multiplies two 32-bit values and then divides the 64-bit result by a third 32-bit value. The return value is rounded up or down to the nearest integer.

```
int MulDiv(
  int nNumber,        // 32-bit signed multiplicand
  int nNumerator,     // 32-bit signed multiplier
  int nDenominator    // 32-bit signed divisor
);
```

Parameters

nNumber
[in] Specifies the multiplicand.

nNumerator
[in] Specifies the multiplier.

nDenominator
[in] Specifies the number by which the result of the multiplication (*nNumber* * *nNumerator*) is to be divided.

Return Values

If the function succeeds, the return value is the result of the multiplication and division. If either an overflow occurred or *nDenominator* was 0, 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **Int32x32To64**, **UInt32x32To64**

PostQueuedCompletionStatus

The **PostQueuedCompletionStatus** function lets you post an I/O completion packet to an I/O completion port. The I/O completion packet will satisfy an outstanding call to the **GetQueuedCompletionStatus** function. The **GetQueuedCompletionStatus** function returns with the three values passed as the second, third, and fourth parameters of the call to **PostQueuedCompletionStatus**.

```
BOOL PostQueuedCompletionStatus(
  HANDLE CompletionPort,            // handle to an I/O completion port
  DWORD dwNumberOfBytesTransferred, // bytes transferred
  ULONG_PTR dwCompletionKey,        // completion key
  LPOVERLAPPED lpOverlapped         // overlapped buffer
);
```

Parameters

CompletionPort

[in] Handle to an I/O completion port to which the I/O completion packet is to be posted.

dwNumberOfBytesTransferred

[in] Specifies a value to be returned through the *lpNumberOfBytesTransferred* parameter of the **GetQueuedCompletionStatus** function.

dwCompletionKey

[in] Specifies a value to be returned through the *lpCompletionKey* parameter of the **GetQueuedCompletionStatus** function.

lpOverlapped

[in] Specifies a value to be returned through the *lpOverlapped* parameter of the **GetQueuedCompletionStatus** 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

For more information concerning *dwNumberOfBytesTransferred*, *dwCompletionKey*, and *lpOverlapped*, see **GetQueuedCompletionStatus** and the descriptions of the parameters those values are returned through.

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **CreateIoCompletionPort**, **GetQueuedCompletionStatus**, **OVERLAPPED**

QueryDosDevice

The **QueryDosDevice** function lets an application obtain information about MS-DOS device names. The function can obtain the current mapping for a particular MS-DOS device name. The function can also obtain a list of all existing MS-DOS device names.

MS-DOS device names are stored as symbolic links in the object name space. The code that converts an MS-DOS path into a corresponding path uses these symbolic links to map MS-DOS devices and drive letters. The **QueryDosDevice** function provides a mechanism whereby a Win32-based application can query the names of the symbolic links used to implement the MS-DOS device namespace as well as the value of each specific symbolic link.

```
DWORD QueryDosDevice(
  LPCTSTR lpDeviceName, // MS-DOS device name string
  LPTSTR lpTargetPath,  // query results buffer
  DWORD ucchMax         // maximum size of buffer
);
```

Parameters

lpDeviceName

[in] Pointer to an MS-DOS device name string specifying the target of the query. The device name cannot have a trailing backslash.

This parameter can be NULL. In that case, the **QueryDosDevice** function will store a list of all existing MS-DOS device names into the buffer pointed to by *lpTargetPath*.

lpTargetPath

[out] Pointer to a buffer that will receive the result of the query. The function fills this buffer with one or more null-terminated strings. The final null-terminated string is followed by an additional NULL.

If *lpDeviceName* is non-NULL, the function obtains information about the particular MS-DOS device specified by *lpDeviceName*. The first null-terminated string stored into the buffer is the current mapping for the device. The other null-terminated strings represent undeleted prior mappings for the device.

If *lpDeviceName* is NULL, the function obtains a list of all existing MS-DOS device names. Each null-terminated string stored into the buffer is the name of an existing MS-DOS device.

ucchMax
[in] Specifies the maximum number of characters that can be stored into the buffer pointed to by *lpTargetPath*.

Return Values

If the function succeeds, the return value is the number of characters stored into the buffer pointed to by *lpTargetPath*.

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

Remarks

The **DefineDosDevice** function provides a means whereby a Win32-based application can create and modify the symbolic links used to implement the MS-DOS device namespace.

MS-DOS device names are global. Once defined, an MS-DOS device name remains visible to all processes until either it is explicitly removed or the system reboots.

Requirements

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

File I/O Overview, *File I/O Functions*, **DefineDosDevice**

ReadDirectoryChangesW

The **ReadDirectoryChangesW** function returns information describing the changes occurring within a directory.

```
BOOL ReadDirectoryChangesW(
  HANDLE hDirectory,                    // handle to directory
  LPVOID lpBuffer,                      // read results buffer
  DWORD nBufferLength,                  // length of buffer
  BOOL bWatchSubtree,                   // monitoring option
  DWORD dwNotifyFilter,                 // filter conditions
  LPDWORD lpBytesReturned,              // bytes returned
```

(continued)

(continued)

```
  LPOVERLAPPED lpOverlapped,                          // overlapped buffer
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine // completion routine
);
```

Parameters

hDirectory
: [in] Handle to the directory to be monitored. This directory must be opened with the FILE_LIST_DIRECTORY access right.

lpBuffer
: [in/out] Pointer to the formatted buffer in which the read results are to be returned. The structure of this buffer is defined by the **FILE_NOTIFY_INFORMATION** structure. This buffer is filled either synchronously or asynchronously, depending on how the directory is opened and what value is given to the *lpOverlapped* parameter. For more information, see the Remarks section.

nBufferLength
: [in] Specifies the length of the buffer pointed to by the *lpBuffer* parameter.

bWatchSubtree
: [in] Specifies whether the **ReadDirectoryChangesW** function will monitor the directory or the directory tree. If TRUE is specified, the function monitors the directory tree rooted at the specified directory. If FALSE is specified, the function monitors only the directory specified by the *hDirectory* parameter.

dwNotifyFilter
: [in] Specifies filter criteria the function checks to determine if the wait operation has completed. This parameter can be one or more of the following values.

Value	Meaning
FILE_NOTIFY_CHANGE_FILE_NAME	Any file name change in the watched directory or subtree causes a change notification wait operation to return. Changes include renaming, creating, or deleting a file.
FILE_NOTIFY_CHANGE_DIR_NAME	Any directory-name change in the watched directory or subtree causes a change notification wait operation to return. Changes include creating or deleting a directory.
FILE_NOTIFY_CHANGE_ATTRIBUTES	Any attribute change in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SIZE	Any file-size change in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change in file size only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.

Value	Meaning
FILE_NOTIFY_CHANGE_LAST_WRITE	Any change to the last write-time of files in the watched directory or subtree causes a change notification wait operation to return. The operating system detects a change to the last write-time only when the file is written to the disk. For operating systems that use extensive caching, detection occurs only when the cache is sufficiently flushed.
FILE_NOTIFY_CHANGE_LAST_ACCESS	Any change to the last access time of files in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_CREATION	Any change to the creation time of files in the watched directory or subtree causes a change notification wait operation to return.
FILE_NOTIFY_CHANGE_SECURITY	Any security-descriptor change in the watched directory or subtree causes a change notification wait operation to return.

lpBytesReturned
[out] For synchronous calls, this parameter receives the number of bytes transferred into the *lpBuffer* parameter. For asynchronous calls, this parameter is undefined. You must use an asynchronous notification technique to retrieve the number of bytes transferred.

lpOverlapped
[in] Pointer to an **OVERLAPPED** structure that supplies data to be used during asynchronous operation. Otherwise, this value is NULL. The **Offset** and **OffsetHigh** members of this structure are not used.

lpCompletionRoutine
[in] Pointer to a completion routine to be called when the operation has been completed or canceled and the calling thread is in an alertable wait state. For more information about this completion routine, see **FileIOCompletionRoutine**.

Return Values

If the function succeeds, the return value is nonzero. For synchronous calls, this means that the operation succeeded. For asynchronous calls, this indicates that the operation was successfully queued.

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

Remarks

To obtain a handle to a directory, use the **CreateFile** function with FILE_FLAG_BACKUP_SEMANTICS as follows:

```
hDir = CreateFile(
  DirName,                              // pointer to the file name
  FILE_LIST_DIRECTORY,                  // access (read/write) mode
  FILE_SHARE_READ|FILE_SHARE_DELETE,    // share mode
  NULL,                                 // security descriptor
  OPEN_EXISTING,                        // how to create
  FILE_FLAG_BACKUP_SEMANTICS,           // file attributes
  NULL                                  // file with attributes to copy
);
```

A call to **ReadDirectoryChangesW** can be completed synchronously or asynchronously. To specify asynchronous completion, open the directory with **CreateFile** as shown above, but additionally specify the FILE_FLAG_OVERLAPPED attribute in the *dwFlagsAndAttributes* parameter. Then specify an **OVERLAPPED** structure when you call **ReadDirectoryChangesW**.

Upon successful synchronous completion, the *lpBuffer* parameter is a formatted buffer and the number of bytes written to the buffer is available in *lpBytesReturned.* If the number of bytes transferred is zero, the buffer was too small to provide detailed information on all the changes that occurred in the directory or subtree. In this case, you should compute the changes by enumerating the directory or subtree.

For asynchronous completion, you can receive notification in one of three ways:

- Using the **GetOverlappedResult** function. To receive notification through **GetOverlappedResult**, do not specify a completion routine in the *lpCompletionRoutine* parameter. Be sure to set the **hEvent** member of the **OVERLAPPED** structure to a unique event.
- Using the **GetQueuedCompletionStatus** function. To receive notification through **GetQueuedCompletionStatus**, do not specify a completion routine in *lpCompletionRoutine.* Associate the directory handle *hDirectory* with a completion port by calling the **CreateIoCompletionPort** function.
- Using a completion routine. To receive notification through a completion routine, do not associate the directory with a completion port. Specify a completion routine in *lpCompletionRoutine.* This routine is called whenever the operation has been completed or canceled while the thread is in an alertable wait state. The **hEvent** member of the **OVERLAPPED** structure is not used by the system, so you can use it yourself.

Requirements

Windows NT/2000: Requires Windows NT 3.51 SP3 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Declared only as Unicode.

File I/O Overview, *File I/O Functions*, **CreateFile**, **CreateIoCompletionPort**, **FILE_NOTIFY_INFORMATION**, **FileIOCompletionRoutine**, **GetOverlappedResult**, **GetQueuedCompletionStatus**, **OVERLAPPED**

ReadFile

The **ReadFile** function reads data from a file, starting at the position indicated by the file pointer. After the read operation has been completed, the file pointer is adjusted by the number of bytes actually read, unless the file handle is created with the overlapped attribute. If the file handle is created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the read operation.

This function is designed for both synchronous and asynchronous operation. The **ReadFileEx** function is designed solely for asynchronous operation. It lets an application perform other processing during a file read operation.

```
BOOL ReadFile(
  HANDLE hFile,                // handle to file
  LPVOID lpBuffer,             // data buffer
  DWORD nNumberOfBytesToRead,  // number of bytes to read
  LPDWORD lpNumberOfBytesRead, // number of bytes read
  LPOVERLAPPED lpOverlapped    // overlapped buffer
);
```

Parameters

hFile

[in] Handle to the file to be read. The file handle must have been created with GENERIC_READ access to the file.

Windows NT/2000: For asynchronous read operations, *hFile* can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the **CreateFile** function, or a socket handle returned by the **socket** or **accept** function.

Windows 95/98: For asynchronous read operations, *hFile* can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous read operations on mailslots, named pipes, or disk files.

lpBuffer

[out] Pointer to the buffer that receives the data read from the file.

nNumberOfBytesToRead

[in] Specifies the number of bytes to be read from the file.

lpNumberOfBytesRead

[out] Pointer to the variable that receives the number of bytes read. **ReadFile** sets this value to zero before doing any work or error checking. If this parameter is zero when

ReadFile returns TRUE on a named pipe, the other end of the message-mode pipe called the **WriteFile** function with *nNumberOfBytesToWrite* set to zero.

Windows NT/2000: If *lpOverlapped* is NULL, *lpNumberOfBytesRead* cannot be NULL. If *lpOverlapped* is not NULL, *lpNumberOfBytesRead* can be NULL. If this is an overlapped read operation, you can get the number of bytes read by calling **GetOverlappedResult**. If *hFile* is associated with an I/O completion port, you can get the number of bytes read by calling **GetQueuedCompletionStatus**.

Windows 95/98: This parameter cannot be NULL.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure. This structure is required if *hFile* was created with FILE_FLAG_OVERLAPPED.

If *hFile* was opened with FILE_FLAG_OVERLAPPED, the *lpOverlapped* parameter must not be NULL. It must point to a valid **OVERLAPPED** structure. If *hFile* was created with FILE_FLAG_OVERLAPPED and *lpOverlapped* is NULL, the function can incorrectly report that the read operation is complete.

If *hFile* was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, the read operation starts at the offset specified in the **OVERLAPPED** structure and **ReadFile** may return before the read operation has been completed. In this case, **ReadFile** returns FALSE and the **GetLastError** function returns ERROR_IO_PENDING. This allows the calling process to continue while the read operation finishes. The event specified in the **OVERLAPPED** structure is set to the signaled state upon completion of the read operation.

If *hFile* was not opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is NULL, the read operation starts at the current file position and **ReadFile** does not return until the operation has been completed.

Windows NT/2000: If *hFile* is not opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, the read operation starts at the offset specified in the **OVERLAPPED** structure. **ReadFile** does not return until the read operation has been completed.

Windows 95/98: For operations on files, disks, pipes, or mailslots, this parameter must be NULL; a pointer to an **OVERLAPPED** structure causes the call to fail. However, Windows 95/98 supports overlapped I/O on serial and parallel ports.

Return Values

The **ReadFile** function returns when one of the following is true: a write operation completes on the write end of the pipe, the number of bytes requested has been read, or an error occurs.

If the function succeeds, the return value is nonzero.

If the return value is nonzero and the number of bytes read is zero, the file pointer was beyond the current end of the file at the time of the read operation. However, if the file was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, the return value is FALSE and **GetLastError** returns ERROR_HANDLE_EOF when the file pointer goes beyond the current end of file.

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

Remarks

If part of the file is locked by another process and the read operation overlaps the locked portion, this function fails.

An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

- File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. To determine a volume's sector size, call the **GetDiskFreeSpace** function.
- File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

Accessing the input buffer while a read operation is using the buffer may lead to corruption of the data read into that buffer. Applications must not read from, write to, reallocate, or free the input buffer that a read operation is using until the read operation completes.

Characters can be read from the console input buffer by using **ReadFile** with a handle to console input. The console mode determines the exact behavior of the **ReadFile** function.

If a named pipe is being read in message mode and the next message is longer than the *nNumberOfBytesToRead* parameter specifies, **ReadFile** returns FALSE and **GetLastError** returns ERROR_MORE_DATA. The remainder of the message may be read by a subsequent call to the **ReadFile** or **PeekNamedPipe** function.

When reading from a communications device, the behavior of **ReadFile** is governed by the current communication time-outs as set and retrieved using the **SetCommTimeouts** and **GetCommTimeouts** functions. Unpredictable results can occur if you fail to set the time-out values. For more information about communication time-outs, see **COMMTIMEOUTS**.

If **ReadFile** attempts to read from a mailslot whose buffer is too small, the function returns FALSE and **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

If the anonymous write pipe handle has been closed and **ReadFile** attempts to read using the corresponding anonymous read pipe handle, the function returns FALSE and **GetLastError** returns ERROR_BROKEN_PIPE.

The **ReadFile** function may fail and return ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY whenever there are too many outstanding asynchronous I/O requests.

The **ReadFile** code to check for the end-of-file condition (eof) differs for synchronous and asynchronous read operations.

When a synchronous read operation reaches the end of a file, **ReadFile** returns TRUE and sets **lpNumberOfBytesRead* to zero. The following sample code tests for end-of-file for a synchronous read operation:

```
// Attempt a synchronous read operation.
bResult = ReadFile(hFile, &inBuffer, nBytesToRead, &nBytesRead, NULL) ;
// Check for end of file.
if (bResult &&  nBytesRead == 0, )
{
    // we're at the end of the file
}
```

An asynchronous read operation can encounter the end of a file during the initiating call to **ReadFile**, or during subsequent asynchronous operation.

If EOF is detected at **ReadFile** time for an asynchronous read operation, **ReadFile** returns FALSE and **GetLastError** returns ERROR_HANDLE_EOF.

If EOF is detected during subsequent asynchronous operation, the call to **GetOverlappedResult** to obtain the results of that operation returns FALSE and **GetLastError** returns ERROR_HANDLE_EOF.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to read from a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the **SetErrorMode** function with SEM_NOOPENFILEERRORBOX.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

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

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **CancelIo**, **CreateFile**, **GetCommTimeouts**, **GetOverlappedResult**, **GetQueuedCompletionStatus**, **OVERLAPPED**, **PeekNamedPipe**, **ReadFileEx**, **SetCommTimeouts**, **SetErrorMode**, **WriteFile**

ReadFileEx

The **ReadFileEx** function reads data from a file asynchronously. It is designed solely for asynchronous operation, unlike the **ReadFile** function, which is designed for both synchronous and asynchronous operation. **ReadFileEx** lets an application perform other processing during a file read operation.

The **ReadFileEx** function reports its completion status asynchronously, calling a specified completion routine when reading is completed or canceled and the calling thread is in an alertable wait state.

```
BOOL ReadFileEx(
  HANDLE hFile,                                     // handle to file
  LPVOID lpBuffer,                                  // data buffer
  DWORD nNumberOfBytesToRead,                       // number of bytes to read
  LPOVERLAPPED lpOverlapped,                        // offset
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine // completion routine
);
```

Parameters

hFile

[in] Handle to the file to be read. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and must have GENERIC_READ access to the file.

Windows NT/2000: This parameter can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the **CreateFile** function, or a socket handle returned by the **socket** or **accept** function.

Windows 95/98: This parameter can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous read operations on mailslots, named pipes, or disk files.

lpBuffer

[out] Pointer to a buffer that receives the data read from the file.

This buffer must remain valid for the duration of the read operation. The application should not use this buffer until the read operation is completed.

nNumberOfBytesToRead
[in] Specifies the number of bytes to be read from the file.

lpOverlapped
[in] Pointer to an **OVERLAPPED** data structure that supplies data to be used during the asynchronous (overlapped) file read operation.

If the file specified by *hFile* supports the concept of byte offsets, the caller of **ReadFileEx** must specify a byte offset within the file at which reading should begin. The caller specifies the byte offset by setting the **OVERLAPPED** structure's **Offset** and **OffsetHigh** members.

The **ReadFileEx** function ignores the **OVERLAPPED** structure's **hEvent** member. An application is free to use that member for its own purposes in the context of a **ReadFileEx** call. **ReadFileEx** signals completion of its read operation by calling, or queuing a call to, the completion routine pointed to by *lpCompletionRoutine*, so it does not need an event handle.

The **ReadFileEx** function does use the **OVERLAPPED** structure's **Internal** and **InternalHigh** members. An application should not set these members.

The **OVERLAPPED** data structure pointed to by *lpOverlapped* must remain valid for the duration of the read operation. It should not be a variable that can go out of scope while the file read operation is in progress.

lpCompletionRoutine
[in] Pointer to the completion routine to be called when the read operation is complete and the calling thread is in an alertable wait state. For more information about the completion routine, see **FileIOCompletionRoutine**.

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**.

If the function succeeds, the calling thread has an asynchronous input/output (I/O) operation pending: the overlapped read operation from the file. When this I/O operation completes, and the calling thread is blocked in an alertable wait state, the system calls the function pointed to by *lpCompletionRoutine*, and the wait state completes with a return code of WAIT_IO_COMPLETION.

If the function succeeds, and the file reading operation completes, but the calling thread is not in an alertable wait state, the system queues the completion routine call, holding the call until the calling thread enters an alertable wait state. For information about alertable waits and overlapped input/output operations, see *Synchronization and Overlapped Input and Output*.

If **ReadFileEx** attempts to read past the end of the file, the function returns zero, and **GetLastError** returns ERROR_HANDLE_EOF.

Remarks

When using **ReadFileEx** you should check **GetLastError** even when the function returns "success" to check for conditions that are "successes" but have some outcome you might want to know about. For example, a buffer overflow when calling **ReadFileEx** will return TRUE, but **GetLastError** will report the overflow with ERROR_MORE_DATA. If the function call is successful and there are no warning conditions, **GetLastError** will return ERROR_SUCCESS.

An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

- File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. To determine a volume's sector size, call the **GetDiskFreeSpace** function.
- File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

If a portion of the file specified by *hFile* is locked by another process, and the read operation specified in a call to **ReadFileEx** overlaps the locked portion, the call to **ReadFileEx** fails.

If **ReadFileEx** attempts to read data from a mailslot whose buffer is too small, the function returns FALSE, and **GetLastError** returns ERROR_INSUFFICIENT_BUFFER.

Accessing the input buffer while a read operation is using the buffer may lead to corruption of the data read into that buffer. Applications must not read from, write to, reallocate, or free the input buffer that a read operation is using until the read operation completes.

The **ReadFileEx** function may fail if there are too many outstanding asynchronous I/O requests. In the event of such a failure, **GetLastError** can return ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to read from a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the

system from displaying this message box, call the **SetErrorMode** function with SEM_NOOPENFILEERRORBOX.

An application uses the **MsgWaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, **WaitForMultipleObjectsEx**, and **SleepEx** functions to enter an alertable wait state. For more information about alertable waits and overlapped input/output, refer to those functions' reference and Synchronization.

Windows 95/98: On this platform, neither **ReadFileEx** nor **WriteFileEx** can be used by the comm ports to communicate. However, you can use **ReadFile** and **WriteFile** to perform asynchronous communication.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **CancelIo**, **CreateFile**, **FileIOCompletionRoutine**, **MsgWaitForMultipleObjectsEx**, **OVERLAPPED**, **ReadFile**, **SetErrorMode**, **SleepEx**, **WaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, **WriteFileEx**

ReadFileScatter

The **ReadFileScatter** function reads data from a file and stores the data into a set of buffers.

The **ReadFileScatter** function starts reading data from the file at a position specified by an **OVERLAPPED** structure.

The **ReadFileScatter** function operates asynchronously.

```
BOOL ReadFileScatter(
  HANDLE hFile,                           // handle to file
  FILE_SEGMENT_ELEMENT aSegmentArray[ ],  // array of buffer pointers
  DWORD nNumberOfBytesToRead,             // number of bytes to read
  LPDWORD lpReserved,                     // reserved; must be NULL
  LPOVERLAPPED lpOverlapped               // OVERLAPPED structure
);
```

Parameters

hFile
[in] Handle to the file to be read.

This file handle must have been created using GENERIC_READ to specify read access to the file, FILE_FLAG_OVERLAPPED to specify asynchronous I/O, and FILE_FLAG_NO_BUFFERING to specify non-cached I/O.

aSegmentArray

[in] Pointer to an array of **FILE_SEGMENT_ELEMENT** pointers to buffers. The function stores the data it reads from the file into this set of buffers.

Each buffer should be the size of a system memory page. Each buffer should be aligned on a system memory page size boundary.

A **FILE_SEGMENT_ELEMENT** pointer is a 64-bit value. The **ReadFileScatter** function uses all 64 bits. Because the operating systems do not currently support 64-bit memory addressing, you must explicitly set the upper 32 bits of each **FILE_SEGMENT_ELEMENT** pointer. This is best done with a cast to **__ptr64**. For compilers where **__ptr64** is not available, setting the upper 32 bits of the pointer to zero is a less elegant workaround.

The function stores the data into the buffers in a sequential manner: it stores data into the first buffer, then into the second buffer, then into the next, filling each buffer, until there is no more data or there are no more buffers.

The final element of the array should be a 64-bit NULL pointer.

nNumberOfBytesToRead

[in] Specifies the number of bytes to read from the file.

lpReserved

[in] This parameter is reserved for future use. You must set it to NULL.

lpOverlapped

[in] Pointer to an **OVERLAPPED** data structure.

The **ReadFileScatter** function requires a valid **OVERLAPPED** structure. The *lpOverlapped* parameter cannot be NULL.

The **ReadFileScatter** function starts reading data from the file at a position specified by the **Offset** and **OffsetHigh** members of the **OVERLAPPED** structure.

The **ReadFileScatter** function may return before the read operation has completed. In that case, the **ReadFileScatter** function returns the value zero, and the **GetLastError** function returns the value ERROR_IO_PENDING. This asynchronous operation of **ReadFileScatter** lets the calling process continue while the read operation completes. You can call the **GetOverlappedResult**, **HasOverlappedIoCompleted**, or **GetQueuedCompletionStatus** function to obtain information about the completion of the read operation.

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 the **GetLastError** function.

If the function attempts to read past the end of the file, the function returns zero, and **GetLastError** returns ERROR_HANDLE_EOF.

If the function returns before the read operation has completed, the function returns zero, and **GetLastError** returns ERROR_IO_PENDING.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP2 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **GetOverlappedResult**, **GetQueuedCompletionStatus**, **HasOverlappedIoCompleted**, **OVERLAPPED**, **ReadFile**, **ReadFileEx**, **ReadFileVlm**, **WriteFileGather**

RemoveDirectory

The **RemoveDirectory** function deletes an existing empty directory.

```
BOOL RemoveDirectory(
  LPCTSTR lpPathName   // directory name
);
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path of the directory to be removed. The path must specify an empty directory, and the calling process must have delete access to the directory.

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

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development.*

Requirements

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

Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **CreateDirectory**

ReplaceFile

The **ReplaceFile** function replaces one file with another file, with the option of creating a backup copy of the original file. The replacement file assumes the name of the replaced file and its identity. The operation is atomic; either all data is saved to a file, or the original file is left unchanged.

```
BOOL ReplaceFile(
  LPCTSTR lpReplacedFileName,    // file name
  LPCTSTR lpReplacementFileName, // replacement file
  LPCTSTR lpBackupFileName,      // optional backup file
  DWORD dwReplaceFlags,          // replace options
  LPVOID lpExclude,              // reserved
  LPVOID lpReserved              // reserved
);
```

Parameters

lpReplacedFileName

[in] Pointer to a null-terminated string that specifies the name of the file that will be replaced by the *lpReplacementFileName* file.

This file is opened with the GENERIC_READ, DELETE, and SYNCHRONIZE access rights. The sharing mode is FILE_SHARE_READ | FILE_SHARED_WRITE | FILE_SHARE_DELETE.

lpReplacementFileName

[in] Pointer to a null-terminated string that specifies the name of the file that will replace the *lpReplacedFileName* file.

The function attemps to open this file with the SYNCHRONIZE, GENERIC_READ, GENERIC_WRITE, DELETE, and WRITE_DAC access rights. If this fails, the function attemps to open the file with the SYNCHRONIZE, GENERIC_READ, DELETE, and WRITE_DAC access rights. No sharing mode is specified.

lpBackupFileName

[in] Pointer to a null-terminated string that specifies the name of the file that will serve as a backup copy of the *lpReplacedFileName* file. If this parameter is NULL, no backup file is created.

dwReplaceFlags
[in] Specifies how the file is to be replaced. This parameter can be one or more of the following values.

Value	Meaning
REPLACEFILE_WRITE_THROUGH	Guarantees that information copied from the replaced file is flushed to disk before the function returns.
REPLACEFILE_IGNORE_MERGE_ERRORS	Ignores errors that occur while merging information from the replaced file to the replacement file.

lpExclude
Reserved for future use.

lpReserved
Reserved for future 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**. **GetLastError** may return one of the following error codes that are specific to this function.

Error Code	Meaning
ERROR_UNABLE_TO_REMOVE_REPLACED	The replaced file could not be deleted. The replaced and replacement files retain their original file names.
ERROR_UNABLE_TO_MOVE_REPLACEMENT	The replacement file could not be renamed. If *lpBackupFileName* was specified, the replaced and replacement files retain their original file names. Otherwise, the replaced file no longer exists and the replacement file exists under its original name.
ERROR_UNABLE_TO_MOVE_REPLACEMENT_2	The replacement file could not be renamed. The replacement file still exists with its original name, but the replaced file exists with the name specified by *lpBackupFileName*. This error occurs only if *lpBackupFileName* is not NULL.

If any other error is returned, such as ERROR_INVALID_PARAMETER, the replaced and replacement files will retain their original file names.

Remarks

The **ReplaceFile** function combines several of steps into a single function. An application can call **ReplaceFile** instead of calling separate functions to save the data to a new file, rename the original file using a temporary name, rename the new file to have the same name as the original file, and delete the original file. Another advantage is that **ReplaceFile** not only copies the new file data, but also preserves the following attributes of the original file:

- creation time
- short file name
- object identifier
- ACLs
- encryption
- compression
- named streams not already in the replacement file

For example, if the replacement file is encrypted, but the replaced file is not encrypted, the resulting file is not encrypted.

The backup file, replaced file, and replacement file must all reside on the same volume.

To delete or rename a file, you must have either delete permission on the file or delete child permission in the parent directory. If you set up a directory with all access except delete and delete child and the ACLs of new files are inherited, then you should be able to create a file without being able to delete it. However, you can then create a file, and you will get all the access you request on the handle returned to you at the time you create the file. If you requested delete permission at the time you created the file, you could delete or rename the file with that handle but not with any other.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File I/O Overview, *File I/O Functions*, **CopyFile**, **CopyFileEx**, **MoveFile**, **MoveFileEx**, **MoveFileWithProgress**

SearchPath

The **SearchPath** function searches for the specified file.

```
DWORD SearchPath(
  LPCTSTR lpPath,         // search path
  LPCTSTR lpFileName,     // file name
  LPCTSTR lpExtension,    // file extension
  DWORD nBufferLength,    // size of buffer
  LPTSTR lpBuffer,        // found file name buffer
  LPTSTR *lpFilePart      // file component
);
```

Parameters

lpPath

[in] Pointer to a null-terminated string that specifies the path to be searched for the file. If this parameter is NULL, the function searches for a matching file in the following directories in the following sequence:

1. The directory from which the application loaded.
2. The current directory.
3. **Windows 95:** The Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory.

   Windows NT/2000: The 32-bit Windows system directory. Use the **GetSystemDirectory** function to get the path of this directory. The name of this directory is SYSTEM32.
4. **Windows NT/2000:** The 16-bit Windows system directory. There is no Win32 function that obtains the path of this directory, but it is searched. The name of this directory is SYSTEM.
5. The Windows directory. Use the **GetWindowsDirectory** function to get the path of this directory.
6. The directories that are listed in the PATH environment variable.

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the file to search for.

lpExtension

[in] Pointer to a null-terminated string that specifies an extension to be added to the file name when searching for the file. The first character of the file name extension must be a period (.). The extension is added only if the specified file name does not end with an extension.

If a file name extension is not required or if the file name contains an extension, this parameter can be NULL.

nBufferLength
[in] Specifies the length, in characters, of the buffer that receives the valid path and file name.

lpBuffer
[out] Pointer to the buffer that receives the path and file name of the file found.

lpFilePart
[out] Pointer to the variable that receives the address (within *lpBuffer*) of the last component of the valid path and file name, which is the address of the character immediately following the final backslash (\) in the path.

Return Values

If the function succeeds, the value returned is the length, in characters, of the string copied to the buffer, not including the terminating null character. If the return value is greater than *nBufferLength*, the value returned is the size of the buffer required to hold the path.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, *File I/O Functions*, **FindFirstFile**, **FindNextFile**, **GetSystemDirectory**, **GetWindowsDirectory**

SetCurrentDirectory

The **SetCurrentDirectory** function changes the current directory for the current process.

```
BOOL SetCurrentDirectory(
  LPCTSTR lpPathName   // new directory name
);
```

Parameters

lpPathName

[in] Pointer to a null-terminated string that specifies the path to the new current directory. This parameter may be a relative path or a full path. In either case, the full path of the specified directory is calculated and stored as the current directory.

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

Each process has a single current directory made up of two parts:

- A disk designator that is either a drive letter followed by a colon, or a server name and share name (*servername**sharename*)
- A directory on the disk designator

Requirements

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

File I/O Overview, *File I/O Functions*, **GetCurrentDirectory**

SetEndOfFile

The **SetEndOfFile** function moves the end-of-file (EOF) position for the specified file to the current position of the file pointer.

```
BOOL SetEndOfFile(
  HANDLE hFile   // handle to file
);
```

Parameters

hFile

[in] Handle to the file to have its EOF position moved. The file handle must have been created with GENERIC_WRITE access to the file.

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

This function can be used to truncate or extend a file. If the file is extended, the contents of the file between the old EOF position and the new position are not defined.

If you called **CreateFileMapping** to create a file-mapping object for *hFile*, you must first call **UnmapViewOfFile** to unmap all views and call **CloseHandle** to close the file-mapping object before you can call **SetEndOfFile**.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **CloseHandle**, **CreateFile**, **CreateFileMapping**, **UnmapViewOfFile**

SetFileApisToANSI

The **SetFileApisToANSI** function causes a set of file I/O functions to use the ANSI character set code page. This function is useful for 8-bit console input and output operations.

```
VOID SetFileApisToANSI(VOID);
```

Parameters

This function has no parameters.

Return Values

This function has no return value.

Remarks

The file I/O functions whose code page is set by **SetFileApisToANSI** are those functions exported by KERNEL32.DLL that accept or return a file name. **SetFileApisToANSI** sets the code page per process, rather than per thread or per computer.

The **SetFileApisToANSI** function complements the **SetFileApisToOEM** function, which causes the same set of file I/O functions to use the OEM character set code page.

The 8-bit console functions use the OEM code page by default. All other functions use the ANSI code page by default. This means that strings returned by the console functions may not be processed correctly by other functions, and vice versa. For example, if the **FindFirstFileA** function returns a string that contains certain extended ANSI characters, and the 8-bit console functions are set to use the OEM code page, then the **WriteConsoleA** function does not display the string properly.

Use the **AreFileApisANSI** function to determine which code page the set of file I/O functions is currently using. Use the **SetConsoleCP** and **SetConsoleOutputCP** functions to set the code page for the 8-bit console functions.

To solve the problem of code page incompatibility, it is best to use Unicode for console applications. Console applications that use Unicode are much more versatile than those that use 8-bit console functions. Barring that solution, a console application can call the **SetFileApisToOEM** function to cause the set of file I/O functions to use OEM character set strings rather than ANSI character set strings. Use the **SetFileApisToANSI** function to set those functions back to the ANSI code page.

When dealing with command lines, a console application should obtain the command line in Unicode form and then convert it to OEM form using the relevant character-to-OEM functions. Note also that the array in the *argv* parameter of the command-line **main** function contains ANSI character set strings in this case.

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.

See Also

File I/O Overview, *File I/O Functions*, **AreFileApisANSI**, **FindFirstFileA**, **SetFileApisToOEM**, **SetConsoleCP**, **SetConsoleOutputCP**, **WriteConsoleA**

SetFileApisToOEM

The **SetFileApisToOEM** function causes a set of file I/O functions to use the OEM character set code page. This function is useful for 8-bit console input and output operations.

```
VOID SetFileApisToOEM(VOID);
```

Parameters

This function has no parameters.

Return Values

This function has no return value.

Remarks

The file I/O functions whose code page is set by **SetFileApisToOEM** are those functions exported by KERNEL32.DLL that accept or return a file name. **SetFileApisToOEM** sets the code page per process, rather than per thread or per computer.

The **SetFileApisToOEM** function is complemented by the **SetFileApisToANSI** function, which causes the same set of file I/O functions to use the ANSI character set code page.

The 8-bit console functions use the OEM code page by default. All other functions use the ANSI code page by default. This means that strings returned by the console functions may not be processed correctly by other functions, and vice versa. For example, if the **FindFirstFileA** function returns a string that contains certain extended ANSI characters, and the 8-bit console functions are set to use the OEM code page, then the **WriteConsoleA** function will not display the string properly.

Use the **AreFileApisANSI** function to determine which code page the set of file I/O functions is currently using. Use the **SetConsoleCP** and **SetConsoleOutputCP** functions to set the code page for the 8-bit console functions.

To solve the problem of code page incompatibility, it is best to use Unicode for console applications. Console applications that use Unicode are much more versatile than those that use 8-bit console functions. Barring that solution, a console application can call the **SetFileApisToOEM** function to cause the set of file I/O functions to use OEM character set strings rather than ANSI character set strings. Use the **SetFileApisToANSI** function to set those functions back to the ANSI code page.

When dealing with command lines, a console application should obtain the command line in Unicode form and then convert it to OEM form using the relevant character-to-OEM functions. Note also that the array in the *argv* parameter of the command-line **main** function contains ANSI character set strings in this case.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **AreFileApisANSI**, **FindFirstFileA**, **SetConsoleCP**, **SetConsoleCP**, **SetConsoleOutputCP**, **SetFileApisToANSI**, **WriteConsoleA**

SetFileAttributes

The **SetFileAttributes** function sets a file's attributes.

```
BOOL SetFileAttributes(
  LPCTSTR lpFileName,        // file name
  DWORD dwFileAttributes     // attributes
);
```

Parameters

lpFileName

[in] Pointer to a string that specifies the name of the file whose attributes are to be set.

Windows NT/2000: There is a default string size limit for paths of MAX_PATH characters. This limit is related to how the **SetFileAttributes** function parses paths. An application can transcend this limit and send in paths longer than MAX_PATH characters by calling the wide (W) version of **SetFileAttributes** and prepending \\?\" to the path. However, each component in the path cannot be more than MAX_PATH characters long. The "\\?\" tells the function to turn off path parsing; it lets paths longer than MAX_PATH be used with **SetFileAttributesW**. However, each component in the path cannot be more than MAX_PATH characters long. This also works with UNC names. The "\\?\" is ignored as part of the path. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private", and "\\?\UNC\wow\hotstuff\coolapps" is seen as "\\wow\hotstuff\coolapps".

Windows 95: This string must not exceed MAX_PATH characters.

dwFileAttributes

[in] Specifies the file attributes to set for the file. This parameter can be one or more of the following values. However, all other values override FILE_ATTRIBUTE_NORMAL.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file is an archive file. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_HIDDEN	The file is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_NOT_CONTENT_INDEXED	The file will not be indexed by the content indexing service.
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file is read-only. Applications can read the file but cannot write to it or delete it.
FILE_ATTRIBUTE_SYSTEM	The file is part of the operating system or is used exclusively by it.
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

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 following table describes how to set the attributes that cannot be set using **SetFileAttributes**.

Attribute	How to set
FILE_ATTRIBUTE_COMPRESSED	To set a file's compression state, use the **DeviceIoControl** function with the **FSCTL_SET_COMPRESSION** operation.

(continued)

(continued)

Attribute	How to set
FILE_ATTRIBUTE_DEVICE	Reserved; do not use.
FILE_ATTRIBUTE_DIRECTORY	Files cannot be converted into directories. To create a directory, use the **CreateDirectory** or **CreateDirectoryEx** function.
FILE_ATTRIBUTE_ENCRYPTED	To create an encrypted file, use the **CreateFile** function with the FILE_ATTRIBUTE_ENCRYPTED attribute. To convert an existing file into an encrypted file, use the **EncryptFile** function.
FILE_ATTRIBUTE_REPARSE_POINT	To associate a reparse point with a file, use the **DeviceIoControl** function with the **FSCTL_SET_REPARSE_POINT** operation.
FILE_ATTRIBUTE_SPARSE_FILE	To set a file's sparse attribute, use the **DeviceIoControl** function with the **FSCTL_SET_SPARSE** operation.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File I/O Overview, *File I/O Functions*, **GetFileAttributes**

SetFilePointer

The **SetFilePointer** function moves the file pointer of an open file.

This function stores the file pointer in two **DWORD** values. To more easily work with file pointers that are larger than a single **DWORD** value, use the **SetFilePointerEx** function.

```
DWORD SetFilePointer(
  HANDLE hFile,                // handle to file
  LONG lDistanceToMove,        // bytes to move pointer
  PLONG lpDistanceToMoveHigh,  // bytes to move pointer
  DWORD dwMoveMethod           // starting point
);
```

Parameters

hFile

[in] Handle to the file whose file pointer is to be moved. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file.

lDistanceToMove

[in] Low-order 32 bits of a signed value that specifies the number of bytes to move the file pointer. If *lpDistanceToMoveHigh* is not NULL, *lpDistanceToMoveHigh* and *lDistanceToMove* form a single 64-bit signed value that specifies the distance to move. If *lpDistanceToMoveHigh* is NULL, *lDistanceToMove* is a 32-bit signed value. A positive value for *lDistanceToMove* moves the file pointer forward in the file, and a negative value moves the file pointer backward.

lpDistanceToMoveHigh

[in] Pointer to the high-order 32 bits of the signed 64-bit distance to move. If you do not need the high-order 32 bits, this pointer may be NULL. When non-NULL, this parameter also receives the high-order **DWORD** of the new value of the file pointer. For more information, see the Remarks section later in this topic.

Windows 95/98: If the pointer *lpDistanceToMoveHigh* is not NULL, then it must point to either 0 or –1, the sign extension of the value of *lDistanceToMove*. Any other value will be rejected.

dwMoveMethod

[in] Starting point for the file pointer move. This parameter can be one of the following values.

Value	Meaning
FILE_BEGIN	The starting point is zero or the beginning of the file.
FILE_CURRENT	The starting point is the current value of the file pointer.
FILE_END	The starting point is the current end-of-file position.

Return Values

If the **SetFilePointer** function succeeds and *lpDistanceToMoveHigh* is NULL, the return value is the low-order **DWORD** of the new file pointer. If *lpDistanceToMoveHigh* is not NULL, the function returns the low order **DWORD** of the new file pointer, and puts the high-order **DWORD** of the new file pointer into the **LONG** pointed to by that parameter.

If the function fails and *lpDistanceToMoveHigh* is NULL, the return value is INVALID_SET_FILE_POINTER. To get extended error information, call **GetLastError**.

If the function fails, and *lpDistanceToMoveHigh* is non-NULL, the return value is INVALID_SET_FILE_POINTER. However, because INVALID_SET_FILE_POINTER is a valid value for the low-order **DWORD** of the new file pointer, you must check **GetLastError** to determine whether an error occurred. If an error occurred, **GetLastError** returns a value other than NO_ERROR. For a code example that illustrates this point, see the Remarks section later in this topic.

If the new file pointer would have been a negative value, the function fails, the file pointer is not moved, and the code returned by **GetLastError** is ERROR_NEGATIVE_SEEK.

Remarks

You cannot use the **SetFilePointer** function with a handle to a nonseeking device such as a pipe or a communications device. To determine the file type for *hFile*, use the **GetFileType** function.

To determine the present position of a file pointer, see *Retrieving a File Pointer.*

Use caution when setting the file pointer in a multithreaded application. You must synchronize access to shared resources. For example, an application whose threads share a file handle, update the file pointer, and read from the file must protect this sequence by using a critical section object or mutex object. For more information about these objects, see *Critical Section Objects* and *Mutex Objects*.

If the *hFile* file handle was opened with the FILE_FLAG_NO_BUFFERING flag set, an application can move the file pointer only to sector-aligned positions. A *sector-aligned position* is a position that is a whole number multiple of the volume's sector size. An application can obtain a volume's sector size by calling the **GetDiskFreeSpace** function. If an application calls **SetFilePointer** with distance-to-move values that result in a position that is not sector-aligned and a handle that was opened with FILE_FLAG_NO_BUFFERING, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

Note that it is not an error to set the file pointer to a position beyond the end of the file. The size of the file does not increase until you call the **SetEndOfFile**, **WriteFile**, or **WriteFileEx** function. A write operation increases the size of the file to the file pointer position plus the size of the buffer written, leaving the intervening bytes uninitialized.

If the return value is INVALID_SET_FILE_POINTER and if *lpDistanceToMoveHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded or failed.

The parameter *lpDistanceToMoveHigh* is used to manipulate huge files. If it is set to NULL, then *lDistanceToMove* has a maximum value of 2^31–2, or 2 gigabytes less two. This is because all file pointer values are signed values. Therefore if there is even a small chance that the file will grow to that size, you should treat the file as a huge file and work with 64-bit file pointers. With file compression on NTFS, and sparse files, it is possible to have files that are large even if the underlying volume is not very large.

If *lpDistanceToMoveHigh* is not NULL, then *lpDistanceToMoveHigh* and *lDistanceToMove* form a single 64-bit signed value. The *lDistanceToMove* parameter is treated as the low-order 32 bits of the value, and *lpDistanceToMoveHigh* as the upper 32 bits. Thus, *lpDistanceToMoveHigh* is a sign extension of *lDistanceToMove.*

To move the file pointer from zero to 2 gigabytes, *lpDistanceToMoveHigh* can be either NULL or a sign extension of *lDistanceToMove.* To move the pointer more than 2 gigabytes, use *lpDistanceToMoveHigh* and *lDistanceToMove* as a single 64-bit quantity. For example, to move in the range from 2 gigabytes to 4 gigabytes set the contents of *lpDistanceToMoveHigh* to zero, or to –1 for a negative sign extension of *lDistanceToMove.*

To work with 64-bit file pointers, you can declare a **LONG**, treat it as the upper half of the 64-bit file pointer, and pass its address in *lpDistanceToMoveHigh*. This means you have to treat two different variables as a logical unit, which is error-prone. The problems can be ameliorated by using the **LARGE_INTEGER** structure to create a 64-bit value and passing the two 32-bit values by means of the appropriate elements of the union.

It is conceptually simpler and better design to use a function to hide the interface to **SetFilePointer**.

Note You can use **SetFilePointer** to determine the length of a file. To do this, use FILE_END for *dwMoveMethod* and seek to location zero. The file offset returned is the length of the file. However, this practice can have unintended side effects, such as failure to save the current file pointer so that the program can return to that location. It is simpler and safer to use **GetFileSize** instead.

You can also use the **SetFilePointer** function to query the current file pointer position. To do this, specify a move method of FILE_CURRENT and a distance of zero.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **GetDiskFreeSpace**, **GetFileSize**, **GetFileType**, **ReadFile**, **ReadFileEx**, **ReadFileVlm**, **SetEndOfFile**, **SetFilePointerEx**, **WriteFile**, **WriteFileEx**, **WriteFileVlm**

SetFilePointerEx

The **SetFilePointerEx** function moves the file pointer of an open file.

```
BOOL SetFilePointerEx(
  HANDLE hFile,                        // handle to file
  LARGE_INTEGER liDistanceToMove,      // bytes to move pointer
  PLARGE_INTEGER lpNewFilePointer,     // new file pointer
  DWORD dwMoveMethod                   // starting point
);
```

Parameters

hFile

[in] Handle to the file whose file pointer is to be moved. The file handle must have been created with GENERIC_READ or GENERIC_WRITE access to the file.

liDistanceToMove

[in] Specifies the number of bytes to move the file pointer. A positive value moves the pointer forward in the file and a negative value moves the file pointer backward.

lpNewFilePointer

[in] Pointer to a variable that receives the new file pointer. If this parameter is NULL, the new file pointer is not returned.

dwMoveMethod

[in] Specifies the starting point for the file pointer move. This parameter can be one of the following values.

Value	Meaning
FILE_BEGIN	The starting point is zero or the beginning of the file. If this flag is specified, then the *liDistanceToMove* parameter is interpreted as an unsigned value.
FILE_CURRENT	The start point is the current value of the file pointer.
FILE_END	The starting point is the current end-of-file 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

You cannot use the **SetFilePointerEx** function with a handle to a nonseeking device such as a pipe or a communications device. To determine the file type for *hFile*, use the **GetFileType** function.

Use caution when setting the file pointer in a multithreaded application. You must synchronize access to shared resources. For example, an application whose threads share a file handle, update the file pointer, and read from the file must protect this sequence by using a critical section object or a mutex object. For more information about these objects, see *Critical Section Objects* and *Mutex Objects*.

If the *hFile* handle was opened with the FILE_FLAG_NO_BUFFERING flag set, an application can move the file pointer only to sector-aligned positions. A *sector-aligned position* is a position that is a whole number multiple of the volume's sector size. An application can obtain a volume's sector size by calling the **GetDiskFreeSpaceEx** function. If an application calls **SetFilePointerEx** with distance-to-move values that result in a position that is not sector-aligned and a handle that was opened with FILE_FLAG_NO_BUFFERING, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

Note that it is not an error to set the file pointer to a position beyond the end of the file. The size of the file does not increase until you call the **SetEndOfFile**, **WriteFile**, or **WriteFileEx** function. A write operation increases the size of the file to the file pointer position plus the size of the buffer written, leaving the intervening bytes uninitialized.

You can use **SetFilePointerEx** to determine the length of a file. To do this, use FILE_END for *dwMoveMethod* and seek to location zero. The file offset returned is the length of the file. However, this practice can have unintended side effects, such as failure to save the current file pointer so that the program can return to that location. It is simpler and safer to use the **GetFileSizeEx** function instead.

You can also use **SetFilePointerEx** to query the current file pointer position. To do this, specify a move method of FILE_CURRENT and a distance of zero.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **GetDiskFreeSpaceEx**, **GetFileSizeEx**, **GetFileType**, **SetEndOfFile**, **WriteFile**, **WriteFileEx**

SetVolumeLabel

The **SetVolumeLabel** function sets the label of a file system volume.

```
BOOL SetVolumeLabel(
  LPCTSTR lpRootPathName,  // name of volume root directory
  LPCTSTR lpVolumeName     // name for the volume
);
```

Parameters

lpRootPathName

[in] Pointer to a null-terminated string specifying the root directory of a file system volume. This is the volume the function will label. A trailing backslash is required. If this parameter is NULL, the root of the current directory is used.

lpVolumeName

[in] Pointer to a string specifying a name for the volume. If this parameter is NULL, the function deletes the label from the specified volume.

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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File I/O Overview, *File I/O Functions*, **GetVolumeInformation**

UInt32x32To64

The **UInt32x32To64** function multiplies two unsigned 32-bit integers, returning an unsigned 64-bit integer result. The function performs optimally on all Win32 platforms.

```
ULONGLONG UInt32x32To64(
  DWORD Multiplier,     // first unsigned 32-bit integer
  DWORD Multiplicand    // second unsigned 32-bit integer
);
```

Parameters

Multiplier
[in] Specifies the first unsigned 32-bit integer for the multiplication.

Multiplicand
[in] Specifies the second unsigned 32-bit integer for the multiplication.

Return Values

The return value is the unsigned 64-bit integer result of the multiplication.

Remarks

This function is implemented on all platforms by optimal inline code: a single multiply instruction that returns a 64-bit result.

Please note that the function's return value is a 64-bit value, not a **LARGE_INTEGER** structure.

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

File I/O Overview, *File I/O Functions*, **Int32x32To64**

UnlockFile

The **UnlockFile** function unlocks a region in an open file. Unlocking a region enables other processes to access the region.

For an alternate way to specify the region, use the **UnlockFileEx** function.

```
BOOL UnlockFile(
  HANDLE hFile,                    // handle to file
  DWORD dwFileOffsetLow,           // low-order word of start
  DWORD dwFileOffsetHigh,          // high-order word of start
  DWORD nNumberOfBytesToUnlockLow, // low-order word of length
  DWORD nNumberOfBytesToUnlockHigh // high-order word of length
);
```

Parameters

hFile
[in] Handle to a file that contains a region locked with **LockFile**. The file handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

dwFileOffsetLow
[in] Specifies the low-order word of the starting byte offset in the file where the locked region begins.

dwFileOffsetHigh
[in] Specifies the high-order word of the starting byte offset in the file where the locked region begins.

Windows 95/98: *dwFileOffsetHigh* must be 0, the sign extension of the value of *dwFileOffsetLow*. Any other value will be rejected.

nNumberOfBytesToUnlockLow
[in] Specifies the low-order word of the length of the byte range to be unlocked.

nNumberOfBytesToUnlockHigh
[in] Specifies the high-order word of the length of the byte range to be unlocked.

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

Unlocking a region of a file releases a lock on the file. The region to unlock must correspond exactly to an existing locked region. For example, two adjacent regions of a file cannot be locked separately and then unlocked as a single region that spans both locked regions.

A process should not be terminated with a portion of a file locked and a file that has locked regions should not be closed.

This function works on a file allocation table (FAT)—based file system only if the operating system is running SHARE.EXE.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **LockFile**, **UnlockFileEx**

UnlockFileEx

The **UnlockFileEx** function unlocks a previously locked byte range in an open file.

```
BOOL UnlockFileEx(
  HANDLE hFile,                       // handle to file
  DWORD dwReserved,                   // reserved
  DWORD nNumberOfBytesToUnlockLow,    // low-order part of length
  DWORD nNumberOfBytesToUnlockHigh,   // high-order part of length
  LPOVERLAPPED lpOverlapped           // unlock region start
);
```

Parameters

hFile

[in] Handle to a file that is to have an existing locked region unlocked. The handle must have been created with either GENERIC_READ or GENERIC_WRITE access to the file.

dwReserved

Reserved parameter; must be zero.

nNumberOfBytesToUnlockLow

[in] Specifies the low-order part of the length of the byte range to unlock.

nNumberOfBytesToUnlockHigh

[in] Specifies the high-order part of the length of the byte range to unlock.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure that the function uses with the unlocking request. This structure contains the file offset of the beginning of the unlock range.

Return Values

If the function succeeds, the return value is nonzero.

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

Remarks

Unlocking a region of a file releases a previously acquired lock on a file. The region to unlock must correspond exactly to an existing locked region. Two adjacent regions of a file cannot be locked separately and then unlocked using a single region that spans both locked regions.

If a process terminates with a portion of a file locked or closes a file that has outstanding locks, the behavior is not specified.

! Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

+ See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **LockFile**, **LockFileEx**, **OVERLAPPED**, **UnlockFile**

WriteFile

The **WriteFile** function writes data to a file and is designed for both synchronous and asynchronous operation. The function starts writing data to the file at the position indicated by the file pointer. After the write operation has been completed, the file pointer is adjusted by the number of bytes actually written, except when the file is opened with FILE_FLAG_OVERLAPPED. If the file handle was created for overlapped input and output (I/O), the application must adjust the position of the file pointer after the write operation is finished.

This function is designed for both synchronous and asynchronous operation. The **WriteFileEx** function is designed solely for asynchronous operation. It lets an application perform other processing during a file write operation.

```
BOOL WriteFile(
  HANDLE hFile,                    // handle to file
  LPCVOID lpBuffer,                // data buffer
  DWORD nNumberOfBytesToWrite,     // number of bytes to write
  LPDWORD lpNumberOfBytesWritten,  // number of bytes written
  LPOVERLAPPED lpOverlapped        // overlapped buffer
);
```

Parameters

hFile

[in] Handle to the file to be written to. The file handle must have been created with GENERIC_WRITE access to the file.

Windows NT/2000: For asynchronous write operations, *hFile* can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the **CreateFile** function, or a socket handle returned by the **socket** or **accept** function.

Windows 95/98: For asynchronous write operations, *hFile* can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous write operations on mailslots, named pipes, or disk files.

lpBuffer

[in] Pointer to the buffer containing the data to be written to the file.

nNumberOfBytesToWrite

[in] Specifies the number of bytes to write to the file.

A value of zero specifies a null write operation. A null write operation does not write any bytes but does cause the time stamp to change.

Named pipe write operations across a network are limited to 65,535 bytes.

lpNumberOfBytesWritten

[out] Pointer to the variable that receives the number of bytes written. **WriteFile** sets this value to zero before doing any work or error checking.

Windows NT/2000: If *lpOverlapped* is NULL, *lpNumberOfBytesWritten* cannot be NULL. If *lpOverlapped* is not NULL, *lpNumberOfBytesWritten* can be NULL. If this is an overlapped write operation, you can get the number of bytes written by calling **GetOverlappedResult**. If *hFile* is associated with an I/O completion port, you can get the number of bytes written by calling **GetQueuedCompletionStatus**.

Windows 95/98: This parameter cannot be NULL.

lpOverlapped

[in] Pointer to an **OVERLAPPED** structure. This structure is required if *hFile* was opened with FILE_FLAG_OVERLAPPED.

If *hFile* was opened with FILE_FLAG_OVERLAPPED, the *lpOverlapped* parameter must not be NULL. It must point to a valid **OVERLAPPED** structure. If *hFile* was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is NULL, the function can incorrectly report that the write operation is complete.

If *hFile* was opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, the write operation starts at the offset specified in the **OVERLAPPED** structure and **WriteFile** may return before the write operation has been completed. In this case, **WriteFile** returns FALSE and the **GetLastError** function returns ERROR_IO_PENDING. This allows the calling process to continue processing while the write operation is being completed. The event specified in the **OVERLAPPED** structure is set to the signaled state upon completion of the write operation.

If *hFile* was not opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is NULL, the write operation starts at the current file position and **WriteFile** does not return until the operation has been completed.

Windows NT/2000: If *hFile* was not opened with FILE_FLAG_OVERLAPPED and *lpOverlapped* is not NULL, the write operation starts at the offset specified in the **OVERLAPPED** structure and **WriteFile** does not return until the write operation has been completed.

Windows 95/98: For operations on files, disks, pipes, or mailslots, this parameter must be NULL; a pointer to an **OVERLAPPED** structure causes the call to fail. However, Windows 95/98 supports overlapped I/O on serial and parallel ports.

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 must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

- File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. To determine a volume's sector size, call the **GetDiskFreeSpace** function.

- File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

If part of the file is locked by another process and the write operation overlaps the locked portion, this function fails.

Accessing the output buffer while a write operation is using the buffer may lead to corruption of the data written from that buffer. Applications must not read from, write to, reallocate, or free the output buffer that a write operation is using until the write operation completes.

Characters can be written to the screen buffer using **WriteFile** with a handle to console output. The exact behavior of the function is determined by the console mode. The data is written to the current cursor position. The cursor position is updated after the write operation.

The system interprets zero bytes to write as specifying a null write operation and **WriteFile** does not truncate or extend the file. To truncate or extend a file, use the **SetEndOfFile** function.

When writing to a nonblocking, byte-mode pipe handle with insufficient buffer space, **WriteFile** returns TRUE with **lpNumberOfBytesWritten* < *nNumberOfBytesToWrite*.

When an application uses the **WriteFile** function to write to a pipe, the write operation may not finish if the pipe buffer is full. The write operation is completed when a read operation (using the **ReadFile** function) makes more buffer space available.

If the anonymous read pipe handle has been closed and **WriteFile** attempts to write using the corresponding anonymous write pipe handle, the function returns FALSE and **GetLastError** returns ERROR_BROKEN_PIPE.

The **WriteFile** function may fail with ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY whenever there are too many outstanding asynchronous I/O requests.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle. I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to write to a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the

system from displaying this message box, call the **SetErrorMode** function with SEM_NOOPENFILEERRORBOX.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development.*

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File I/O Overview, *File I/O Functions*, **CancelIo**, **CreateFile**, **GetLastError**, **GetOverlappedResult**, **GetQueuedCompletionStatus**, **OVERLAPPED**, **ReadFile**, **SetEndOfFile**, **SetErrorMode**, **WriteFileEx**

WriteFileEx

The **WriteFileEx** function writes data to a file. It is designed solely for asynchronous operation, unlike **WriteFile**, which is designed for both synchronous and asynchronous operation.

WriteFileEx reports its completion status asynchronously, calling a specified completion routine when writing is completed or canceled and the calling thread is in an alertable wait state.

```
BOOL WriteFileEx(
  HANDLE hFile,                                      // handle to output file
  LPCVOID lpBuffer,                                  // data buffer
  DWORD nNumberOfBytesToWrite,                       // number of bytes to write
  LPOVERLAPPED lpOverlapped,                         // overlapped buffer
  LPOVERLAPPED_COMPLETION_ROUTINE lpCompletionRoutine // completion routine
);
```

Parameters

hFile

[in] Handle to the file to be written to. This file handle must have been created with the FILE_FLAG_OVERLAPPED flag and with GENERIC_WRITE access to the file.

Windows NT/2000: This parameter can be any handle opened with the FILE_FLAG_OVERLAPPED flag by the **CreateFile** function, or a socket handle returned by the **socket** or **accept** function.

Windows 95/98: This parameter can be a communications resource opened with the FILE_FLAG_OVERLAPPED flag by **CreateFile**, or a socket handle returned by **socket** or **accept**. You cannot perform asynchronous write operations on mailslots, named pipes, or disk files.

lpBuffer

[in] Pointer to the buffer containing the data to be written to the file.

This buffer must remain valid for the duration of the write operation. The caller must not use this buffer until the write operation is completed.

nNumberOfBytesToWrite

[in] Specifies the number of bytes to write to the file.

If *nNumberOfBtyesToWrite* is zero, this function does nothing; in particular, it does not truncate the file. For additional discussion, see the following Remarks section.

lpOverlapped

[in] Pointer to an **OVERLAPPED** data structure that supplies data to be used during the overlapped (asynchronous) write operation.

For files that support byte offsets, you must specify a byte offset at which to start writing to the file. You specify this offset by setting the **Offset** and **OffsetHigh** members of the **OVERLAPPED** structure. For files that do not support byte offsets, **Offset** and **OffsetHigh** are ignored.

The **WriteFileEx** function ignores the **OVERLAPPED** structure's **hEvent** member. An application is free to use that member for its own purposes in the context of a **WriteFileEx** call. **WriteFileEx** signals completion of its writing operation by calling, or queuing a call to, the completion routine pointed to by *lpCompletionRoutine*, so it does not need an event handle.

The **WriteFileEx** function does use the **Internal** and **InternalHigh** members of the **OVERLAPPED** structure. You should not change the value of these members.

The **OVERLAPPED** data structure must remain valid for the duration of the write operation. It should not be a variable that can go out of scope while the write operation is pending completion.

lpCompletionRoutine

[in] Pointer to a completion routine to be called when the write operation has been completed and the calling thread is in an alertable wait state. For more information about this completion routine, see **FileIOCompletionRoutine**.

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**.

If the **WriteFileEx** function succeeds, the calling thread has an asynchronous I/O (input/output) operation pending: the overlapped write operation to the file. When this I/O operation finishes, and the calling thread is blocked in an alertable wait state, the

operating system calls the function pointed to by *lpCompletionRoutine*, and the wait completes with a return code of WAIT_IO_COMPLETION.

If the function succeeds and the file-writing operation finishes, but the calling thread is not in an alertable wait state, the system queues the call to **lpCompletionRoutine,* holding the call until the calling thread enters an alertable wait state. See *Synchronization* for more information about alertable wait states and overlapped input/output operations.

Remarks

When using **WriteFileEx** you should check **GetLastError** even when the function returns "success" to check for conditions that are "successes" but have some outcome you might want to know about. For example, a buffer overflow when calling **WriteFileEx** will return TRUE, but **GetLastError** will report the overflow with ERROR_MORE_DATA. If the function call is successful and there are no warning conditions, **GetLastError** will return ERROR_SUCCESS.

An application must meet certain requirements when working with files opened with FILE_FLAG_NO_BUFFERING:

- File access must begin at byte offsets within the file that are integer multiples of the volume's sector size. To determine a volume's sector size, call the **GetDiskFreeSpace** function.
- File access must be for numbers of bytes that are integer multiples of the volume's sector size. For example, if the sector size is 512 bytes, an application can request reads and writes of 512, 1024, or 2048 bytes, but not of 335, 981, or 7171 bytes.
- Buffer addresses for read and write operations must be sector aligned (aligned on addresses in memory that are integer multiples of the volume's sector size). One way to sector align buffers is to use the **VirtualAlloc** function to allocate the buffers. This function allocates memory that is aligned on addresses that are integer multiples of the system's page size. Because both page and volume sector sizes are powers of 2, memory aligned by multiples of the system's page size is also aligned by multiples of the volume's sector size.

If part of the output file is locked by another process, and the specified write operation overlaps the locked portion, the **WriteFileEx** function fails.

Accessing the output buffer while a write operation is using the buffer may lead to corruption of the data written from that buffer. Applications must not read from, write to, reallocate, or free the output buffer that a write operation is using until the write operation completes.

The **WriteFileEx** function may fail, returning the messages ERROR_INVALID_USER_BUFFER or ERROR_NOT_ENOUGH_MEMORY if there are too many outstanding asynchronous I/O requests.

To cancel all pending asynchronous I/O operations, use the **CancelIo** function. This function only cancels operations issued by the calling thread for the specified file handle.

I/O operations that are canceled complete with the error ERROR_OPERATION_ABORTED.

If you are attempting to write to a floppy drive that does not have a floppy disk, the system displays a message box prompting the user to retry the operation. To prevent the system from displaying this message box, call the **SetErrorMode** function with SEM_NOOPENFILEERRORBOX.

An application uses the **WaitForSingleObjectEx**, **WaitForMultipleObjectsEx**, **MsgWaitForMultipleObjectsEx**, **SignalObjectAndWait**, and **SleepEx** functions to enter an alertable wait state. Refer to *Synchronization* for more information about alertable wait states and overlapped input/output operations.

Windows 95/98: On this platform, neither **WriteFileEx** nor **ReadFileEx** can be used by the comm ports to communicate. However, you can use **WriteFile** and **ReadFile** to perform asynchronous communication.

Requirements

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.

See Also

File I/O Overview, *File I/O Functions*, **CancelIo**, **CreateFile**, **FileIOCompletionRoutine**, **MsgWaitForMultipleObjectsEx**, **OVERLAPPED**, **ReadFileEx**, **SetEndOfFile**, **SetErrorMode**, **SleepEx**, **SignalObjectAndWait**, **WaitForMultipleObjectsEx**, **WaitForSingleObjectEx**, **WriteFile**, **WriteFileVlm**

WriteFileGather

The **WriteFileGather** function gathers data from a set of buffers and writes the data to a file.

The **WriteFileGather** function starts writing data to the file at a position specified by an **OVERLAPPED** structure.

The **WriteFileGather** function operates asynchronously.

```
BOOL WriteFileGather(
  HANDLE hFile,                              // handle to file
  FILE_SEGMENT_ELEMENT aSegmentArray[ ],     // array of buffer pointers
  DWORD nNumberOfBytesToWrite,               // number of bytes to write
  LPDWORD lpReserved,                        // reserved; must be NULL
  LPOVERLAPPED lpOverlapped                  // overlapped buffer
```

```
);
```

Parameters

hFile

[in] Handle to the file to write to.

This file handle must have been created using GENERIC_WRITE to specify write access to the file, FILE_FLAG_OVERLAPPED to specify asynchronous I/O, and FILE_FLAG_NO_BUFFERING to specify non-cached I/O.

aSegmentArray

[out] Pointer to an array of **FILE_SEGMENT_ELEMENT** pointers to buffers. The function gathers the data it writes to the file from this set of buffers.

Each buffer should be the size of a system memory page. Each buffer should be aligned on a system memory page size boundary.

A **FILE_SEGMENT_ELEMENT** pointer is a 64-bit value. The **WriteFileGather** function uses all 64 bits. Because the operating systems do not currently support 64-bit memory addressing, you must explicitly set the upper 32 bits of each **FILE_SEGMENT_ELEMENT** pointer. This is best done with a cast to **__ptr64**. For compilers where **__ptr64** is not available, setting the upper 32 bits of the pointer to zero is a less elegant workaround.

The function gathers the data from the buffers in a sequential manner: it writes data to the file from the first buffer, then from the second buffer, then from the next, until there is no more data to write.

The final element of the array should be a 64-bit NULL pointer.

nNumberOfBytesToWrite

[in] Specifies the number of bytes to write to the file.

If *nNumberOfBytesToWrite* is zero, the function performs a null write operation. A null write operation does not write any bytes to the file, but it does cause the file's time stamp to change.

Note that this behavior differs from file writing functions on the MS-DOS platform, where a write count of zero bytes truncates a file. **WriteFileGather** does not truncate or extend the file. To truncate or extend a file, use the **SetEndOfFile** function.

lpReserved

This parameter is reserved for future use. You must set it to NULL.

lpOverlapped

[in] Pointer to an **OVERLAPPED** data structure.

The **WriteFileGather** function requires a valid **OVERLAPPED** structure. The *lpOverlapped* parameter cannot be NULL.

The **WriteFileGather** function starts writing data to the file at a position specified by the **Offset** and **OffsetHigh** members of the **OVERLAPPED** structure.

The **WriteFileGather** function may return before the write operation has completed. In that case, the **WriteFileGather** function returns the value zero, and the **GetLastError** function returns the value ERROR_IO_PENDING. This asynchronous operation of

WriteFileGather lets the calling process continue while the write operation completes. You can call the **GetOverlappedResult**, **HasOverlappedIoCompleted**, or **GetQueuedCompletionStatus** function to obtain information about the completion of the write operation.

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 the **GetLastError** function.

If the function returns before the write operation has completed, the function returns zero, and **GetLastError** returns ERROR_IO_PENDING.

Remarks

If part of the file specified by *hFile* is locked by another process, and the write operation overlaps the locked portion, the **WriteFileGather** function fails.

Requirements

Windows NT/2000: Requires Windows NT 4.0 SP2 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File I/O Overview, *File I/O Functions*, **CreateFile**, **GetOverlappedResult**, **GetQueuedCompletionStatus**, **HasOverlappedIoCompleted**, **OVERLAPPED**, **ReadFile**, **ReadFileEx**, **ReadFileScatter**, **ReadFileVlm**

File I/O Structures

BY_HANDLE_FILE_INFORMATION

The **BY_HANDLE_FILE_INFORMATION** structure contains information retrieved by the **GetFileInformationByHandle** function.

```
typedef struct _BY_HANDLE_FILE_INFORMATION {
    DWORD    dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
    FILETIME ftLastWriteTime;
    DWORD    dwVolumeSerialNumber;
```

```
    DWORD    nFileSizeHigh;
    DWORD    nFileSizeLow;
    DWORD    nNumberOfLinks;
    DWORD    nFileIndexHigh;
    DWORD    nFileIndexLow;
} BY_HANDLE_FILE_INFORMATION, *PBY_HANDLE_FILE_INFORMATION;
```

Members

dwFileAttributes

Specifies file attributes. This member can be one or more of the following values.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file or directory is an archive file. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DIRECTORY	The handle identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file has no other attributes. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE	The file data is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.
FILE_ATTRIBUTE_REPARSE_POINT	The file has an associated reparse point.
FILE_ATTRIBUTE_SPARSE_FILE	The file is a sparse file.
FILE_ATTRIBUTE_SYSTEM	The file or directory is part of the operating system or is used exclusively by the operating system.

(continued)

(continued)

Attribute	Meaning
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all the data in memory for quicker access, rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

ftCreationTime
Specifies the time the file was created. If the underlying file system does not support this time member, **ftCreationTime** is zero.

ftLastAccessTime
Specifies the time the file was last accessed. If the underlying file system does not support this time member, **ftLastAccessTime** is zero.

ftLastWriteTime
Specifies the last time the file was written to.

dwVolumeSerialNumber
Specifies the serial number of the volume that contains the file.

nFileSizeHigh
Specifies the high-order word of the file size.

nFileSizeLow
Specifies the low-order word of the file size.

nNumberOfLinks
Specifies the number of links to this file. For the FAT file system this member is always 1. For NTFS, it may be more than 1.

nFileIndexHigh
Specifies the high-order word of a unique identifier associated with the file.

nFileIndexLow
Specifies the low-order word of a unique identifier associated with the file. This identifier and the volume serial number uniquely identify a file. This number may change when the system is restarted or when the file is opened. After a process opens a file, the identifier is constant until the file is closed. An application can use this identifier and the volume serial number to determine whether two handles refer to the same file.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.

File I/O Overview, *File I/O Structures*, **GetFileInformationByHandle**

FILE_NOTIFY_INFORMATION

The **FILE_NOTIFY_INFORMATION** structure describes the changes found by the **ReadDirectoryChangesW** function.

```
typedef struct _FILE_NOTIFY_INFORMATION {
    DWORD NextEntryOffset;
    DWORD Action;
    DWORD FileNameLength;
    WCHAR FileName[1];
} FILE_NOTIFY_INFORMATION, *PFILE_NOTIFY_INFORMATION;
```

Members

NextEntryOffset

Specifies the number of bytes that must be skipped to get to the next record. A value of zero indicates that this is the last record.

Action

Specifies the type of change that occurred.

Value	Meaning
FILE_ACTION_ADDED	The file was added to the directory.
FILE_ACTION_REMOVED	The file was removed from the directory.
FILE_ACTION_MODIFIED	The file was modified. This can be a change in the time stamp or attributes.
FILE_ACTION_RENAMED_OLD_NAME	The file was renamed and this is the old name.
FILE_ACTION_RENAMED_NEW_NAME	The file was renamed and this is the new name.

FileNameLength

Specifies the length, in bytes, of the file name portion of the record. Note that this length does not include the terminating null character.

FileName

This is a variable-length field that contains the file name relative to the directory handle. The file name is in the Unicode character format and is not null-terminated.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

File I/O Overview, *File I/O Structures*, **ReadDirectoryChangesW**

LARGE_INTEGER

The **LARGE_INTEGER** structure is used to represent a 64-bit signed integer value.

Note Your C compiler may support 64-bit integers natively. For example, Microsoft Visual C++ supports the **__int64** sized integer type. For more information, see the documentation included with your C compiler.

```
typedef union _LARGE_INTEGER {
    struct {
        DWORD LowPart;
        LONG  HighPart;
    };
    LONGLONG QuadPart;
} LARGE_INTEGER, *PLARGE_INTEGER;
```

Members

LowPart
Specifies the low-order 32 bits.

HighPart
Specifies the high-order 32 bits.

QuadPart
Specifies a 64-bit signed integer.

Remarks

The **LARGE_INTEGER** structure is actually a union. If your compiler has built-in support for 64-bit integers, use the **QuadPart** member to store the 64-bit integer. Otherwise, use the **LowPart** and **HighPart** members to store the 64-bit integer.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

See Also

File I/O Overview, *File I/O Structures*, **LUID**, **ULARGE_INTEGER**

OFSTRUCT

The **OFSTRUCT** structure contains information about a file that the **OpenFile** function opened or attempted to open.

```
typedef struct _OFSTRUCT {
    BYTE cBytes;
    BYTE fFixedDisk;
    WORD nErrCode;
    WORD Reserved1;
    WORD Reserved2;
    CHAR szPathName[OFS_MAXPATHNAME];
} OFSTRUCT, *POFSTRUCT;
```

Members

cBytes
Specifies the length, in bytes, of the structure.

fFixedDisk
Specifies whether the file is on a hard (fixed) disk. This member is nonzero if the file is on a hard disk.

nErrCode
Specifies the MS-DOS error code if the **OpenFile** function failed.

Reserved1
Reserved; do not use.

Reserved2
Reserved; do not use.

szPathName
Specifies the path and file name of the file.

Requirements

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.

See Also

File I/O Overview, *File I/O Structures*, **OpenFile**

ULARGE_INTEGER

The **ULARGE_INTEGER** structure is used to specify a 64-bit unsigned integer value.

Note Your C compiler may support 64-bit integers natively. For example, Microsoft Visual C++ supports the **__int64** sized integer type. For more information, see the documentation included with your C compiler.

```
typedef union _ULARGE_INTEGER {
    struct {
        DWORD LowPart;
        DWORD HighPart;
    };
    ULONGLONG QuadPart;
} ULARGE_INTEGER, *PULARGE_INTEGER;
```

Members

LowPart
Specifies the low-order 32 bits.

HighPart
Specifies the high-order 32 bits.

QuadPart
Specifies a 64-bit unsigned integer.

Remarks

The **ULARGE_INTEGER** structure is actually a union. If your compiler has built-in support for 64-bit integers, use the **QuadPart** member to store the 64-bit integer. Otherwise, use the **LowPart** and **HighPart** members to store the 64-bit integer.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winnt.h; include windows.h.

See Also

File I/O Overview, *File I/O Structures*, **LARGE_INTEGER**

WIN32_FILE_ATTRIBUTE_DATA

The **WIN32_FILE_ATTRIBUTE_DATA** structure contains attribute information for a file or directory. The **GetFileAttributesEx** function uses this structure.

```
typedef struct _WIN32_FILE_ATTRIBUTE_DATA{
    DWORD      dwFileAttributes;
    FILETIME   ftCreationTime;
```

```
    FILETIME    ftLastAccessTime;
    FILETIME    ftLastWriteTime;
    DWORD       nFileSizeHigh;
    DWORD       nFileSizeLow;
} WIN32_FILE_ATTRIBUTE_DATA, *LPWIN32_FILE_ATTRIBUTE_DATA;
```

Members

dwFileAttributes

Specifies FAT-style attribute information for the file or directory.

The following attributes are defined.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file or directory is an archive file. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DIRECTORY	The handle identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file or directory has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE	The data of the file is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.
FILE_ATTRIBUTE_REPARSE_POINT	The file has an associated reparse point.
FILE_ATTRIBUTE_SPARSE_FILE	The file is a sparse file.
FILE_ATTRIBUTE_SYSTEM	The file or directory is part of the operating system or is used exclusively by the operating system.

(continued)

(continued)

Attribute	Meaning
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access rather than flushing the data back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

ftCreationTime
A **FILETIME** structure that specifies when the file or directory was created.

ftLastAccessTime
A **FILETIME** structure. For a file, the structure specifies when the file was last read from or written to. For a directory, the structure specifies when the directory was created. For both files and directories, the specified date will be correct, but the time of day will always be set to midnight.

ftLastWriteTime
A **FILETIME** structure. For a file, the structure specifies when the file was last written to. For a directory, the structure specifies when the directory was created.

nFileSizeHigh
Specifies the high-order **DWORD** of the file size. This member has no meaning for directories.

nFileSizeLow
Specifies the low-order **DWORD** of the file size. This member has no meaning for directories.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

See Also

File I/O Overview, *File I/O Structures*, **GetFileAttributesEx**, **GET_FILEEX_INFO_LEVELS**

WIN32_FIND_DATA

The **WIN32_FIND_DATA** structure describes a file found by the **FindFirstFile**, **FindFirstFileEx**, or **FindNextFile** function.

```
typedef struct _WIN32_FIND_DATA {
    DWORD dwFileAttributes;
    FILETIME ftCreationTime;
    FILETIME ftLastAccessTime;
```

```
    FILETIME ftLastWriteTime;
    DWORD    nFileSizeHigh;
    DWORD    nFileSizeLow;
    DWORD    dwReserved0;
    DWORD    dwReserved1;
    TCHAR    cFileName[ MAX_PATH ];
    TCHAR    cAlternateFileName[ 14 ];
} WIN32_FIND_DATA, *PWIN32_FIND_DATA;
```

Members

dwFileAttributes

Specifies the file attributes of the file found. This member can be one or more of the following values.

Attribute	Meaning
FILE_ATTRIBUTE_ARCHIVE	The file or directory is an archive file or directory. Applications use this attribute to mark files for backup or removal.
FILE_ATTRIBUTE_COMPRESSED	The file or directory is compressed. For a file, this means that all of the data in the file is compressed. For a directory, this means that compression is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_DIRECTORY	The handle identifies a directory.
FILE_ATTRIBUTE_ENCRYPTED	The file or directory is encrypted. For a file, this means that all data in the file is encrypted. For a directory, this means that encryption is the default for newly created files and subdirectories.
FILE_ATTRIBUTE_HIDDEN	The file or directory is hidden. It is not included in an ordinary directory listing.
FILE_ATTRIBUTE_NORMAL	The file or directory has no other attributes set. This attribute is valid only if used alone.
FILE_ATTRIBUTE_OFFLINE	The file data is not immediately available. This attribute indicates that the file data has been physically moved to offline storage. This attribute is used by Remote Storage, the hierarchical storage management software in Windows 2000. Applications should not arbitrarily change this attribute.
FILE_ATTRIBUTE_READONLY	The file or directory is read-only. Applications can read the file but cannot write to it or delete it. In the case of a directory, applications cannot delete it.
FILE_ATTRIBUTE_REPARSE_POINT	The file has an associated reparse point.
FILE_ATTRIBUTE_SPARSE_FILE	The file is a sparse file.

(continued)

(continued)

Attribute	Meaning
FILE_ATTRIBUTE_SYSTEM	The file or directory is part of the operating system or is used exclusively by the operating system.
FILE_ATTRIBUTE_TEMPORARY	The file is being used for temporary storage. File systems attempt to keep all of the data in memory for quicker access, rather than flushing it back to mass storage. A temporary file should be deleted by the application as soon as it is no longer needed.

ftCreationTime
Specifies a **FILETIME** structure containing the time the file was created. **FindFirstFile** and **FindNextFile** report file times in Coordinated Universal Time (UTC) format. These functions set the **FILETIME** members to zero if the file system containing the file does not support this time member. You can use the **FileTimeToLocalFileTime** function to convert from UTC to local time, and then use the **FileTimeToSystemTime** function to convert the local time to a **SYSTEMTIME** structure containing individual members for the month, day, year, weekday, hour, minute, second, and millisecond.

ftLastAccessTime
Specifies a **FILETIME** structure containing the time that the file was last accessed. The time is in UTC format; the **FILETIME** members are zero if the file system does not support this time member.

ftLastWriteTime
Specifies a **FILETIME** structure containing the time that the file was last written to. The time is in UTC format; the **FILETIME** members are zero if the file system does not support this time member.

nFileSizeHigh
Specifies the high-order **DWORD** value of the file size, in bytes. This value is zero unless the file size is greater than **MAXDWORD**. The size of the file is equal to (**nFileSizeHigh** * (**MAXDWORD**+1)) + **nFileSizeLow**.

nFileSizeLow
Specifies the low-order **DWORD** value of the file size, in bytes.

dwReserved0
If the **dwFileAttributes** member includes the FILE_ATTRIBUTE_REPARSE_POINT attribute, this member specifies the reparse tag. Otherwise, this value is undefined and should not be used.

dwReserved1
Reserved for future use.

cFileName
A null-terminated string that is the name of the file.

cAlternateFileName
A null-terminated string that is an alternative name for the file. This name is in the classic 8.3 (filename.ext) file name format.

Remarks

If a file has a long file name, the complete name appears in the **cFileName** field, and the 8.3 format truncated version of the name appears in the **cAlternateFileName** field. Otherwise, **cAlternateFileName** is empty. As an alternative, you can use the **GetShortPathName** function to find the 8.3 format version of a 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 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Unicode: Declared as Unicode and ANSI structures.

See Also

File I/O Overview, *File I/O Structures*, **FINDEX_INFO_LEVELS FindFirstFile**, **FindFirstFileEx**, **FindNextFile**, **FILETIME**, **FileTimeToLocalFileTime**, **FileTimeToSystemTime**, **GetShortPathName**

File I/O Enumeration Types

FINDEX_INFO_LEVELS

The **FINDEX_INFO_LEVELS** enumeration type defines values that are used with the **FindFirstFileEx** function to specify the information level of the returned data.

```
typedef enum _FINDEX_INFO_LEVELS {
    FindExInfoStandard
} FINDEX_INFO_LEVELS ;
```

Enumerator Value	Meaning
FindExInfoStandard	The **FindFirstFileEx** function obtains a standard set of attribute information. The data is returned in a **WIN32_FIND_DATA** structure.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

File I/O Overview, *File I/O Enumeration Types*, **FindFirstFileEx**, **WIN32_FIND_DATA**

FINDEX_SEARCH_OPS

The **FINDEX_SEARCH_OPS** enumeration type defines values that are used with the **FindFirstFileEx** function to specify the type of filtering to perform.

```
typedef enum _FINDEX_INFO_LEVELS {
    FindExSearchNameMatch,
    FindExSearchLimitToDirectories,
    FindExSearchLimitToDevices,
} FINDEX_INFO_LEVELS ;
```

Enumerator Value	Meaning
FindExSearchNameMatch	Search for a file that matches the specified file name. Note that *lpSearchFilter* parameter of **FindFirstFileEx** must be NULL when this search operation is used.
FindExSearchLimitToDevices	This filtering type is not available. For information on enumerating devices, see *Device Interfaces*.
FindExSearchLimitToDirectories	This is an advisory flag. If the file system supports directory filtering, the function searches for a "file" that matches the specified file name and that is a directory. If the file system does not support directory filtering, this flag is silently ignored. The *lpSearchFilter* parameter of **FindFirstFileEx** must be NULL when this search operation is used. If you want directory filtering, use this flag on all file systems, but be sure to examine the file attribute data stored into the **lpFindFileData* parameter of **FindFirstFileEx** to determine whether the function has indeed returned a handle to a directory.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

File I/O Overview, *File I/O Enumeration Types*, **FindFirstFileEx**

GET_FILEEX_INFO_LEVELS

The **GET_FILEEX_INFO_LEVELS** enumeration type defines values that are used with the **GetFileAttributesEx** function to specify the type of attribute information to obtain.

```
typedef enum _GET_FILEEX_INFO_LEVELS {
    GetFileExInfoStandard
} GET_FILEEX_INFO_LEVELS ;
```

Enumerator Value	Meaning
GetFileExInfoStandard	The **GetFileAttributesEx** function obtains a standard set of attribute information. The data is returned in a **WIN32_FILE_ATTRIBUTE_DATA** structure.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 98.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.

File I/O Overview, *File I/O Enumeration Types*, **GetFileAttributesEx**, **WIN32_FILE_ATTRIBUTE_DATA**

CHAPTER 10

File Systems

Win32-based applications rely on file systems to store and retrieve information on mass storage devices. File systems provide the underlying support that applications need to create and access files and directories on the individual volumes associated with the devices.

Each file system consists of one or more drivers and supporting dynamic-link libraries that define the data formats and features of the file system. These determine the conventions used for file names, the level of security and recoverability available, and the general performance of input and output (I/O) operations.

About File Systems

Depending on the configuration of a computer, a Win32-based application may have access to volumes managed by any of the following file systems: NTFS, FAT, or protected-mode FAT. Because the different volumes your application connects to can be managed by different file systems, it is important to understand the differences between file systems and to prepare your application to work effectively with all file systems.

Before you access files and directories on a given volume, you should determine the capabilities of the file system by using the **GetVolumeInformation** function. This function returns values that you can use to adapt your application to work effectively with the file system.

In general, you should avoid using static buffers for file names and paths. Instead, use the values returned by **GetVolumeInformation** to allocate buffers as you need them. If you must use static buffers, reserve 256 characters for file names and 260 characters for paths.

Shared File System Features

This section describes features common to both the NTFS and File Allocation Table (FAT) file systems: opportunistic locks.

Opportunistic Locks

An *opportunistic lock* (also called an oplock) is a lock placed by a client on a file residing on a server. A client places an opportunistic lock so it can cache data locally, thus reducing network traffic and improving apparent response time. Opportunistic locks are used by network redirectors on clients and by servers.

Opportunistic locks coordinate data caching and *coherency* between clients and servers and among multiple clients. Data that is *coherent* is data that is the same across the network. In other words, if data is coherent, data on the server and all the clients is synchronized.

Opportunistic locks are not commands by the client to the server. They are requests from the client to the server. From the point of view of the client, they are opportunistic. In other words, the server grants such locks whenever other factors make the locks possible.

Opportunistic locks and the associated operations implemented in the Microsoft Windows 2000 operating system are a superset of the opportunistic lock portion of the Common Internet File System (CIFS) protocol, an Internet Draft. The CIFS protocol is an enhanced version of the Server Message Block (SMB) protocol. For more information on the CIFS protocol, see Common Internet File System Protocol in the Networking Services section of the Platform Software Development Kit (SDK).

Note that the CIFS Internet Draft explicitly provides that a CIFS implementation may implement opportunistic locks by refusing to grant them.

For additional information on opportunistic locks, see the CIFS Internet Draft discussion on the topic. To learn the current status of any Internet Draft, check the 1id-abstracts.txt listing contained in the Internet Drafts shadow directories on ftp.is.co.za (Africa), nic.nordu.net (Europe), munnari.oz.au (Pacific Rim), ds.internic.net (U.S. East Coast), or ftp.isi.edu (U.S. West Coast). The file name for the current CIFS Internet Draft is in the format draft-leach-cifs*.txt, where * stands for specific draft information.

Any discrepancies between this discussion and the current CIFS Internet Draft should be resolved in favor of the Internet Draft.

Alternatives to Opportunistic Lock Operations

The following operations are of very limited use to Win32-based applications. Probably the only practical use for these operations at the Win32 level is to test a network redirector or a server opportunistic lock handler.

Typically, file systems implement support for opportunistic locks. Applications generally leave opportunistic lock management to the file system drivers. Anyone implementing a file system for Windows 2000 should use the Installable File System (IFS) Kit for Windows 2000.

Anyone developing a device driver other than an installable file system should use the Windows 2000 Driver Development Kit (DDK).

Local Caching

Local caching of data is a technique used to speed network access to data files. It simply means caching data when possible on clients rather than on servers.

Local caching of data to be written allows multiple write operations on the same region of a file to be combined into one write operation across the network. Local caching reduces

network traffic because the data is written once. Such caching improves the apparent response time of applications because the applications do not wait for the data to be sent across the network to the server.

Local caching of data to be read can appear to speed things up by means of reading ahead. A simple example is an application accessing data sequentially, such as a compiler's preprocessor. In such cases, the network layer of the operating system reads data across the network before the application requests the data. Ideally, the network delivers the data before the application requests it from the file system, resulting in near-instantaneous response. In practice, this rarely happens, but often reading ahead speeds applications by anticipating the next request.

Local caching can also help reduce network traffic by reading a portion of a file across the network once and then keeping it in the local cache. Subsequent read operations on that portion by the application read from the local cache.

One type of application that can benefit from local caching is batch files. Command processors such as the Command.com command processor in Microsoft Windows 95 read and execute a batch file one line at a time. For each line, the command processor opens the file, searches to the beginning of the line, reads as much as it needs, closes the file, then executes the line. Each line results in a lot of network traffic. Network traffic can be reduced considerably by caching the entire batch file on a client.

Local caching also helps with another problem associated with networks, especially networks that perform work over modems and other thin pipes: slow response time. Users do not want to wait while data is retrieved over the network, modified, and then written back. With reading ahead and write caching, it often appears that these functions operate much faster than they actually do.

A hazard of local caching is that written data only has as much integrity as the client itself for as long as the data is cached on the client. With modern operating systems and hardware support such as uninterruptible power supplies, the risk of losing locally cached data is reduced. But the risk still exists, and you should consider both the trade-off between data integrity and apparent response speed and the trade-off between data integrity and reduced network traffic.

If written data is cached on the client, the data must be flushed to the server as soon as possible.

Data Coherency

Data that is *coherent* is data that is the same across the network. In other words, if data is coherent, data on the server and all the clients is synchronized. One type of software system that provides data coherency is a revision control system (RCS). Such a system is usually fairly simple, with only one user allowed to modify a specified file at a time. Others can read the file but cannot change it.

The user who can change a file is said to have checked it out. The user then checks in the modified file so that others may see the changes. Only after the user has checked a file back in can another user check it out.

An RCS requires the active intervention of users to operate in a useful manner. A file system that operates across a network should handle the problem automatically.

Providing local caching of coherent data is fairly simple when you have one thread on one client accessing a file across the network at a time. However, in most situations many different threads on one or more computers may be reading the same file. This situation is still fairly straightforward. Because the data in the file is static, each client computer can have its own local copy with no implications for data coherency.

A more common situation is one thread modifying the file, and a lot of other threads reading it. The moment a write operation occurs, all of the local caches of that file are obsolete. The server must notify each client to abandon its cache. Any subsequent read operations for the file must be performed across the network.

In another common situation, multiple threads on one or more network clients might try to write to the same file. This situation is similar to a one in which several RCS users all want to make changes to the same file. Each user in sequence must check out the file, make changes, and then check the file back in. Similarly, in a local caching scheme the server must hand off the privilege of writing to a file to one client thread at a time.

Typical Use

The typical way to use opportunistic locks is first to open a file with permissions and flags appropriate to the application opening the file. All files for which opportunistic locks will be requested must be opened for overlapped (asynchronous) operation. Once the files are opened for overlapped operation, use the **DeviceIoControl** function with the appropriate operation to request an opportunistic lock. For a list of the opportunistic lock operations, see *Opportunistic Lock Operations*.

Applications are notified that an opportunistic lock is broken by using the **hEvent** member of the **OVERLAPPED** structure associated with the file. Applications may also use functions such as **GetOverlappedResult** and **HasOverlappedIoCompleted**. The application is responsible for associating the correct file with the broken opportunistic lock.

For more information on notification, see *Synchronization*.

Server Response to Open Requests on Locked Files

The life of an opportunistic lock includes three distinct time spans. During each, the server determines by different means its reaction to a request from a client to open a file locked by another client. In general, you can minimize the impact your application has on other clients and the impact they have on your application by granting as much sharing as possible, requesting the minimum access level necessary, and using the least intrusive opportunistic lock suitable for your application.

First is the period after the server opens a file for a client but before it grants a lock. During this time, no lock exists on the file, and the server depends on sharing, access modes, and the type of opportunistic lock you request to determine its reaction to another request to open the same file. For example, if you open the file in question for write access, you may inhibit granting opportunistic locks that allow read caching access

to other clients. The time span before the server grants a lock is typically in the millisecond range but may be longer.

Once the opportunistic lock is granted, the server examines the lock to determine server reaction to an open request on a locked file. Again, how your application opened the file and the type of lock it holds affects how the server responds. For more information on how the server responds in each case, see *Level 1 Opportunistic Locks*, *Level 2 Opportunistic Locks*, *Batch Opportunistic Locks*, and *Filter Opportunistic Locks*.

Finally, there is the span after the server determines that your lock should be broken (ended) but before your application completes its reaction to the break. Depending on the type of lock, your application can downgrade the lock to a lower level or to none at all. Your application can also close the file and the lock. During this time, the server holds in abeyance any requests from other clients to open the formerly locked file. This time span may range from milliseconds to tens of seconds. For more information, see *Breaking Opportunistic Locks*.

Types of Opportunistic Locks

The opportunistic lock operations work with four types of opportunistic locks: level 1, level 2, batch, and filter. *Exclusive opportunistic locks* are level 1, batch, and filter locks. If a thread has any type of exclusive lock on a file, it cannot also have a level 2 lock on the same file.

Level 1 Opportunistic Locks

A level 1 opportunistic lock on a file allows a client to read ahead in the file and cache both read-ahead and write data from the file locally. As long as the client has sole access to a file, there is no danger to data coherency in providing a level 1 opportunistic lock.

The client can request a level 1 opportunistic lock after opening a file. If no other client (or no other thread on the same client) has the file open, the server may grant the opportunistic lock. The client can then cache read and write data from the file locally. If another client has opened the file, then the server refuses the opportunistic lock and the client does no local caching. (The server may refuse the opportunistic lock for other reasons as well, such as not supporting opportunistic locks.)

When the server opens a file that already has a level 1 opportunistic lock on it, the server examines the sharing state of the file before it breaks the level 1 opportunistic lock. If the server breaks the lock after this examination, the time the client with the former lock spends flushing its write cache is time the client requesting the file must wait. This time expenditure means that level 1 opportunistic locks should be avoided on files that many clients open.

However, because the server checks the sharing state before it breaks the lock, in the case where the server would deny an open request due to a sharing conflict the server does not break the lock. For example, if you have opened a file, denied sharing for write operations, and obtained a level 1 lock, the server denies another client's request to

open the file for writing before it even examines your lock on the file. In this instance, your opportunistic lock is not broken.

For an example of how a level 1 opportunistic lock works, see *Level 1 Opportunistic Lock Example*. For more information on breaking opportunistic locks, see *Breaking Opportunistic Locks*.

Level 2 Opportunistic Locks

A level 2 opportunistic lock informs a client that there are multiple concurrent clients of a file and that none has yet modified it. This lock allows the client to perform read operations and obtain file attributes using cached or read-ahead local information, but the client must send all other requests (such as for write operations) to the server. Your application should use the level 2 opportunistic lock when you expect other applications to write to the file at random or read the file at random or sequentially.

A level 2 opportunistic lock is very similar to a filter opportunistic lock. In most situations, your application should use the level 2 opportunistic lock. Only use the filter lock if you do not want open operations for reading to cause sharing-mode violations in the time span between your application's opening the file and receiving the lock. For more information, see *Filter Opportunistic Locks* and *Server Response to Open Requests on Locked Files*.

For more information on breaking opportunistic locks, see *Breaking Opportunistic Locks*.

Batch Opportunistic Locks

A batch opportunistic lock manipulates file openings and closings. For example, in the execution of a batch file, the batch file may be opened and closed once for each line of the file. A batch opportunistic lock opens the batch file on the server and keeps it open. As the command processor "opens" and "closes" the batch file, the network redirector intercepts the open and close commands. All the server receives are the seek and read commands. If the client is also reading ahead, the server receives a particular read request at most one time.

When opening a file that already has a batch opportunistic lock, the server checks the sharing state of the file after breaking the lock. This check gives the holder of the lock a chance to complete flushing its cache and to close the file handle. An open operation attempted during the sharing check does not cause the sharing check to fail if the lock holder releases the lock.

For an example of how a batch opportunistic lock works, see *Batch Opportunistic Lock Example*. For more information on breaking opportunistic locks, see *Breaking Opportunistic Locks*.

Filter Opportunistic Locks

A filter opportunistic lock locks a file so that it cannot be opened for either write or delete access. All clients must be able to share the file. Filter locks allow applications to perform nonintrusive filtering operations on file data (for example, a compiler opening source code or a cataloging program).

A filter opportunistic lock differs from a level 2 opportunistic lock in that it allows open operations for reading to occur without sharing-mode violations in the time span between your application's opening the file and receiving the lock. The filter opportunistic lock is the best lock to use when it is important to allow other clients reading access. In other cases, your application should use a level 2 opportunistic lock. A filter opportunistic lock differs from a batch opportunistic lock in that it does not allow multiple openings and closings to be handled by the network redirector the way a batch opportunistic lock does.

Your application should request a filter opportunistic lock on a file in three steps:

1. Use the **CreateFile** function to open a handle to the file with the *DesiredAccess* parameter set to zero, indicating no access, and the *dwShareMode* parameter set to the FILE_SHARE_READ flag to allow sharing for reading. The handle obtained at this point is called the locking handle.
2. Request a lock on this handle with the **DeviceIoControl** operation **FSCTL_REQUEST_FILTER_OPLOCK**.
3. When the lock is granted, use **CreateFile** to open the file again with *DesiredAccess* set to the GENERIC_READ flag if you want read access, the GENERIC_WRITE flag if you want write access, or both. Set *dwShareMode* to the FILE_SHARE_READ flag to allow others to read the file while you have it open, the FILE_SHARE_DELETE flag to allow others to mark the file for deletion while you have it open, or both. The handle obtained at this point is called the read handle.

Use the read handle to read from or write to the contents of the file.

When opening a file that already has a filter opportunistic lock, the server breaks the lock and then checks the sharing state of the file. This check gives the client that held the former opportunistic lock a chance to abandon any cached data and acknowledge the break. An open operation attempted during this sharing check does not cause the sharing check to fail if the former lock holder releases the lock. The file system holds in abeyance the open operation until the lock's owner closes both the read handle and then the locking handle. Once this is done, the other client's open request can proceed.

Breaking Opportunistic Locks

Breaking an opportunistic lock is the process of degrading the lock that one client has on a file so that another client can open the file, with or without an opportunistic lock. When the other client requests the open operation, the server delays the open operation and notifies the client holding the opportunistic lock.

The client holding the lock then takes actions appropriate to the type of lock, for example abandoning read buffers, closing the file, and so on. Only when the client holding the opportunistic lock notifies the server that it is done does the server open the file for the client requesting the open operation. However, when a level 2 lock is broken, the server reports to the client that it has been broken but does not wait for any acknowledgement, as there is no cached data to be flushed to the server.

In acknowledging a break of any exclusive lock (filter, level 1, or batch), the holder of a broken lock cannot request another exclusive lock. It can degrade an exclusive lock to a

level 2 lock or no lock at all. The holder typically releases the lock and closes the file when it is about to close the file anyway.

Applications are notified that an opportunistic lock is broken by using the **hEvent** member of the **OVERLAPPED** structure associated with the file on which the lock is broken. Applications may also use functions such as **GetOverlappedResult** and **HasOverlappedIoCompleted**.

Opportunistic Lock Examples

The following examples show data and SMB message movements as opportunistic locks are made and broken. Note that clients can cache file attribute data as well as file data.

Level 1 Opportunistic Lock Example

The following diagram shows a network-traffic view of a level 1 opportunistic lock on a file. The arrows indicate the direction of data movement, if any.

Event	Client X	Server	Client Y
1	Opens file, requests level 1 lock →		
2		← Grants level 1 opportunistic lock	
3	Performs read, write, and other operations →		
4			← Requests to open file
5		← Breaks opportunistic lock	
6	Discards read-ahead data		
7	Writes data →		
8	Sends "close" or "done" message →		
9		Okays open operation →	
10	Performs read, write, and other operations →		← Performs read, write, and other operations

In event 1, client X opens a file and as part of the open operation requests a level 1 opportunistic lock on the file. In event 2, the server grants the level 1 lock because no other client has the file open. The client proceeds to access the file in the usual manner in event 3.

In event 4, client Y attempts to open the file and requests an opportunistic lock. The server sees that client X has the file open. The server ignores Y's request while client X flushes any write data and abandons its read cache for the file.

The server forces X to clean up by sending to X an SMB message breaking the opportunistic lock, event 5. Client X "silently" discards any read-ahead data; in other words, this process generates no network traffic. In event 7, client X writes any cached write data to the server. When client X is done writing cached data to the server, client X sends either a "close" or a "done" message to the server, event 8.

Once the server has been notified that client X is done flushing its write cache to the server or has closed the file, then the server can open the file for client Y, in event 9. Because the server now has two clients with the same file open, it grants an opportunistic lock to neither. Both clients proceed to read from the file, and one or neither writes to the file.

Batch Opportunistic Lock Example

The following diagram shows a network-traffic view of a batch opportunistic lock. The arrows indicate the direction of data movement, if any.

Event	Client X	Server	Client Y
1	Opens file, requests batch lock →		
2		← Grants batch opportunistic lock	
3	Reads file →		
4		← Sends data	
5	Closes file		
6	Opens file		
7	Searches for data		
8	Reads data →		
9		← Sends data	
10	Closes file		
11			← Opens file
12		← Breaks opportunistic lock	
13	Closes file →		
14		Okays open operation →	
15			← Performs read, write, and other operations

In the batch opportunistic lock, client X opens a file, event 1, and the server grants client X a batch lock in event 2. Client X attempts to read data, event 3, to which the server responds with data, event 4.

Event 5 shows the batch opportunistic lock at work. The application on Client X closes the file. However, the network redirector filters out the close operation and does not transmit a close message, thus performing a "silent" close. The network redirector can do this because client X has sole ownership of the file. Later on, in event 6, the application reopens the file. Again, no data flows across the network. As far as the server is concerned, this client has had the file open since event 2.

Events 7, 8, and 9 show the usual course of network traffic. In event 10, another silent close occurs.

In event 11, client Y attempts to open the file. The server's view of the file is that client X has it open, even though the application on client X has closed it. Therefore, the server sends an a message breaking the opportunistic lock to client X. Client X now sends the close message across the network, event 13. Event 14 follows as the server opens the file for client Y. The application on client X has closed the file, so it does no more transfers to or from the server for that file. Client Y begins data transfers as usual in event 15.

Between the time client X is granted the lock on the file in event 2 and the final close at event 13, any file data that the client has cached is valid, in spite of the intervening application open and close operations. However, after the opportunistic lock is broken, cached data cannot be considered valid.

Filter Opportunistic Lock Example

The following diagram shows a network-traffic view of a filter opportunistic lock. The arrows indicate the direction of data movement, if any.

Event	Client X	Server	Client Y
1	Opens file with no access rights →		
2		← Opens the file	
3	Requests filter lock →		
4		← Grants lock	
5	Opens file for reading →		
6		← Reopens the file	
7	Reads data using the read handle →		
8		← Sends data	
9		← Sends data	
10		← Sends data	

11			← Opens the file
12		Opens the file →	
13			← Requests filter lock
14		Denies filter lock →	
15			← Reads data
16		Sends data →	
17	Reads (cached) data		
18	Closes file →		
19			← Closes file

In the filter opportunistic lock, client X opens a file, event 1, and the server responds in event 2. The client then requests a filter opportunistic lock in event 3, followed by the server granting the opportunistic lock in event 4. Client X then opens the file again for reading in event 5, to which the server responds in event 6. The client then attempts to read data, to which the server responds with data, event 8.

Event 9 shows the filter opportunistic lock at work. The server reads ahead of the client and sends the data over the network even though the client has not requested it. The client caches the data. In event 10, the server also anticipates a future request for data and sends another portion of the file for the client to cache.

In event 11 and 12, another client, Y, opens the file. Client Y also requests a filter opportunistic lock. In event 14, the server denies it. In event 15, client Y requests data, which the server sends in event 16. None of this affects client X. At any time, another client can open this file for read access. No other client affects client X's filter lock.

Event 17 shows client X reading data. However, because the server has already sent the data and the client has cached it, no traffic crosses the network.

In this example, client X never tries to read all the data in the file, so the read-ahead indicated by events 9 and 10 is "wasted"; that is, the data is never actually used. This is an acceptable loss because the read-ahead has sped up the application.

In event 18, client X closes the file. The client's network redirector abandons the cached data. The server closes the file.

Opportunistic Lock Operations

If a Win32 application requests opportunistic locks, all files for which it requests locks must be opened for overlapped (asynchronous) input and output using the **CreateFile** function with the FILE_FLAG_OVERLAPPED flag. Once the files are opened for overlapped operation, you can use the **DeviceIoControl** function with one of the operations listed following to work with those files' opportunistic locks.

FSCTL_OPBATCH_ACK_CLOSE_PENDING
FSCTL_OPLOCK_BREAK_ACK_NO_2
FSCTL_OPLOCK_BREAK_ACKNOWLEDGE

FSCTL_OPLOCK_BREAK_NOTIFY
FSCTL_REQUEST_BATCH_OPLOCK
FSCTL_REQUEST_FILTER_OPLOCK
FSCTL_REQUEST_OPLOCK_LEVEL_1
FSCTL_REQUEST_OPLOCK_LEVEL_2

NTFS File System

Microsoft Windows NT/Windows 2000 provides support for the NTFS file system. This file system supports object-oriented applications by treating all files as objects with user- and system-defined attributes. NTFS provides all the capabilities of the FAT file system without many of its limitations. Accessing files under NTFS is often faster than accessing similar files under the FAT file system.

NTFS also includes features not present in FAT, such as security, Unicode file names, automatic creation of MS-DOS aliases, multiple data streams, and unique functionality specific to the POSIX subsystem. For more information about security, see *Access Control*. For more information about Unicode, see *Unicode and Character Sets*.

NTFS file names can be any practical length (up to 255 characters). There is no requirement that NTFS file names have extensions; however, many applications still create and use them. For more information, see *File Name Conventions*.

The following features are supported on NTFS:

- File System Recovery
- File Compression
- File Encryption
- Disk Quotas
- Sparse Files
- Distributed Link Tracking
- Reparse Points
- Volume Mount Points and Mounting Volumes

File System Recovery

NTFS is a fully recoverable file system. It is designed to restore consistency to a disk after a CPU failure, system crash, or I/O error. NTFS allows the operating system to recover without your having to use disk-checking utilities. However, NTFS provides some disk utilities in case recovery fails or corruption occurs outside the control of the file system.

File Compression

Windows NT 3.51 and later support file compression on an individual file basis for NTFS volumes.

Compression Attribute

On NTFS volumes, each file and directory has a *compression attribute*. Other file systems may also implement a compression attribute for individual files and directories.

You can determine whether a file system supports a compression attribute for files and directories by calling the **GetVolumeInformation** function and examining the FS_FILE_COMPRESSION bit flag.

Use the **GetFileAttributes** or **GetFileAttributesEx** function to determine the compression attribute of a file or directory.

If a file's compression attribute is set, all of the data in the file is compressed. If the attribute is clear, none of the data in the file is compressed. There is no partially compressed state. The compression attribute is a simple Boolean indicator of compression state.

A directory's compression attribute provides a default compression attribute for newly created files and subdirectories. When you call **CreateFile** or **CreateDirectory** to create a new file or directory, the new file or directory inherits the compression attribute of its parent directory.

Compression State

Each file and directory on a volume that supports compression for individual files and directories has a *compression state*.

Whereas the compression attribute of a file or directory indicates simply whether the file or directory is compressed or not compressed, the compression state also specifies the format of any compressed data.

Use the **FSCTL_GET_COMPRESSION DeviceIoControl** operation to determine the compression state of a file or directory.

Compression state is encoded as a 16-bit value. A compression state value of COMPRESSION_FORMAT_NONE indicates that a file is not compressed. A value of COMPRESSION_FORMAT_DEFAULT indicates that a file is compressed, using the default compression format. Any other value indicates that a file is compressed, using the compression format specified by the compression state value.

Use the **FSCTL_SET_COMPRESSION DeviceIoControl** operation to set the compression state of a file or directory. This operation also sets the compression attribute of the file or directory.

Setting the compression state of a file to a nonzero value compresses the file, using the compression format encoded by the compression state value. Setting a file's compression state to zero decompresses the file. These are synchronous operations.

The file is compressed or decompressed immediately when you set its compression state.

Setting a directory's compression state does not cause any immediate compression or decompression. Instead, setting a directory's compression state sets a default compression state that will be given to all newly created files and subdirectories.

Obtaining the Size of a Compressed File

Use the **GetCompressedFileSize** function to obtain the compressed size of a file. If the file is compressed, its compressed size will be less than its uncompressed size. Use the **GetFileSize** function to determine the uncompressed size of a file.

File Encryption

Windows 2000 supports the Encrypted File System (EFS), which provides cryptographic protection of files on NTFS volumes. EFS provides file encryption on an individual file basis using a public-key system.

Note that EFS encryption and NTFS file compression are mutually exclusive; you cannot compress an encrypted file. However, encrypted files tend to be compressed already, due to the nature of cryptographic algorithms. Sparse files may be encrypted.

You can determine whether a file system supports file encryption for files and directories by calling the **GetVolumeInformation** function and examining the FILE_SUPPORTS_ENCRYPTION bit flag. Note that the following items cannot be encrypted:

- system files
- system directories
- root directories

Handling Encrypted Files and Directories

A programmer or user may mark a directory or file as encrypted. A file so marked is encrypted by NTFS using the current encryption driver. If the file is later marked as not encrypted, it is decrypted and left in a plaintext (unsecured) state.

Directories are not themselves encrypted. Rather, by default, in an "encrypted" directory all new files in the directory are encrypted at creation. A user must specifically change the status of a new file to decrypted if the user does not want the file to be encrypted. An encrypted directory is visible. To make a directory inaccessible to other users, use the standard methods of access control.

To encrypt a file, use the **CreateFile** function with the FILE_ATTRIBUTE_ENCRYPTED flag. To encrypt an existing file, use the **EncryptFile** function. All data streams in the file are encrypted. If the file is already encrypted, **EncryptFile** does nothing but return a nonzero value, which indicates success. If the file is compressed, **EncryptFile** decompresses the file before encrypting it.

To decrypt an encrypted file, use the **DecryptFile** function. If the file is not encrypted, **DecryptFile** does nothing but return a nonzero value indicating success.

The **EncryptionDisable** function disables or enables encryption of the indicated directory and the files in it. It does not affect encryption of subdirectories below the indicated directory.

To retrieve the encryption status of a file, use the **FileEncryptionStatus** function. Alternatively, call the **GetFileAttributes** function and examine the FILE_ATTRIBUTE_ENCRYPTED flag in the return value.

Encrypted Files and User Keys

To create a new key for a user, use the **SetUserFileEncryptionKey** function. To add user keys to an encrypted file, use the **AddUsersToEncryptedFile** function. To query the user keys for an encrypted file, use the **QueryUsersOnEncryptedFile** function. To remove user keys from an encrypted file, use the **RemoveUsersFromEncryptedFile** function.

Disk Quotas

NTFS version 5.0 supports *disk quotas*, which allow administrators to control the amount of data that each user can store on an NTFS volume. Administrators can optionally configure the system to log an event when users are near their quota, and to deny further disk space to users who exceed their quota. Administrators can also generate reports and use the event monitor to track quota issues.

You can determine whether a file system supports disk quotas by calling the **GetVolumeInformation** function and examining the FILE_VOLUME_QUOTAS bit flag.

Disk quotas are transparent to the user. When a user asks how much space is free on a disk, the system reports only the available quota allowance the user has available. If the user exceeds this allowance, the system returns the same error it would return to indicate that the disk was full.

To obtain more free disk space after exceeding the quota allowance, the user must do one of the following:

- Delete some files.
- Have another user claim ownership of some files.
- Have the administrator increase the quota allowance.

Programs that need to retrieve the actual amount of free disk space can call the **GetDiskFreeSpaceEx** function and look at the *TotalNumberOfFreeBytes* parameter.

Disk Quota Limits

The disk space used by each file is charged directly to the user who owns the file. The owner of a file is identified by the security identifier (SID) in the security information for the file. The total disk space charged to a user is the sum of the length of all data streams. In other words, property set streams and resident user data streams affect the user's quota. Compressing or decompressing files does not affect the disk space

reported for the files. Therefore, quota settings on one volume can be compared to settings on another volume.

The following are the types of disk quota limits:

- Warning threshold. You can configure the system to generate a system logfile entry when the disk space charged to the user exceeds this value.
- Hard quota. You can configure the system to generate a system logfile entry when the disk space charged to the user exceeds this value. You can also configure the system to deny additional disk space to the user when the disk space charged to the user exceeds this value.

NTFS automatically creates a user quota entry when a user first writes to the volume. Entries that are created automatically are assigned the default warning threshold and hard quota limit values for the volume.

Disk Quota States

The administrator can turn quota enforcement on and off. There are three quota states, as shown in the following table.

State	Description
Quota disabled	Quota usage changes are not tracked, but the quota limits are not removed. In this state, performance is not affected by disk quotas. This is the default state.
Quota tracked	Quota usage changes are tracked, but quota limits are not enforced. In this state, no quota violation events are generated and no file operations fail because of disk quota violations.
Quota enforced	Quota usage changes are tracked and quota limits are enforced.

Administering Disk Quotas

The administrator can set quotas for specific users on a volume. The administrator can also set default quotas for the volume. A new user on the volume receives the default quota unless the administrator established a quota specifically for that user.

The administrator can query the level of quota tracking, the default quota limits, and the per-user quota information. The per-user quota information contains the user's hard quota limit, warning threshold, and the quota usage.

Sparse Files

A very large file without a lot of data is said to contain a sparse data set. Applications that use sparse data sets include image processors and high-speed databases. In versions of NTFS prior to version 5.0, the portions of the file that did not contain useful data occupied valuable disk space.

The file compression introduced in NTFS 3.51 is a partial solution to the problem. The portions of the file that do not contain useful data are set to zero, and file compression compacts the non-data portions. However, file compression has its own drawbacks. Access time may increase due to data compression and decompression.

Windows 2000 NTFS introduces another solution, called a *sparse file*. When the sparse file facilities are used, the system does not allocate hard drive space to a file except in regions where it contains something other than zeros. The default data value of a sparse file is zero.

Sparse File Operations

You can determine whether a file system supports sparse files by calling the **GetVolumeInformation** function and examining the FILE_SUPPORTS_SPARSE_FILES bit flag.

Most applications are not aware of sparse files and will not create sparse files. The fact that an application is reading a sparse file is transparent to the application. An application that is aware of sparse-files should determine whether its data set is suitable to be kept in a sparse file. After that determination is made, the application must explicitly declare a file as sparse, using the **FSCTL_SET_SPARSE DeviceIoControl** operation.

After an application has set a file to be sparse, the application can use the **FSCTL_SET_ZERO_DATA DeviceIoControl** operation to set a region of the file to zero. In addition, the application can use the **FSCTL_QUERY_ALLOCATED_RANGES DeviceIoControl** operation to speed searches for nonzero data in the sparse file.

When you perform a write operation (with a function or operation other than **FSCTL_SET_ZERO_DATA**) whose data consists of nothing but zeros, zeros will be written to the disk for the entire length of the write. To zero out a range of the file and maintain sparseness, use **FSCTL_SET_ZERO_DATA**.

A sparseness-aware application may also set an existing file to be sparse. If an application sets an existing file to be sparse, it should then scan the file for regions which contain zeros, and use **FSCTL_SET_ZERO_DATA** to reset those regions, thereby possibly deallocating some physical disk storage. An application upgraded to sparse file awareness should perform this conversion.

When you perform a read operation from a zeroed-out portion of a sparse file, the operating system may not read from the hard drive. Instead, the system recognizes that the portion of the file to be read contains zeros, and it returns a buffer full of zeros without actually reading from the disk.

As with any other file, the system can write data to or read data from any position in a sparse file. Nonzero data being written to a previously zeroed portion of the file may result in allocation of disk space. Zeros being written over nonzero data (only with **FSCTL_SET_ZERO_DATA**) may result in a deallocation of disk space.

Note It is up to the application to maintain sparseness by writing zeros with **FSCTL_SET_ZERO_DATA**.

Defragmenting tools that handle compressed files on NTFS file systems will correctly handle sparse files on Windows 2000 NTFS volumes.

Obtaining the Size of a Sparse File

Use the **GetCompressedFileSize** function to obtain the actual size allocated on disk for a sparse file. This total does not include the size of the regions which were deallocated because they were filled with zeroes. Use the **GetFileSize** function to determine the total size of a file, including the size of the sparse regions that were deallocated.

Sparse Files and Disk Quota

A sparse file affects user quotas by the nominal size of the file, not the actual allocated amount of disk space. That is, creating a 50-MB file with all zero bytes consumes 50 MB of that user's quota. This means that as the user writes data to the sparse file, he need not worry about exceeding his hard quota limit, because he has already been charged for the space. System administrators should not count on the size of a sparse file remaining small. Therefore they should not grant their users hard quota limits that exceed the physical space available, despite the use of sparse files.

Distributed Link Tracking

Windows 2000 provides the distributed link-tracking service enabling client applications to track link sources that have been moved. As a result, clients that subscribe to the link-tracking service can maintain the integrity of their references, and the objects referenced can be moved transparently.

A link source is an object referenced by a link client. For example, in a Microsoft Word document that contains an OLE link to a Microsoft Excel worksheet, the Word document is the link client and the Excel worksheet is the link source.

The distributed link-tracking service tracks link sources for shell shortcuts and OLE links within NTFS version 5.0 volumes. The link client can mend the broken link with updated information on the new location of the link source.

Link Tracking Features

Shell shortcuts in Windows 95/98 include heuristic link tracking that uses a tree-search algorithm to find a likely match for a moved link source. The search algorithm is based on the last known path of the file as well as file information that includes the creation date, the file size, and the file name and extension. OLE linking includes the same heuristic link tracking. Windows NT 4.0 also includes the same heuristic link tracking with some added improvements in searching name spaces to yield results in some common scenarios. The improvements include the following steps subject to time limits imposed by the client application:

1. Search four directory levels down from the last directory.
2. Move up one directory and repeat steps 1 and 2 another three times which can yield results if the object has been moved nearby.
3. Search four levels down from the desktop root which can yield results if the object has been moved to a location on the same desktop.
4. Search four levels down from the root on each local fixed drive.
5. Repeat steps 1–3 without the four directory limit.

These link-tracking schemes are transparent to the end user. However, they do not always yield positive results and can be time consuming.

Windows 2000 adds a new distributed link tracking service for NTFS. The distributed link-tracking service can be used to track links to files. This service is used by shell shortcuts and by OLE links. See the topic on the **IShellLink** interface for more information on shell shortcuts. See the topic on the **IOleLink** interface for more information on OLE links. If a link is made to a file on an NTFS version 5.0 volume and the file is moved to any other NTFS version 5.0 volume within the same domain, the file will be found by the tracking service, subject to time considerations. Additionally, if the file is moved outside the domain or within a workgroup, it will most likely be found.

When a link is created to a file, the target file is considered the *link source* and the creator of the link is the *link client*. For example, if a shell shortcut is created to link to a text document, the text document is the link source and the shell shortcut is the link client.

The distributed link-tracking service maintains file links for the following situations occurring within a domain:

- The link source file is moved from one NTFS version 5.0 volume to another NTFS version 5.0 volume within the same domain.
- The name of the machine that holds the link source is renamed.
- The network shares on the link source machine are changed.
- The volume holding the link source file is moved to another machine within the same domain.

The distributed link-tracking service also attempts to maintain links in the preceding situations even when they do not occur within a domain, i.e., they are cross-domain or within a workgroup. Links can always be maintained in these situations when the network share on the link source machine is changed. They can also be maintained when a link source is moved within a machine. Links can usually be maintained when the link source is moved to another machine, though this form of tracking is less reliable over time.

Link Tracking Components

This section describes how distributed link tracking is implemented.

Link tracking functionality is primarily implemented in the form of two system services.

Tracking Service
The tracking service runs on all machines and manages the link-tracking activities for that machine. These activities include searching for link sources and processing link source moves. When a link source is moved, the service passes information to the Tracking (Server) Service which runs on domain controllers (DCs), and which is described below.

Tracking (Server) Service
The server portion of the tracking service runs on each domain controller in a domain. The service accepts notifications of file and volume moves from the tracking service on a given machine and allows the tracking services to query a link source's current location.

This server service maintains information in the DC about volumes and files which have been moved. The information on moves cannot grow above a certain size and it is automatically removed if it becomes incorrect or stale.

The link tracking services are exposed via the **IShellLink** and **IOleLink** interfaces. Thus, they are used by shell shortcuts. When the **IShellLink::Resolve** method is called and the referent file cannot be found (for example, when the user activates a shell shortcut), the tracking service is called automatically to find the file. Similarly, when the **IOleLink** implementation cannot find a file, for example in its **BindToSource** method, it automatically calls on the tracking service.

Link Tracking Limitations

The distributed link-tracking services are available only on Windows 2000, and are only available for link sources on Windows 2000 NTFS volumes. Thus if a link source is moved to a non-Windows 2000 NTFS volume (for example, to a FAT volume), or if a link source is moved to a computer running Windows NT 4.0, the tracking information is lost. Additionally, if a link source is moved even between Windows 2000 NTFS volumes, but the computer performing the move is running an earlier version of Windows NT or Windows 95/98, the link tracking information is lost. When the link tracking information is lost, no harm is done to the link-source file itself, it is simply not trackable by the distributed link-tracking services.

Links to files on removable media are not maintained. Also, the tracking service does not recognize a new NTFS volume until the system is rebooted. A new volume might become available because of repartitioning, reformatting a FAT volume to NTFS, or connecting a new external drive.

Reparse Points

On a Windows 2000 NTFS volume, a file or directory can contain a *reparse point*, which is a collection of user-defined data. The format of this data is understood by the application which stores the data, and a *file system filter*, which you install to interpret the data and process the file. When an application sets a reparse point, it stores this data, plus a *reparse tag*, which uniquely identifies the data it is storing. When the file system

opens a file with a reparse point, it attempts to find the file system filter associated with the data format identified by the reparse tag. If such a file system filter is found, the filter processes the file as directed by the reparse data. If no such file system filter is found, the file open operation fails.

For example, reparse points are used to implement NTFS links and the Microsoft Remote Storage Server (RSS). RSS uses an administrator-defined set of rules to move infrequently used files to long term storage, such as tape or CD-ROM. It uses reparse points to store information about the file in the file system. This information is stored in a stub file that contains a reparse point whose data points to the device where the actual file is now located. The file system filter can use this information to retrieve the file.

Reparse Point Tags

Each reparse point has an identifier tag so that you can efficiently differentiate between the different types of reparse points, without having to examine the user-defined data in the reparse point. Windows 2000 has a set of predefined tags and a range of tags reserved for Microsoft. If you use any of the reserved tags when setting a reparse point, the operation fails. Tags not included in these ranges are not reserved and are available for your application.

When you set a reparse point, you must tag the data to be placed in the reparse point. After the reparse point has been established, a new set operation fails if the tag for the new data does not match the tag for the existing data. If the tags match, the set operation overwrites the existing reparse point.

Tag Contents

Reparse tags are stored as **ULONG** values. The bits define certain attributes, as shown in the following diagram.

```
 3 3 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1
 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0 9 8 7 6 5 4 3 2 1 0
+-+-+-+-----------------------+-------------------------------+
|M|L|N|    Reserved bits      |          Tag Value            |
+-+-+-+-----------------------+-------------------------------+
```

The low 16 bits determine the kind of reparse point. The high 16 bits have 13 bits reserved for future use and 3 bits that denote specific attributes of the tags and the data represented by the reparse point. The following table describes these bits.

Bit	Description
M	Microsoft bit. If this bit is set, the tag is owned by Microsoft. All other tags must use zero for this bit.
L	High-latency bit. If this bit is set, the operating system is expected to be slow to retrieve the first byte of data. Your application should display some indication to the user that the operation is in progress.
N	Name surrogate bit. If this bit is set, the file represents another named entity in the system.

A set of macros is defined in the Winnt.h header file to assist in testing tags. The set includes:

- **IsReparseTagHighLatency**
- **IsReparseTagMicrosoft**
- **IsReparseTagNameSurrogate**

Each macro returns a nonzero value if the associated bit is set.

To ensure uniqueness of tags, Microsoft provides a mechanism to distribute new tags. For more information, see the Microsoft Installable File System (IFS) kit.

Reparse Point Operations

You can determine whether a file system supports reparse points by calling the **GetVolumeInformation** function and examining the FILE_SUPPORTS_REPARSE_POINTS bit flag.

You can set, modify, obtain, and remove reparse points by using the **DeviceIoControl** function. The following table describes the reparse point operations that you can perform using **DeviceIoControl**.

Operation	Description
FSCTL_SET_REPARSE_POINT	Allows the calling program to set a new reparse point, or to modify an existing one.
FSCTL_GET_REPARSE_POINT	Obtains the information stored in an existing reparse point.
FSCTL_DELETE_REPARSE_POINT	Removes an existing reparse point.

If you are modifying, getting, or deleting a reparse point, you must specify the same reparse tag in the operation that is contained in the file. Otherwise, the operation will fail with the error ERROR_REPARSE_ATTRIBUTE_CONFLICT.

To determine whether a file contains a reparse point, use the **GetFileAttributes** function. If the file has an associated reparse point, the FILE_ATTRIBUTE_REPARSE_POINT attribute is set.

To overwrite an existing reparse point without already having a handle to the file, call **CreateFile** with the FILE_OPEN_REPARSE_POINT flag. This flag allows you to open the file whether or not the corresponding file system filter is installed and working correctly.

Reparse Points and File Operations

Reparse points enable new behavior that you should be aware of when writing applications that manipulate files. For example, backup applications should specify BACKUP_REPARSE_DATA in the **WIN32_STREAM_ID** structure when backing up files with reparse points. Virus detection applications should search for reparse points that indicate linked files. Most applications should take special actions for files that have been moved to long-term storage, if only to notify the user that it may take a while to retrieve the file.

Reparse Point Restrictions

The following restrictions apply to reparse points:

- Reparse points can be established for a directory, but the directory must be empty. Otherwise, NTFS fails to establish the reparse point. In addition, you cannot create directories or files in a directory that contains a reparse point.
- Reparse points and extended attributes are mutually exclusive. NTFS cannot create a reparse point when the file contains extended attributes, and it cannot create extended attributes on a file that contains a reparse point.
- Reparse point data cannot exceed 16 kilobytes. Setting a reparse point fails if the amount of data to be placed in the reparse point exceeds this limit.

Volume Mount Points and Mounting Volumes

A *volume mount point* is a directory on a volume that an application can use to "mount" a different volume, that is, to set it up for use at the location a user specifies. In other words, you can use a volume mount point as a gateway to the volume. When a volume is mounted at a volume mount point, users and applications can refer to the mounted volume by the path of the volume mount point or a drive letter. For example, with a volume mount point set the user might refer to drive D as "C:\mnt\Ddrive" as well as "D:".

Using volume mount points, you can unify into one logical file system disparate file systems such as Windows 2000 NTFS, a 16-bit FAT file system, an ISO-9660 file system on a CD-ROM drive, and so on. Neither users nor applications need information about the volume on which a specific file resides. All the information they need to locate a specified file is a complete path. Volumes can be rearranged, substituted, or subdivided into many volumes without users or applications needing to change settings.

Your application can designate any directory on a volume other than the root as a volume mount point. Instead of making the root directory a volume mount point, your application should create a subdirectory and make that a volume mount point.

When a volume is mounted, volume mount points refer to the drive, not the medium in the drive. Thus, you need not unmount and mount volumes to change removable media.

You can set multiple volume mount points to refer to the same physical drive. For example, you might have a volume mount point for each of several compact discs, all of which share one CD-ROM drive. Each point refers to the same physical CD-ROM drive,

and you can access any disc in the drive from any of the volume mount points. The advantage of setting multiple volume mount points is that the different paths make clear the difference between different CDs' applications, for example between a word processor and a game.

To give a brief example of the use of volume mount points, suppose you have a computer with four volumes on it. You have two partitions on a single hard drive, a CD-ROM drive, and a removable media drive. In a conventional system, you might refer to these volumes as C, D, E, and F, respectively.

To use volume mount points, you might build a directory on C: called \mnt:

```
mkdir c:\mnt
```

Below that, you might build directories Ddrive, CDROM, and removeable:

```
mkdir c:\mnt\Ddrive
mkdir c:\mnt\CDROM
mkdir c:\mnt\removeable
```

Then you might mount each volume at its volume mount point, using a hypothetical command-line utility called Mountvolume:

```
mountvolume c:\mnt\Ddrive d:
mountvolume c:\mnt\CDROM e:
mountvolume c:\mnt\removeable f:
```

At this point, you can refer to the root directory of the removeable drive as either F:\ or as C:\mnt\removeable. You can also refer to files on the mounted volume with a concatenated path. Thus, you can refer to the file "D:\Program Files\Windows NT\Accessories\Wordpad.exe" as "C:\Mnt\Ddrive\Program Files\Windows NT\Accessories\Wordpad.exe".

Once the volume mount points have been established, they are maintained through computer restarts automatically.

If a volume fails, paths that cross the failed drive break. Thus, it is best to mount all volumes to the boot drive (usually drive C). It is a useful mnemonic for system administrators to use one directory as the volume mount point for all volumes on the system, but nothing in the volume mount point design requires that you use only one directory.

A volume mount point is a directory where a volume *can be* mounted but not necessarily where a volume currently *is* mounted.

NTFS volume mount points are implemented by using reparse points and are subject to their restrictions. For more information, see *Reparse Points*. Volume mount points are supported by Windows 2000 NTFS and higher. It is not necessary to manipulate reparse points in order to use volume mount points; the volume mount point functions handle all the reparse point details for you.

Because volume mount points are directories, you can rename, remove, move, and otherwise manipulate them, just as you can with directories.

Volume mount points are available only in Windows 2000. Only NTFS volumes can hold a volume mount point, although any local drive can be mounted on one.

Unique Volume Names

Two factors can make it hard to reliably mount a specific volume at a specified volume mount point across operating system restarts. One factor is that two different volumes can have the same label, which makes them indistinguishable except by drive letter. The other factor is that drive letters do not necessarily remain the same. If a computer's administrator does not use the Disk Administrator to enforce drive letters, then drive letters can change as drives are removed from or added to the system.

To solve this problem, the system refers to volumes to be mounted with unique volume names. These are strings of this form:

"\\?\Volume{*GUID*}\"

where *GUID* is a globally unique identifier (GUID) that identifies the volume. The **\\?** turns off path parsing and is ignored as part of the path, as discussed in *Path Lengths*. Note the trailing backslash. All volume mount point functions that take a unique volume name as a parameter require the trailing backslash; all volume mount point functions that return a unique volume name provide the trailing backslash. You can use **CreateFile** to open a volume by referring to its unique volume name, but without a trailing backslash. When using **CreateFile**, a unique volume name with a backslash refers to the root directory of the volume.

The operating system assigns a unique volume name to a volume when the computer first encounters it, for example during formatting or installation. The volume mount point functions use unique volume names to refer to volumes. To learn the unique volume name of any drive, use the **GetVolumeNameForVolumeMountPoint** function.

Path Lengths

Path lengths may be a concern when a volume with a deep directory tree is mounted to another volume. This is because paths are concatenated by mounting. The path of a file on a mounted volume thus includes the path of the volume mount point. The globally defined constant MAX_PATH defines the maximum number of characters a path can have. You can avoid this constraint by doing both of the following:

- Referring to volumes by their unique volume names, which have **\\?** prepended to the path.
- Using Unicode so that you use the Unicode (W) versions of file functions, which support the **\\?** prefix.

The **\\?** turns off path parsing. By using this form, you can work with paths that are nearly 32,000 Unicode characters long. However, each component in the path cannot be more than a file-system-specific value indicated by the function **GetVolumeInformation**.

You must use full paths with this technique. This technique also works with universal naming convention (UNC) names such as "\\OtherComputer\Directory\Filename.ext".

The **\\?** is ignored as part of the path example, and "\\?\C:\myworld\private" is seen as "C:\myworld\private".

Mounting a Volume

Mounting a volume is a two-step process. First, call **GetVolumeNameForVolumeMountPoint** with the DOS drive letter of the volume you want to mount. Then, use the returned unique volume name and the directory where you want to mount the volume in a call to **SetVolumeMountPoint**. See the example program mount.c.

Enumerating Volumes

To make a complete list of the volumes on a computer, or to manipulate each volume in turn, you can enumerate volumes. Within a volume, you can scan for volume mount points or other objects on the volume.

Three functions allow an application to enumerate volumes on a computer:

- **FindFirstVolume**
- **FindNextVolume**
- **FindVolumeClose**

These three functions operate in a manner very similar to the **FindFirstFile**, **FindNextFile**, and **FindClose** functions.

Begin a search for volumes with **FindFirstVolume**. If the search is successful, process the results according to the design of your application. Then use **FindNextVolume** in a loop to locate and process each subsequent volume. When the supply of volumes is exhausted, close the search with **FindVolumeClose**.

You should not assume any correlation between the order of volumes returned with these functions and the order of volumes returned by other tools. In particular, do not assume any correlation between volume order and drive letters as assigned by the BIOS (if any) or the Disk Administrator.

See *Volume Mount Point Examples* for an example of how to enumerate the volumes on a computer.

Scanning Volume Mount Points on a Volume

To enumerate all of the volume mount points on a volume, or to manipulate each in turn, scan a volume for volume mount points. Three functions allow an application to enumerate the volume mount points on a specified NTFS volume:

- **FindFirstVolumeMountPoint**
- **FindNextVolumeMountPoint**

- **FindVolumeMountPointClose**

These three functions operate in a manner very similar to the **FindFirstFile**, **FindNextFile**, and **FindClose** functions.

To enumerate volume mount points on a volume, first find out if the volume is a Windows 2000 NTFS volume and thus supports volume mount points. To do so, use the volume name returned by the **FindFirstVolume** and **FindNextVolume** functions to call the **GetVolumeInformation** function. The names returned include a trailing backslash (\) to be compatible with the **GetDriveType** function and related functions. For more information on the functions used to scan the volumes on a computer, see *Enumerating Volumes*.

If the volume is an NTFS volume and supports volume mount points, begin a search for the volume's volume mount points with **FindFirstVolumeMountPoint**. If the search is successful, process the results according to your application's requirements. Then use **FindNextVolumeMountPoint** in a loop to locate and process each subsequent volume mount point. When the supply of volume mount points is exhausted, close the process with **FindVolumeMountPointClose**.

Volume mount point searches are confined to the specified volume. To search all of the volume mount points on a computer, use the volume enumerating functions to scan all the volumes, and search each volume in turn. For more information, see *Enumerating Volumes*.

You should not assume any correlation between the order of volume mount points returned by these functions and the order of volume mount points returned by other tools.

See *Volume Mount Point Examples* for an example of how to enumerate the mount points on a volume.

Checking Directories for Volume Mount Points

It is useful to determine if a directory is a volume mount point when, for example, you are using a backup utility or search utility that is constrained to one volume. Such a utility can reach information on multiple volumes if you mount all volumes to the one the utility addresses.

To determine if a specified directory is a volume mount point, first call the **GetFileAttributes** function and inspect the FILE_ATTRIBUTE_REPARSE_POINT flag in the return value to see if the directory has an associated reparse point. If it does, use the **FindFirstFile** and **FindNextFile** functions to obtain the reparse tag. To determine if the reparse point is a volume mount point (and not some other form of reparse point), test whether the tag value equals the value IO_REPARSE_TAG_MOUNT_POINT. For more information, see *Reparse Points*.

To obtain the target volume of a volume mount point, use the function **GetVolumeNameForVolumeMountPoint**.

Persistent Assignment of Drive Letters

You can assign a drive letter (for example, x:\) to a local volume using **SetVolumeMountPoint**, provided there is no volume already assigned to that drive letter. If the local volume already has a drive letter then **SetVolumeMountPoint** will fail. To handle this, first delete the drive letter using **DeleteVolumeMountPoint**.

Windows 2000 allows at most one drive letter per volume, so you cannot have C:\ and F:\ pointing to the same volume.

Caution Deleting an existing drive letter and assigning a new one may break existing paths, such as those in desktop shortcuts. It may also break the path to the program making the drive letter changes. With Windows 2000's virtual memory management, this may break the application, leaving the system in an unstable and possibly unusable state. It is the program designer's responsibility to avoid such potential catastrophes.

Volume Mount Point Reference

The volume mount point functions can be divided into three groups: general-purpose functions, functions used to scan for volumes, and functions used to scan a volume for volume mount points.

General-Purpose Volume Mount Point Functions

Function	Description
DeleteVolumeMountPoint	Unmounts a volume from the specified volume mount point.
GetVolumeNameForVolumeMountPoint	Returns the unique volume name for a specified volume mount point or root directory.
GetVolumePathName	Returns the volume mount point at which the specified path is mounted.
SetVolumeMountPoint	Mounts the specified volume at the specified volume mount point.

Volume Mount Point Volume-Scanning Functions

Function	Description
FindFirstVolume	Returns the name of a volume on a computer. **FindFirstVolume** is used to begin enumerating the volumes of a computer.
FindNextVolume	Continues a volume search started by a call to **FindFirstVolume**.
FindVolumeClose	Closes a search for volumes.

Volume Mount Point Scanning Functions

Function	Description
FindFirstVolumeMountPoint	Returns the name of a volume mount point on the specified volume. **FindFirstVolumeMountPoint** is used to begin scanning the volume mount points on a volume.
FindNextVolumeMountPoint	Continues a volume mount point search started by a call to **FindFirstVolumeMountPoint**.
FindVolumeMountPointClose	Closes a search for volume mount points.

NTFS Change Journal

The NTFS file system maintains a log or change journal that records changes to files. NTFS maintains the change journal in order to recover file system indexing, for example after a computer or volume failure. The ability to recover indexing means the file system can avoid the time-consuming process of reindexing the whole volume in such cases. The change journal software is available only with the Microsoft Windows 2000 operating system.

The change journal provides support for any service that tracks changes to a volume. Such services can include indexing packages as well as storage management software.

As files, directories, and other NTFS objects are added, deleted, and modified, NTFS enters records into the change journal in streams, one for each volume on the computer. Each record indicates the type of change and the object changed. The offset from the beginning of the stream for a particular record is called the update sequence number (USN) for the particular record. New records are appended to the end of the stream.

NTFS may delete old records in order to conserve space. If needed records have been deleted, the indexing service recovers by reindexing the volume, as it does when no change journal exists.

The change journal logs only the fact of a change to a file and the reason for the change (for example, write operations, truncation, lengthening, deletion, and so on). It does not record enough information to allow reversing the change.

In addition, multiple changes to the same file may result in only one reason flag being added to the current record. If the same kind of change occurs more than once, NTFS does not write a new record for the changes after the first. For example, several write operations with no intervening close and reopen operations result in only one change record with the reason flag USN_REASON_DATA_OVERWRITE set.

To illustrate how the change journal works, suppose a user accesses a file in the following fashion:

1. Writes to the file.
2. Sets the time stamp for the file.

3. Writes to the file.
4. Truncates the file.
5. Writes to the file.
6. Closes the file.

In this case, NTFS takes the following actions in the change journal (where I indicates a bitwise OR operation).

Event	NTFS action
Initial write operation	NTFS writes a new USN record with the USN_REASON_DATA_OVERWRITE reason flag set. For more information on possible reason flags, see the **USN_RECORD** structure topic.
Setting of file time stamp	NTFS writes a new USN record with the flag setting USN_REASON_DATA_OVERWRITE I USN_REASON_BASIC_INFO_CHANGE.
Second write operation	NTFS does not write a new USN record. Because USN_REASON_DATA_OVERWRITE is already set for the existing record, no changes are made to the record.
File truncation	NTFS writes a new USN record with the flag setting USN_REASON_DATA_OVERWRITE I USN_REASON_BASIC_INFO_CHANGE I USN_REASON_DATA_TRUNCATION.
Third write operation	NTFS does not write a new USN record. Because USN_REASON_DATA_OVERWRITE is already set for the existing record, no changes are made to the record.
Close operation	If the user making changes is the only user of the file, NTFS writes a new USN record with the following flag setting: USN_REASON_DATA_OVERWRITE I USN_REASON_BASIC_INFO_CHANGE I USN_REASON_DATA_TRUNCATION I USN_REASON_CLOSE.

The change journal accumulates a series of records between the first opening and last closing of a file. Each record has a new reason flag set, indicating that a new kind of change has occurred. The sequence of records gives a partial history of the file. The final record, created when the file is closed, adds the USN_REASON_CLOSE flag. This record represents a summary of changes to the file, but unlike the prior records, gives no indication of the order of the changes.

The next user to access and change the file generates a new USN record with a single reason flag.

Using the Change Journal Identifier

NTFS associates an unsigned 64-bit identifier with each change journal. The journal is stamped with this identifier when it is created. The file system restamps the journal with a new identifier where the existing USN records either are or may be unusable.

For example, NTFS restamps a change journal with a new identifier when a volume is moved from Windows 2000 to Microsoft Windows NT version 4.0 and then back to Windows 2000. Such a move can happen in a dual-boot environment or when working with removable media.

To obtain the identifier of the current change journal on a specified volume, use the **FSCTL_QUERY_USN_JOURNAL** operation for the **DeviceIoControl** function. To perform this and all other change journal operations, you must have system administrator privileges. That is, you must be a member of the Administrators group.

When an administrator deletes and recreates the change journal, for example when the current USN value approaches the maximum possible USN value, the USN values begin again from zero. When NTFS stamps a journal with a new identifier rather than recreating the journal, it does not reset the USN to zero but continues from the current USN. In either case, all existing USNs are less than any future USNs.

When you need information on a specific set of records, use the **DeviceIoControl** operation **FSCTL_QUERY_USN_JOURNAL** to obtain the change journal identifier. Then use the **FSCTL_READ_USN_JOURNAL** operation to read the journal records of interest. NTFS only returns records that are valid for the journal specified by the identifier.

Your application needs both the records' USNs and the identifier to read the journal. This requirement provides an integrity check for cases where your application should ignore the existing records in the file and where records were written in previous instances of the journal for the same volume.

To obtain the records in which you are interested, you must start at the oldest record (that is, with the lowest USN) and scan forward until you locate the first record of interest.

Note In order for a Windows NT 4.0 system to read an NTFS file system in Windows 2000, either you should upgrade the Windows NT 4.0 system to Windows NT 4.0 Service Pack 4 or higher, or install Windows 2000 on the same computer. Installation of Windows 2000 beta 2 or higher upgrades Windows NT 4.0 to read and write Windows 2000 volumes and upgrades all volumes to Windows 2000 NTFS.

The Windows NT 4.0 drivers for NTFS volumes in Windows 2000 do not maintain the change journal, and Windows NT 4.0 does not allow access to the change journal.

Creating, Modifying, and Deleting a Change Journal

Administrators can use the Windows 2000 user interface to create, delete, and recreate change journals at will. An administrator should delete a journal when the current USN value approaches the maximum possible USN value, as indicated by the **MaxUsn** member of the **USN_JOURNAL_DATA** structure. An administrator might also delete and recreate a change journal to reclaim disk space. To perform this and all other nonprogrammatic change journal operations, you must have system administrator privileges. That is, you must be a member of the Administrators group.

To create or modify a change journal on a specified volume programmatically, use the **FSCTL_CREATE_USN_JOURNAL** operation with the function **DeviceIoControl**.

When you create a new change journal or modify an existing one, NTFS sets information for that change journal from information in the **CREATE_USN_JOURNAL_DATA** structure, which **FSCTL_CREATE_USN_JOURNAL** takes as input. **CREATE_USN_JOURNAL_DATA** has the members **MaximumSize** and **AllocationDelta**.

MaximumSize is the target maximum size for the change journal in bytes. The change journal can grow larger than this value, but at NTFS checkpoints NFTS examines the journal and trims it when its size exceeds the value of **MaximumSize** plus the value of **AllocationDelta**. (At *NTFS checkpoints,* the operating system writes records that allow NTFS to determine what processing is required to recover from a failure.)

AllocationDelta is the number of bytes of disk memory added to the end and removed from the beginning of the change journal each time memory is allocated or deallocated. In other words, allocation and deallocation take place in units of this size. An integer multiple of a cluster size is a reasonable value for this member.

If an administrator modifies an existing change journal to have a larger **MaximumSize** value, for example if a volume is being reindexed too often, the change journal simply receives new entries until it exceeds the new maximum size. If the new maximum size is smaller than the old one, then at the next NTFS checkpoint, NTFS deletes enough old entries to truncate the change journal to a size less than the new maximum size.

To delete a change journal, use the **FSCTL_DELETE_USN_JOURNAL** operation with **DeviceIoControl**. When you use this operation, the operation walks through all of the files on the volume and resets the USN for each file to zero. The operation then deletes the existing change journal. This operation persists across system restarts until it completes. Any attempt to read, create, or modify the change journal during this process fails with the error code ERROR_JOURNAL_DELETE_IN_PROGRESS.

You can also use the **FSCTL_DELETE_USN_JOURNAL** operation to determine if a deletion started by some other process is in progress. For example, your application, when it is started, can determine if a deletion is in progress. Because journal deletions persist across system restarts, services and applications started at system restart should check for an ongoing deletion.

Change journals do not necessarily run automatically at startup. In order to start a change journal, an administrator may do so explicitly or start another service that requires a change journal.

Obtaining a Volume Handle for Change Journal Operations

To obtain a handle to a volume for use with change journal operations, pass in the *lpFileName* parameter of the **CreateFile** function a value of the following form:

```
\\.\X:
```

where *X* is the letter identifying the drive on which the volume appears. The volume must be an NTFS volume on Windows 2000.

Walking a Buffer of Change Journal Records

The two operations for the **DeviceIoControl** function that return change journal records, **FSCTL_READ_USN_JOURNAL** and **FSCTL_ENUM_USN_DATA**, return almost the same data. Both return zero or more change journal records, each in a **USN_RECORD** structure. Use **FSCTL_ENUM_USN_DATA** when you want a listing (enumeration) of all change journal records between two USNs. Use **FSCTL_READ_USN_JOURNAL** when you want to be more selective, such as selecting specific reasons for changes or returning when a file is closed.

USN_RECORD contains the name of the file to which the record in question applies. The file name varies in length, so **USN_RECORD** is a variable-length structure. Its first member, **RecordLength**, is the length of the structure (including the file name) in bytes. Returned **USN_RECORD** structures are aligned on 64-bit boundaries.

Both of these operations return only the subset of change journal records between the boundaries specified. These operations also return the next record number to be retrieved, so that you can continue reading records from the end boundary forward.

To walk the buffer of change journal records returned by either operation from the first entry onward, you must round up the length of the buffer to match the alignment of the **USN_RECORD** structures.

To simplify matters, when you read records from a change journal declare your input buffer on a 64-bit boundary. That way, the first record always occurs at the boundary equal to the value of the **DeviceIoControl** *lpOutBuffer* parameter plus 8.

The size in bytes of any record specified by a **USN_RECORD** structure is at most `((MaximumComponentLength - 1) * Width) + Size` where *MaximumComponentLength* is the maximum length, in characters, of the record's file name, *Width* is the size of a wide character, and *Size* is the size of the structure. To obtain this maximum length, call the **GetVolumeInformation** function and examine the value pointed to by the *lpMaximumComponentLength* parameter. You subtract one from *MaximumComponentLength* to account for the fact that the definition of **USN_RECORD** includes one character of the file name.

Thus, in C the largest possible record size is:

```
(MaximumComponentLength * sizeof(WCHAR) + sizeof(USN_RECORD) - sizeof(WCHAR))
```

When working with the **FileName** member of **USN_RECORD**, do not count on the change journal's file name containing a trailing '\0' delimiter. To determine the length of the file name, use the **FileNameLength** member.

Change Journal Operations

The following **DeviceIoControl** function operations work with the NTFS change journal.

FSCTL_CREATE_USN_JOURNAL
FSCTL_DELETE_USN_JOURNAL
FSCTL_ENUM_USN_DATA
FSCTL_MARK_HANDLE
FSCTL_QUERY_USN_JOURNAL
FSCTL_READ_USN_JOURNAL

Change Journal Structures

The following structures hold information relating to the NTFS change journal.

CREATE_USN_JOURNAL_DATA
DELETE_USN_JOURNAL_DATA
MARK_HANDLE_INFO
MFT_ENUM_DATA
READ_USN_JOURNAL_DATA
USN_JOURNAL_DATA
USN_RECORD

FAT File System

The File Allocation Table (FAT) file system organizes data on fixed disks and floppy disks. The distinguishing feature of the FAT file system is its file name convention. The file name convention consists of a file name (up to eight characters), a separating period (.), and a file name extension (up to three characters).

The main advantage of FAT volumes is that they are accessible by MS-DOS, Microsoft Windows, and OS/2 systems. FAT is also the only file system currently supported on floppy disks and other removable media.

Valid FAT file names have the following form:

[[*drive*:]][[*directory*\]]*filename*[[.*extension*]]

The *drive* parameter must name an existing drive and can be any letter from A through Z. The drive letter must be followed by a colon (:).

The *directory* parameter specifies the directory that contains the file's directory entry. This value must be followed by a backslash (\) to separate it from the file name. If the specified directory is not in the current directory, *directory* must include the names of all

directories in the file's path, separated by backslashes. The root directory is specified by using a backslash at the beginning of the name. For example, if the directory ABCD is in the directory SAMPLE and SAMPLE is in the root directory, the correct directory specification is \SAMPLE\ABCD. A directory name consists of any combination of up to eight letters, digits, or the following special characters:

$ % ' – _ @ { } ~ ` ! # ( )

A directory name can also have an extension that is any combination of up to three letters, digits, or special characters, preceded by a period (.).

The *filename* and *extension* parameters specify the file. *Filename* can be any combination of up to eight letters, digits, or the special characters previously listed; *extension* can be any combination of up to three letters, digits, or special characters, all preceded by a period. *Filename* can also include embedded (preceded and followed by one or more letters, digits, or special characters just noted) spaces. For example, the string "disk 1" is a valid value for *filename*.

FAT volumes do not distinguish between uppercase and lowercase letters.

Protected-Mode FAT File System

The protected-mode FAT file system organizes data on fixed and floppy disks. Protected-mode FAT is compatible with the FAT file system, using file allocation tables and directory entries to store information about the contents of a disk. Protected-mode FAT also supports long file names, storing these names and other information such as the date and time the file was last accessed in the FAT structures.

Protected-mode FAT allows file names of up to 255 characters, including the terminating null character. This is similar to NTFS, which allows file names of up to 256 characters.

Protected-mode FAT allows paths of up to 260 characters, including the terminating null character.

File System Reference

File System Functions

AddUsersToEncryptedFile

The **AddUsersToEncryptedFile** function adds user keys to a specified encrypted file.

```
DWORD AddUsersToEncryptedFile(
  LPCWSTR lpFileName,                   // file name
  PENCRYPTION_CERTIFICATE_LIST pUsers  // user keys
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated Unicode string that specifies the name of the encrypted file.

pUsers

[in] Pointer to a **ENCRYPTION_CERTIFICATE_LIST** structure that contains the list of new user keys to be added to the file.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a Win32 error code. For a complete list of error codes, see *Error Codes* or the Platform SDK header file WinError.h.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.

See Also

File Systems Overview, *File System Functions*, **ENCRYPTION_CERTIFICATE_LIST**

CreateHardLink

The **CreateHardLink** function establishes an NTFS hard link between an existing file and a new file. An NTFS hard link is similar to a POSIX hard link.

```
BOOL CreateHardLink(
  LPCTSTR lpFileName,                          // new file name
  LPCTSTR lpExistingFileName,                  // extant file name
  LPSECURITY_ATTRIBUTES lpSecurityAttributes  // SD
);
```

Parameters

lpFileName

[in] Pointer to the name of the new directory entry to be created.

lpExistingFileName

[in] Pointer to the name of the existing file to which the new link will point.

lpSecurityAttributes

[in] Pointer to a **SECURITY_ATTRIBUTES** structure that specifies a security descriptor for the new file.

If this parameter is NULL, it leaves the file's existing security descriptor unmodified.

If this parameter is not NULL, it modifies the file's security descriptor.

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

Any directory entry for a file, whether created with **CreateFile** or **CreateHardLink**, is a hard link to the associated file. Additional hard links, created with the **CreateHardLink** function, allow you to have multiple directory entries for a file, that is, multiple hard links to the same file. These may be different names in the same directory, or they may be the same (or different) names in different directories. However, all hard links to a file must be on the same volume.

Because hard links are just directory entries for a file, whenever an application modifies a file through any hard link, all applications using any other hard link to the file see the changes. Also, all of the directory entries are updated if the file changes. For example, if the file's size changes, all of the hard links to the file will show the new size.

The security descriptor belongs to the file to which the hard link points. The link itself, being merely a directory entry, has no security descriptor. Thus, if you change the security descriptor of any hard link, you're actually changing the underlying file's security descriptor. All hard links that point to the file will thus allow the newly specified access. There is no way to give a file different security descriptors on a per-hard-link basis.

Use **DeleteFile** to delete hard links. You can delete them in any order regardless of the order in which they were created.

Flags, attributes, access, and sharing as specified in **CreateFile** operate on a per-file basis. That is, if you open a file with no sharing allowed, another application cannot share the file by creating a new hard link to the file.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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.

File Systems Overview, *File System Functions*, **CreateFile**, **DeleteFile**, **SECURITY_ATTRIBUTES**

DecryptFile

The **DecryptFile** function decrypts an encrypted file or directory.

```
BOOL DecryptFile(
  LPCTSTR lpFileName,   // file name
  DWORD dwReserved      // reserved; must be zero
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated Unicode string that specifies the name of the file to decrypt.

The caller must have FILE_READ_DATA, FILE_WRITE_DATA, FILE_READ_ATTRIBUTES, FILE_WRITE_ATTRIBUTES, and SYNCHRONIZE access to the file.

dwReserved

Reserved; must be zero.

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 **DecryptFile** function requires exclusive access to the file being decrypted, and will fail if another process is using the file. If the file is not encrypted, **DecryptFile** simply returns a nonzero value, which indicates success.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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.

File Systems Overview, *File System Functions*, **CreateFile**, **EncryptFile**

DeleteVolumeMountPoint

The **DeleteVolumeMountPoint** function unmounts the volume from the specified volume mount point.

```
BOOL DeleteVolumeMountPoint(
  LPCTSTR lpszVolumeMountPoint  // volume mount point path
);
```

Parameters

lpszVolumeMountPoint

[in] Pointer to a string that indicates the volume mount point to be unmounted. This may be a root directory (X:\, in which case the DOS drive letter assignment is removed) or a directory on a volume (X:\mnt\). A trailing backslash is 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

It is not an error to attempt to unmount a volume from a volume mount point when there is no volume actually mounted at that volume mount point.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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.

File Systems Overview, *File System Functions*, **GetVolumeNameForVolumeMountPoint**, **GetVolumePathName**, **SetVolumeMountPoint**

EncryptFile

The **EncryptFile** function encrypts a file or directory. All data streams in a file are encrypted. All new files created in an encrypted directory are encrypted.

```
BOOL EncryptFile(
  LPCTSTR lpFileName  // file name
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the file or directory to encrypt.

The caller must have FILE_READ_DATA, FILE_WRITE_DATA, FILE_READ_ATTRIBUTES, FILE_WRITE_ATTRIBUTES, and SYNCHRONIZE access to the file.

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 **EncryptFile** function requires exclusive access to the file being encrypted, and will fail if another process is using the file.

If the file is already encrypted, **EncryptFile** simply returns a nonzero value, which indicates success. If the file is compressed, **EncryptFile** will decompress the file before encrypting it.

To decrypt an encrypted file, use the **DecryptFile** function.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **DecryptFile**

EncryptionDisable

The **EncryptionDisable** function disables or enables encryption of the indicated directory and the files in it. It does not affect encryption of subdirectories below the indicated directory.

```
BOOL EncryptionDisable(
    LPCWSTR DirPath,   // directory name
    BOOL Disable       // encryption option
);
```

Parameters

DirPath

[in] Pointer to a null-terminated Unicode string that specifies the name of the directory for which to enable or disable encryption.

Disable

[in] Indicates whether to disable encryption (TRUE) or enable it (FALSE).

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

Under normal circumstances, **EncryptFile** will not encrypt files and directories with the FILE_ATTRIBUTE_SYSTEM attribute set. It is possible to override the FILE_ATTRIBUTE_SYSTEM attribute and encrypt files. Also, if a file or directory is marked with the FILE_ATTRIBUTE_SYSTEM attribute, it will normally be invisible to the user in directory listings and Windows Explorer directory windows. **EncryptionDisable** disables encryption of directories and files. It does not affect the visibility of files with the FILE_ATTRIBUTE_SYSTEM attribute set.

If TRUE is passed in, **EncryptionDisable** will write

```
[Encryption]
Disable=1
```

to the Desktop.ini file in the directory (creating it if necessary). If the section already exists but *Disable* is set to 0, it will be set to 1.

Thereafter, **EncryptFile** will fail on the directory and the files in it, and the code that **GetLastError** returns will be FILE_DIR_DISALLOWED. This function does not affect encryption of subdirectories within the given directory.

The user can also manually add or edit the above lines in the Desktop.ini file and produce the same effect.

EncryptionDisable affects only **FileEncryptionStatus** and **EncryptFile**. Once the directory is encrypted, any new files and new subdirectories created without the FILE_ATTRIBUTE_SYSTEM attribute will be encrypted.

If FALSE is passed in, **EncryptionDisable** will write

```
[Encryption]
Disable=0
```

and file encryption is permitted on the files in that directory.

If you try to use **EncryptionDisable** to set the directory to the state it is already in, the function succeeds but has no effect.

If you try to use **EncryptionDisable** to disable or enable encryption on a file, the attempt will fail.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

See Also

File Systems Overview, *File System Functions*, **DecryptFile**, **EncryptFile**, **FileEncryptionStatus**, **GetFileAttributes**

FileEncryptionStatus

The **FileEncryptionStatus** function retrieves the encryption status of the specified file.

```
BOOL FileEncryptionStatus(
  LPCTSTR lpFileName,  // file name
  LPDWORD lpStatus     // encryption status
);
```

Parameters

lpFileName
[in] Pointer to a null-terminated string that specifies the name of the file.

lpStatus
[out] Pointer to a variable that receives the encryption status of the file. This parameter can be one of the following values:

Value	Meaning
FILE_ENCRYPTABLE	The file can be encrypted.
FILE_IS_ENCRYPTED	The file is encrypted.
FILE_SYSTEM_ATTR	The file is a system file. System files cannot be encrypted.
FILE_ROOT_DIR	The file is a root directory. Root directories cannot be encrypted.
FILE_SYSTEM_DIR	The file is a system directory. System directories cannot be encrypted.
FILE_UNKNOWN	The encryption status is unknown. The file may be encrypted.
FILE_SYSTEM_NOT_SUPPORT	The file system does not support file encryption.
FILE_USER_DISALLOWED	Reserved for future use.
FILE_READ_ONLY	The file is a read-only file.

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: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use advapi32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

File Systems Overview, *File System Functions*, **EncryptFile**

FindFirstVolume

The **FindFirstVolume** function returns the name of a volume on a computer. **FindFirstVolume** is used to begin scanning the volumes of a computer.

```
HANDLE FindFirstVolume(
  LPTSTR lpszVolumeName,    // output buffer
  DWORD cchBufferLength     // size of output buffer
);
```

Parameters

lpszVolumeName

[out] Pointer to a buffer that receives the unique volume name of the first volume found.

cchBufferLength

[in] Length, in characters, of the buffer to receive the name.

Return Values

If the function succeeds, the return value is a search handle used in a subsequent call to the **FindNextVolume** and **FindVolumeClose** functions.

If the function fails to find any volumes, the return value is the INVALID_HANDLE_VALUE error code. To get extended error information, call **GetLastError**.

Remarks

The **FindFirstVolume** function opens a volume search handle and returns information about the first volume found on a computer. Once the search handle is established, you can use the **FindNextVolume** function to search for other volumes. When the search handle is no longer needed, close it by using the **FindVolumeClose** function.

You should not assume any correlation between the order of volumes returned with these functions and the order of volumes on the computer. In particular, do not assume any correlation between volume order and drive letters as assigned by the BIOS (if any) or the Disk Administrator.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **FindNextVolume**, **FindVolumeClose**

FindFirstVolumeMountPoint

The **FindFirstVolumeMountPoint** function returns the name of a volume mount point on the specified volume. **FindFirstVolumeMountPoint** is used to begin scanning the volume mount points on a volume.

```
HANDLE FindFirstVolumeMountPoint(
  LPTSTR lpszRootPathName,     // volume name
  LPTSTR lpszVolumeMountPoint, // output buffer
  DWORD cchBufferLength        // size of output buffer
);
```

Parameters

lpszRootPathName
[in] Unique volume name of the volume to scan for volume mount points. A trailing backslash is required.

lpszVolumeMountPoint
[out] Pointer to a buffer that receives the name of the first volume mount point found.

cchBufferLength
[in] Specifies the length, in characters, of the buffer that receives the volume mount point name.

Return Values

If the function succeeds, the return value is a search handle used in a subsequent call to the **FindNextVolumeMountPoint** and **FindVolumeMountPointClose** functions.

If the function fails to find a volume mount point on the volume, the return value is the INVALID_HANDLE_VALUE error code. To get extended error information, call **GetLastError**.

Remarks

The **FindFirstVolumeMountPoint** function opens a mount-point search handle and returns information about the first volume mount point found on the specified volume. Once the search handle is established, you can use the **FindNextVolumeMountPoint** function to search for other volume mount points. When the search handle is no longer needed, close it with the **FindVolumeMountPointClose** function.

You should not assume any correlation between the order of volume mount points returned by these functions and the order of volume mount points returned by other tools.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **FindNextVolumeMountPoint**, **FindVolumeMountPointClose**

FindNextVolume

The **FindNextVolume** function continues a volume search started by a call to the **FindFirstVolume** function. **FindNextVolume** finds one volume per call.

```
BOOL FindNextVolume(
  HANDLE hFindVolume,        // volume search handle
  LPTSTR lpszVolumeName,     // output buffer
  DWORD cchBufferLength      // size of output buffer
);
```

Parameters

hFindVolume
[in] Volume search handle returned by a previous call to the **FindFirstVolume** function.

lpszVolumeName
[out] Pointer to a string that receives the unique volume name found.

cchBufferLength
[in] Specifies the length, in characters, of the buffer that receives the name.

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**. If no matching files can be found, the **GetLastError** function returns the ERROR_NO_MORE_FILES error code. In that case, close the search with the **FindVolumeClose** function.

Remarks

Once the search handle is established by calling **FindFirstVolume**, you can use the **FindNextVolume** function to search for other volumes.

You should not assume any correlation between the order of volumes returned with these functions and the order of volumes on the computer. In particular, do not assume any correlation between volume order and drive letters as assigned by the BIOS (if any) or the Disk Administrator.

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **FindFirstVolume**, **FindVolumeClose**

FindNextVolumeMountPoint

The **FindNextVolumeMountPoint** function continues a volume mount point search started by a call to the **FindFirstVolumeMountPoint** function. **FindNextVolumeMountPoint** finds one volume mount point per call.

```
BOOL FindNextVolumeMountPoint(
  HANDLE hFindVolumeMountPoint,    // search handle
  LPTSTR lpszVolumeMountPoint,     // output buffer
  DWORD cchBufferLength            // size of output buffer
);
```

Parameters

hFindVolumeMountPoint
[in] Mount-point search handle returned by a previous call to the **FindFirstVolumeMountPoint** function.

lpszVolumeMountPoint
[out] Pointer to a string that receives the name of the volume mount point found.

cchBufferLength
[in] Specifies the length, in characters, of the buffer that receives the names.

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**. If no matching files can be found, the **GetLastError** function returns the ERROR_NO_MORE_FILES error code. In that case, close the search with the **FindVolumeMountPointClose** function.

Remarks

Once the search handle is established by calling **FindFirstVolumeMountPoint**, you can use the **FindNextVolumeMountPoint** function to search for other volume mount points.

You should not assume any correlation between the order of volume mount points returned with these functions and the order of volume mount points returned by other tools.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **FindFirstVolumeMountPoint**, **FindVolumeMountPointClose**

FindVolumeClose

The **FindVolumeClose** function closes the specified volume search handle. The **FindFirstVolume** and **FindNextVolume** functions use this search handle to locate volumes.

```
BOOL FindVolumeClose(
  HANDLE hFindVolume       // search handle
);
```

Parameters

hFindVolume
[in] Volume search handle to close. This handle must have been previously opened by the **FindFirstVolume** 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

After the **FindVolumeClose** function is called, the handle *hFindVolume* cannot be used in subsequent calls to either **FindNextVolume** or **FindVolumeClose**.

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

File Systems Overview, *File System Functions*, **FindFirstVolume**, **FindNextVolume**

FindVolumeMountPointClose

The **FindVolumeMountPointClose** function closes the specified mount-point search handle. The **FindFirstVolumeMountPoint** and **FindNextVolumeMountPoint** functions use this search handle to locate volume mount points on a specified volume.

```
BOOL FindVolumeMountPointClose(
  HANDLE hFindVolumeMountPoint // search handle
);
```

Parameters

hFindVolumeMountPoint

[in] Mount-point search handle to close. This handle must have been previously opened by the **FindFirstVolumeMountPoint** 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

After the **FindVolumeMountPointClose** function is called, the handle *hFindVolumeMountPoint* cannot be used in subsequent calls to either **FindNextVolumeMountPoint** or **FindVolumeMountPointClose**.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

File Systems Overview, *File System Functions*, **FindFirstVolumeMountPoint**, **FindNextVolumeMountPoint**

FreeEncryptionCertificateHashList

The **FreeEncryptionCertificateHashList** function frees a certificate hash list.

```
VOID FreeEncryptionCertificateHashList(
  PENCRYPTION_CERTIFICATE_HASH_LIST pHashes  // hash list
);
```

Parameters

pHashes

[in] Pointer to a certificate hash list structure, **ENCRYPTION_CERTIFICATE_HASH_LIST**, which was returned by the **QueryUsersOnEncryptedFile** or **QueryRecoveryAgentsOnEncryptedFile** function.

Return Values

This function does not return a value.

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.

See Also

File Systems Overview, *File System Functions*, **ENCRYPTION_CERTIFICATE_HASH_LIST**, **QueryRecoveryAgentsOnEncryptedFile**, **QueryUsersOnEncryptedFile**

GetCompressedFileSize

The **GetCompressedFileSize** function obtains the actual number of bytes of disk storage used to store a specified file. If the file is located on a volume that supports compression, and the file is compressed, the value obtained is the compressed size of the specified file. If the file is located on a volume that supports sparse files, and the file is a sparse file, the value obtained is the sparse size of the specified file.

If the file is not located on a volume that supports compression or sparse files, or if the file is not compressed or a sparse file, the value obtained is the actual file size, the same as the value returned by a call to **GetFileSize**.

```
DWORD GetCompressedFileSize(
  LPCTSTR lpFileName,      // file name
  LPDWORD lpFileSizeHigh   // high-order DWORD of file size
);
```

Parameters

lpFileName

[in] Pointer to a null-terminated string that specifies the name of the file.

lpFileSizeHigh

[out] Pointer to a **DWORD** variable that receives the high-order doubleword of the compressed file size. The function's return value is the low-order doubleword of the compressed file size.

This parameter can be NULL if the high-order doubleword of the compressed file size is not needed. Files less than 4 gigabytes in size do not need the high-order doubleword.

Return Values

If the function succeeds, the return value is the low-order doubleword of the actual number of bytes of disk storage used to store the specified file, and if *lpFileSizeHigh* is non-NULL, the function puts the high-order doubleword of that actual value into the **DWORD** pointed to by that parameter. This is the compressed file size for compressed files, the actual file size for noncompressed files.

If the function fails, and *lpFileSizeHigh* is NULL, the return value is –1. To get extended error information, call **GetLastError**.

If the function fails, and *lpFileSizeHigh* is non-NULL, the return value is –1, and **GetLastError** returns a value other than NO_ERROR.

Remarks

Calling the **GetCompressedFileSize** function with the name of a nonseeking device, such as a pipe or a communications device, has no meaning.

Note that if the return value is –1 and *lpFileSizeHigh* is non-NULL, an application must call **GetLastError** to determine whether the function has succeeded or failed.

An application can determine whether a volume is compressed by calling **GetVolumeInformation**, then checking the status of the FS_VOL_IS_COMPRESSED flag in the **DWORD** value pointed to by that function's *lpFileSystemFlags* parameter.

An application can determine whether a file is compressed by implementing the following pseudocode:

```
call GetVolumeInformation on the file's volume
if the file's volume is compressed
    call GetCompressedFileSize on the file
    call GetFileSize on the file
    if the sizes don't match
        the file is compressed
```

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **GetFileSize**, **GetVolumeInformation**

GetVolumeInformation

The **GetVolumeInformation** function returns information about a file system and volume whose root directory is specified.

```
BOOL GetVolumeInformation(
  LPCTSTR lpRootPathName,           // root directory
  LPTSTR lpVolumeNameBuffer,        // volume name buffer
  DWORD nVolumeNameSize,            // length of name buffer
  LPDWORD lpVolumeSerialNumber,     // volume serial number
  LPDWORD lpMaximumComponentLength,  // maximum file name length
  LPDWORD lpFileSystemFlags,        // file system options
  LPTSTR lpFileSystemNameBuffer,    // file system name buffer
  DWORD nFileSystemNameSize         // length of file system name buffer
);
```

Parameters

lpRootPathName

[in] Pointer to a string that contains the root directory of the volume to be described. If this parameter is NULL, the root of the current directory is used. A trailing backslash is required. For example, you would specify \\MyServer\MyShare as \\MyServer\MyShare\, or the C drive as "C:\".

lpVolumeNameBuffer

[out] Pointer to a buffer that receives the name of the specified volume.

nVolumeNameSize
[in] Specifies the length, in characters, of the volume name buffer. This parameter is ignored if the volume name buffer is not supplied.

lpVolumeSerialNumber
[out] Pointer to a variable that receives the volume serial number. This parameter can be NULL if the serial number is not required.

Windows 95/98: If the queried volume is a network drive, the serial number will not be returned.

lpMaximumComponentLength
[out] Pointer to a variable that receives the maximum length, in characters, of a file name component supported by the specified file system. A file name component is that portion of a file name between backslashes.

The value stored in variable pointed to by **lpMaximumComponentLength* is used to indicate that long names are supported by the specified file system. For example, for a FAT file system supporting long names, the function stores the value 255, rather than the previous 8.3 indicator. Long names can also be supported on systems that use the NTFS file system.

lpFileSystemFlags
[out] Pointer to a variable that receives flags associated with the specified file system. This parameter can be any combination of the following flags; however, FS_FILE_COMPRESSION and FS_VOL_IS_COMPRESSED are mutually exclusive.

Value	Meaning
FS_CASE_IS_PRESERVED	The file system preserves the case of file names when it places a name on disk.
FS_CASE_SENSITIVE	The file system supports case-sensitive file names.
FS_UNICODE_STORED_ON_DISK	The file system supports Unicode in file names as they appear on disk.
FS_PERSISTENT_ACLS	The file system preserves and enforces ACLs. For example, NTFS preserves and enforces ACLs, and FAT does not.
FS_FILE_COMPRESSION	The file system supports file-based compression.
FS_VOL_IS_COMPRESSED	The specified volume is a compressed volume; for example, a DoubleSpace volume.
FILE_NAMED_STREAMS	The file system supports named streams.
FILE_SUPPORTS_ENCRYPTION	The file system supports the Encrypted File System (EFS).
FILE_SUPPORTS_OBJECT_IDS	The file system supports object identifiers.
FILE_SUPPORTS_REPARSE_POINTS	The file system supports reparse points.
FILE_SUPPORTS_SPARSE_FILES	The file system supports sparse files.
FILE_VOLUME_QUOTAS	The file system supports disk quotas.

lpFileSystemNameBuffer
[out] Pointer to a buffer that receives the name of the file system (such as FAT or NTFS).

nFileSystemNameSize
[in] Specifies the length, in characters, of the file system name buffer. This parameter is ignored if the file system name buffer is not supplied.

Return Values

If all the requested information is retrieved, the return value is nonzero.

If not all the requested information is retrieved, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

If you are attempting to obtain information about a floppy drive that does not have a floppy disk or a CD-ROM drive that does not have a compact disc, the system displays a message box asking the user to insert a floppy disk or a compact disc, respectively. To prevent the system from displaying this message box, call the **SetErrorMode** function with SEM_FAILCRITICALERRORS.

The FS_VOL_IS_COMPRESSED flag is the only indicator of volume-based compression. The file system name is not altered to indicate compression. This flag comes back set on a DoubleSpace volume, for example. With volume-based compression, an entire volume is either compressed or not compressed.

The FS_FILE_COMPRESSION flag indicates whether a file system supports file-based compression. With file-based compression, individual files can be compressed or not compressed.

The FS_FILE_COMPRESSION and FS_VOL_IS_COMPRESSED flags are mutually exclusive; both bits cannot come back set.

The maximum component length value, stored in the **DWORD** variable pointed to by *lpMaximumComponentLength*, is the only indicator that a volume supports longer-than-normal FAT (or other file system) file names. The file system name is not altered to indicate support for long file names.

The **GetCompressedFileSize** function obtains the compressed size of a file. The **GetFileAttributes** function can determine whether an individual file is compressed.

Requirements

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.

File Systems Overview, *File System Functions*, **GetCompressedFileSize**, **GetFileAttributes**, **SetErrorMode**, **SetVolumeLabel**

GetVolumeNameForVolumeMountPoint

The **GetVolumeNameForVolumeMountPoint** function takes a volume mount point or root directory and returns the corresponding unique volume name.

```
BOOL GetVolumeNameForVolumeMountPoint(
  LPCTSTR lpszVolumeMountPoint, // volume mount point or directory
  LPTSTR lpszVolumeName,        // volume name buffer
  DWORD cchBufferLength         // size of volume name buffer
);
```

Parameters

lpszVolumeMountPoint

[in] Pointer to a string that contains either the path of a volume mount point with a trailing backslash (\) or a drive letter indicating a root directory in the form "D:\".

lpszVolumeName

[out] Pointer to a string that receives the volume name. This name is a unique volume name of the form **"\\?\Volume{*GUID*}\"** where *GUID* is the GUID that identifies the volume.

cchBufferLength

[in] Specifies the length, in characters, of the output buffer. A reasonable size for the buffer to accommodate the largest possible volume name is 50 characters.

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 *lpszVolumeMountPoint* input string may be a drive letter with appended backslash (\), such as "D:\". Alternatively, it may be a path to a volume mount point, again with appended backslash (\), such as "c:\mnt\edrive\".

Use **GetVolumeNameForVolumeMountPoint** to obtain unique volume names for use with other functions that work with volume mount points and volume mounting. For more information, see *Unique Volume Names*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **DeleteVolumeMountPoint**, **GetVolumePathName**, **SetVolumeMountPoint**

GetVolumePathName

The **GetVolumePathName** function returns the volume mount point at which the specified path is mounted.

```
BOOL GetVolumePathName(
  LPCTSTR lpszFileName,          // file path
  LPTSTR lpszVolumePathName,     // volume mount point
  DWORD cchBufferLength          // size of buffer
);
```

Parameters

lpszFileName
[in] Specifies the input path string. Both absolute and relative file and directory names, such as ".", are acceptable in this path.

lpszVolumePathName
[out] Pointer to a string that receives the volume mount point for the input path.

cchBufferLength
[in] Specifies the length, in characters, of the output buffer.

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

Passed a specified path, **GetVolumePathName** returns the path to the volume mount point. In other words, it returns the root of the volume where the end point of the specified path resides.

For example, assume you have volume D mounted at C:\Mnt\Ddrive and volume E mounted at C:\Mnt\Ddrive\Mnt\Edrive. Assume you have a file with the path E:\Dir\Subdir\MyFile. If you pass **GetVolumePathName** the input string "C:\Mnt\Ddrive\Mnt\Edrive\Dir\Subdir\MyFile", it returns the output path "C:\Mnt\Ddrive\Mnt\Edrive".

The length of the path returned by this call always is less than or equal to that of the path passed in. A slower but safer way to set the size of the return buffer is to call the **GetFullPathName** function and then make sure that the buffer is at least as big as the full path **GetFullPathName** returns.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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

File Systems Overview, *File System Functions*, **DeleteVolumeMountPoint**, **GetFullPathName**, **GetVolumeNameForVolumeMountPoint**, **SetVolumeMountPoint**

QueryRecoveryAgentsOnEncryptedFile

The **QueryRecoveryAgentsOnEncryptedFile** function retrieves a list of recovery agents for the specified file.

```
DWORD QueryRecoveryAgentsOnEncryptedFile(
  LPCWSTR lpFileName,                                  // file name
  PENCRYPTION_CERTIFICATE_HASH_LIST *pRecoveryAgents  // hash list
);
```

Parameters

lpFileName
[in] Pointer to a null-terminated Unicode string that specifies the name of the file to query.

pRecoveryAgents
[out] Receives a list of recovery agents, represented by a **ENCRYPTION_CERTIFICATE_HASH_LIST** structure.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a Win32 error code. For a complete list of error codes, see *Error Codes* or the Platform SDK header file WinError.h.

Remarks

When the list of recovery agents is no longer needed, free it by calling the **FreeEncryptionCertificateHashList** function.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.

See Also

File Systems Overview, *File System Functions*, **ENCRYPTION_CERTIFICATE_HASH_LIST**, **FreeEncryptionCertificateHashList**

QueryUsersOnEncryptedFile

The **QueryUsersOnEncryptedFile** function retrieves a list of users for the specified file.

```
DWORD QueryUsersOnEncryptedFile(
  LPCWSTR lpFileName,                          // file name
  PENCRYPTION_CERTIFICATE_HASH_LIST *pUsers  // hash list
);
```

Parameters

lpFileName
[in] Pointer to a null-terminated Unicode string that specifies the name of the file to query.

pUsers
[out] Receives a list of users, represented by a **ENCRYPTION_CERTIFICATE_HASH_LIST** structure.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a Win32 error code. For a complete list of error codes, see *Error Codes* or the Platform SDK header file WinError.h.

Remarks

When the list of users is no longer needed, call the **FreeEncryptionCertificateHashList** function to free the list.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.

See Also

File Systems Overview, *File System Functions*, **ENCRYPTION_CERTIFICATE_HASH_LIST**, **FreeEncryptionCertificateHashList**

RemoveUsersFromEncryptedFile

The **RemoveUsersFromEncryptedFile** function removes specified certificate hashes from a specified file.

```
DWORD RemoveUsersFromEncryptedFile(
  LPCWSTR lpFileName,                          // file name
  PENCRYPTION_CERTIFICATE_HASH_LIST pHashes  // hash list
);
```

Parameters

lpFileName
[in] Pointer to a null-terminated Unicode string that specifies the name of the file.

pHashes
[in] Pointer to an **ENCRYPTION_CERTIFICATE_HASH_LIST** structure that contains a list of certificate hashes to be removed from the file.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a Win32 error code. For a complete list of error codes, see *Error Codes* or the Platform SDK header file WinError.h.

Remarks

The **RemoveUsersFromEncryptedFile** function removes the specified certificate hashes if they exist in the specified file. If any of the certificate hashes are not found in the specified file, they are ignored and no error code is returned.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.

See Also

File Systems Overview, *File System Functions*,
ENCRYPTION_CERTIFICATE_HASH_LIST

SetUserFileEncryptionKey

The **SetUserFileEncryptionKey** function sets the user's current key to the specified certificate.

```
DWORD SetUserFileEncryptionKey(
  PENCRYPTION_CERTIFICATE pEncryptionCertificate  // certificate
);
```

Parameters

pEncryptionCertificate

[in] Pointer to a certificate that will be the user's key. This parameter is a pointer to an **ENCRYPTION_CERTIFICATE** structure.

Return Values

If the function succeeds, the return value is ERROR_SUCCESS.

If the function fails, the return value is a Win32 error code. For a complete list of error codes, see *Error Codes* or the Platform SDK header file WinError.h.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.
Library: Use advapi32.lib.

See Also

File Systems Overview, *File System Functions*, **ENCRYPTION_CERTIFICATE**

SetVolumeMountPoint

The **SetVolumeMountPoint** function mounts the specified volume at the specified volume mount point.

```
BOOL SetVolumeMountPoint(
  LPCTSTR lpszVolumeMountPoint, // mount point
  LPCTSTR lpszVolumeName        // volume to be mounted
);
```

Parameters

lpszVolumeMountPoint

[in] Pointer to a string that indicates the volume mount point where the volume is to be mounted. This may be a root directory (X:\) or a directory on a volume (X:\mnt\). The string must end with a trailing backslash ('\').

lpszVolumeName

[in] Pointer to a string that indicates the volume to be mounted. This string must be a unique volume name of the form **"\\?\Volume{*GUID*}\"** where *GUID* is a GUID that identifies the volume. The **\\?** turns off path parsing and is ignored as part of the path, as discussed in *Path Lengths*. For example, "\\?\C:\myworld\private" is seen as "C:\myworld\private".

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 not an error to attempt to mount a volume at a volume mount point at which a volume is already mounted. In this case, the system unmounts the preceding volume without sending notifications before attempting to mount the new volume.

It is an error to attempt to mount a volume on a directory that has any files or subdirectories in it. This error occurs for system and hidden directories as well as other directories, and it occurs for system and hidden files.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
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.

File Systems Overview, *File System Functions*, **DeleteVolumeMountPoint**, **GetVolumeNameForVolumeMountPoint**, **GetVolumePathName**

File System Control Codes

The following control codes are used with NTFS change journals.

Value	Meaning
FSCTL_CREATE_USN_JOURNAL	Creates an NTFS change journal stream on a target volume or modifies an existing change journal stream.
FSCTL_DELETE_USN_JOURNAL	Deletes the NTFS change journal on a volume or awaits notification of deletion of an NTFS change journal.
FSCTL_ENUM_USN_DATA	Creates an enumeration that lists the NTFS change journal entries between two specified boundaries.
FSCTL_MARK_HANDLE	Marks a specified file or directory and its NTFS change journal record with information about changes to that file or directory.
FSCTL_QUERY_USN_JOURNAL	Queries for information on the current NTFS change journal, its records, and its capacity.
FSCTL_READ_USN_JOURNAL	Returns to the calling process the set of NTFS change journal records between two specified USN values.

The following control codes are used with opportunistic locks.

Value	Meaning
FSCTL_OPBATCH_ACK_CLOSE_PENDING	Notifies a server that a client application is about to close a file. An application uses this operation following notification that an opportunistic lock on the file is about to be broken.
FSCTL_OPLOCK_BREAK_ACK_NO_2	Responds to notification that an opportunistic lock on a file is about to be broken. An application uses this operation to loose all opportunistic locks on the file but keep the file open.

FSCTL_OPLOCK_BREAK_ACKNOWLEDGE	Responds to notification that an exclusive opportunistic lock on a file is about to be broken. An application uses this operation to indicate that the file should receive a level 2 opportunistic lock.
FSCTL_OPLOCK_BREAK_NOTIFY	Allows the calling application to wait for completion of an opportunistic lock break.
FSCTL_REQUEST_BATCH_OPLOCK	Requests a batch opportunistic lock on a file.
FSCTL_REQUEST_FILTER_OPLOCK	Requests a filter opportunistic lock on a file.
FSCTL_REQUEST_OPLOCK_LEVEL_1	Requests a level 1 opportunistic lock on a file.
FSCTL_REQUEST_OPLOCK_LEVEL_2	Requests a level 2 opportunistic lock on a file.

The following control codes are used with reparse points.

Value	Meaning
FSCTL_DELETE_REPARSE_POINT	Deletes a reparse point for a file or directory.
FSCTL_GET_REPARSE_POINT	Returns reparse point data for a file or directory.
FSCTL_SET_REPARSE_POINT	Sets a reparse point on a file or directory.

The following control codes are used with sparse files.

Value	Meaning
FSCTL_QUERY_ALLOCATED_RANGES	Scans a file for ranges of the file for which disk space is allocated.
FSCTL_SET_SPARSE	Marks a file as a sparse file.
FSCTL_SET_ZERO_DATA	Sets a range of a files bytes to zeroes.

File System Interfaces

IDiskQuotaControl

The **IDiskQuotaControl** interface provides methods for controlling the disk quota facilities of a single NTFS volume. The client can query and set volume-specific quota attributes through **IDiskQuotaControl**. The client can also enumerate all per-user quota entries on the volume. A client instantiates this interface by calling the **CoCreateInstance** function using the class identifier CLSID_DiskQuotaControl.

Virtual function table

IUnknown Methods

QueryInterface
AddRef
Release

IDiskQuotaControl Methods

Initialize
SetQuotaState
GetQuotaState
SetQuotaLogFlags
GetQuotaLogFlags
SetDefaultQuotaThreshold
GetDefaultQuotaThreshold
GetDefaultQuotaThresholdText
SetDefaultQuotaLimit
GetDefaultQuotaLimit
GetDefaultQuotaLimitText
AddUserSid
AddUserName
DeleteUser
FindUserSid
FindUserName
CreateEnumUsers
CreateUserBatch
InvalidateSidNameCache
GiveUserNameResolutionPriority
ShutdownNameResolution

Remarks

The disk quota control object also implements the **IConnectionPointContainer** interface to service the **IDiskQuotaEvents** interface.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*

IDiskQuotaControl::AddUserName

Adds a new quota entry on the volume for the specified user. The user is identified by domain and account name.

```
HRESULT AddUserName(
  LPCWSTR pszLogonName,
  DWORD fNameResolution,
  PDISKQUOTA_USER *ppUser
);
```

Parameters

pszLogonName

Pointer to the user's account logon name string.

fNameResolution

Indicates how the user account information is to be obtained. The volume's quota information identifies users by SID. The user account information (such as container, logon name, and display name) must be obtained from the network domain controller, or the local computer if it is not on a network. This parameter can be one of the following values.

Value	Meaning
DISKQUOTA_USERNAME_RESOLVE_NONE	Do not resolve user account information.
DISKQUOTA_USERNAME_RESOLVE_SYNC	Resolve user account information synchronously. **AddUserName** returns when the information is resolved. If the information exists in the disk quota SID cache, it is returned immediately. Otherwise, the method must locate the information. This can take several seconds.
DISKQUOTA_USERNAME_RESOLVE_ASYNC	Resolve user account information asynchronously. **AddUserName** returns immediately. The caller must implement the **IDiskQuotaEvents** interface to receive notification when the information is available. If the information was cached during a previous request, notification occurs as soon as the object is serviced. Otherwise, the method obtains the information from the network domain controller, then notifies **IDiskQuotaEvents**.

ppUser

Pointer to receive the **IDiskQuotaUser** interface pointer to the newly created quota user object.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
S_OK	Success.
S_FALSE	User already exists. Not added.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
ERROR_USER_UNKNOWN	The specified user name is unknown.

(continued)

(continued)

Value	Meaning
E_INVALIDARG	A pointer parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Remarks

The NTFS file system automatically creates a user quota entry when a user first writes to the volume. Entries that are created automatically are assigned the default warning threshold and hard quota limit values for the volume. This method allows you to create a user quota entry before a user has written information to the volume. Therefore, you can pre-assign a warning threshold or hard quota limit value different than the volume default settings.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::AddUserSid

Adds a new quota entry on the volume for the specified user. The user is identified by security identifier (SID).

```
HRESULT AddUserSid(
  PSID pUserSid,
  DWORD fNameResolution,
  PDISKQUOTA_USER *ppUser
);
```

Parameters

pUserSid
Pointer to a buffer containing the user's SID.

fNameResolution
Indicates how the user account information is to be obtained. The volume's quota information identifies users by SID. The user account information (such as domain name, account name, and full name) must be obtained from the network domain controller, or the local computer if it is not on a network. This parameter can be one of the following values:

Value	Meaning
DISKQUOTA_USERNAME_RESOLVE_NONE	Do not resolve user account information.
DISKQUOTA_USERNAME_RESOLVE_SYNC	Resolve user account information synchronously. **AddUserSid** returns when the information is resolved. If the information exists in the disk quota SID cache, it is returned immediately. Otherwise, the method must locate the information. This can take several seconds.
DISKQUOTA_USERNAME_RESOLVE_ASYNC	Resolve user account information asynchronously. **AddUserSid** returns immediately. The caller must implement the **IDiskQuotaEvents** interface to receive notification when the information is available. If the information was cached during a previous request, notification occurs as soon as the object is serviced. Otherwise, the method obtains the information from the network domain controller, then notifies **IDiskQuotaEvents**.

ppUser

Pointer to receive the **IDiskQuotaUser** interface pointer to the newly created quota user object.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_INVALID_SID	The specified SDI is unknown.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pUserSid* or *ppUser* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Remarks

The NTFS file system automatically creates a user quota entry when a user first writes to the volume. Entries that are created automatically are assigned the default warning threshold and hard quota limit values for the volume. This method allows you to create a user quota entry before a user has written information to the volume. Therefore, you can

pre-assign a warning threshold or hard quota limit value different than the volume default settings.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::CreateEnumUsers

Creates an enumerator object for enumerating quota users on the volume. The newly created object implements the **IEnumDiskQuotaUsers** interface.

```
HRESULT CreateEnumUsers(
  PSID *rgpUserSids,
  DWORD cpSids,
  DWORD fNameResolution,
  PENUM_DISKQUOTA_USERS *ppEnum
);
```

Parameters

rgpUserSids
Array of security identifier (SID) pointers representing the user objects to be included in the enumeration. If this value is NULL, all user entries are enumerated.

cpSids
Number of items in the *rgpUserSids* array. Ignored if *rgpUserSids* is NULL.

fNameResolution
Indicates how the user account information is to be obtained. The volume's quota information identifies users by SID. The user account information (such as domain name, account name, and full name) must be obtained from the network domain controller, or the local computer if it is not on a network. This parameter can be one of the following values.

Value	Meaning
DISKQUOTA_USERNAME_RESOLVE_NONE	Do not resolve user account information.

(continued)

(continued)

Value	Meaning
DISKQUOTA_USERNAME_RESOLVE_SYNC	Resolve user account information synchronously. The **IEnumDiskQuotaUsers::Next** method returns when the information is resolved. If the information exists in the disk quota SID cache, it is returned immediately. Otherwise, the method must locate the information. This can take several seconds.
DISKQUOTA_USERNAME_RESOLVE_ASYNC	Resolve user account information asynchronously. The **IEnumDiskQuotaUsers::Next** method returns immediately. The caller must implement the **IDiskQuotaEvents** interface to receive notification when the information is available. If the information was cached during a previous request, notification occurs as soon as the object is serviced. Otherwise, the method obtains the information from the network domain controller, then notifies **IDiskQuotaEvents**.

ppEnum
Pointer to a pointer to the **IEnumDiskQuotaUsers** enumerator.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *ppEnum* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::CreateUserBatch

Creates a batching object for optimizing updates to the quota settings of multiple users simultaneously.

```
HRESULT CreateUserBatch(
  PDISKQUOTA_USER_BATCH *ppBatch
);
```

Parameters

ppBatch
Pointer to receive the **IDiskQuotaUserBatch** interface pointer.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *ppBatch* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, File System Interfaces, **IDiskQuotaControl**

IDiskQuotaControl::DeleteUser

Removes a user entry from the volume quota information file, if the user's charged quota amount is zero bytes.

```
HRESULT DeleteUser(
  PDISKQUOTA_USER pUser
);
```

Parameters

pUser

Pointer to the **IDiskQuotaUser** interface of the user whose quota record is marked for deletion.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_FILE_EXISTS	The user owns files on the volume.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pUser* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Remarks

This method does not actually remove the quota entry from the volume. It marks the entry for deletion. NTFS performs the actual deletion at a later time. Following a call to **IDiskQuotaControl::DeleteUser**, the **IDiskQuotaUser** interface is still active. This method does not delete the user object from memory. To release the user object, call **IUnknown::Release**.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::FindUserName

Locates a specific entry in the volume quota information. The user's account logon name is used as the search key.

```
HRESULT FindUserName(
  LPCWSTR pszLogonName,
  PDISKQUOTA_USER *ppUser
);
```

Parameters

pszLogonName
Pointer to the user's account logon name.

ppUser
Pointer to receive the **IDiskQuotaUser** interface pointer to the quota user object.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_INVALID_SID	The SID for the user is invalid.
ERROR_NONE_MAPPED	There is no mapping available for the SID.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pUserSid* or *ppUser* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Remarks

This method will return a user object even if there is no quota record for the user in the quota file. This is consistent with the idea of automatic user addition and default quota settings. If there is currently no quota entry for the requested user, and the user would be added to the quota file if he were to request disk space, the returned user object will have warning threshold and hard quota limits equal to the volume default settings.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::FindUserSid

Locates a specific user entry in the volume quota information. The user's security identifier (SID) is used as the search key.

```
HRESULT FindUserSid(
  PSID pUserSid,
  DWORD fNameResolution,
  PDISKQUOTA_USER *ppUser
);
```

Parameters

pUserSid
Pointer to the user's SID.

fNameResolution
Indicates how the user account information is to be obtained. The volume's quota information identifies users by SID. The user account information (such as domain name, account name, and full name) must be obtained from the network domain controller, or the local computer if it is not on a network. This parameter can be one of the following values.

Value	Meaning
DISKQUOTA_USERNAME_RESOLVE_NONE	Do not resolve user account information.
DISKQUOTA_USERNAME_RESOLVE_SYNC	Resolve user account information synchronously. **FindUserSid** returns when the information has been resolved. If the information exists in the disk quota SID cache, it is returned immediately. Otherwise, the method must locate the information. This can take several seconds.
DISKQUOTA_USERNAME_RESOLVE_ASYNC	Resolve user account information asynchronously. **FindUserSid** returns immediately. The caller must implement the **IDiskQuotaEvents** interface to receive notification when the information is available. If the information was cached during a previous request, notification occurs as soon as the object is serviced. Otherwise, the method obtains the information from the network domain controller, then notifies **IDiskQuotaEvents**.

ppUser
Pointer to receive the **IDiskQuotaUser** interface pointer to the quota user object.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
ERROR_INVALID_SID	The SID for the user is invalid.
E_INVALIDARG	The *pUserSid* or *ppUser* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Remarks

This method will return a user object even if there is no quota record for the user in the quota file. This is consistent with the idea of automatic user addition and default quota settings. If there is currently no quota entry for the requested user, and the user would be added to the quota file if he were to request disk space, the returned user object will have warning threshold and hard quota limits equal to the volume default settings.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GetDefaultQuotaLimit

Retrieves the default quota limit for the volume. This limit is applied automatically to new users of the volume.

```
HRESULT GetDefaultQuotaLimit(
  PLONGLONG pllLimit
);
```

Parameters

pllLimit

Pointer to the variable to receive the quota limit.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pllLimit* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GetDefaultQuotaLimitText

Retrieves the default quota limit for the volume. This limit is applied automatically to new users of the volume. The limit is expressed as a text string, for example "10.5 MB". If the volume has no limit, the string returned is "No Limit" (localized). If the buffer is too small, it is truncated to fit the buffer.

```
HRESULT GetDefaultQuotaLimitText(
  LPWSTR pszText,
  DWORD cchText
);
```

Parameters

pszText

Pointer to the buffer to receive the text string.

cchText
Size of the buffer, in characters.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pszText* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GetDefaultQuotaThreshold

Retrieves the default quota warning threshold for the volume. This threshold is applied automatically to new users of the volume.

```
HRESULT GetDefaultQuotaThreshold(
  PLONGLONG pllThreshold
);
```

Parameters

pllThreshold
Pointer to the variable to receive the default warning threshold value, in bytes.

Return Values

This method returns a file system error or one of the following values:

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pllThreshold* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GetDefaultQuotaThresholdText

Retrieves the default warning threshold for the volume. This threshold is expressed as a text string, for example "10.5 MB". If the volume does not have a threshold, the string returned is "No Limit" (localized). If the buffer is too small, the string is truncated to fit the buffer.

```
HRESULT GetDefaultQuotaThresholdText(
  LPWSTR pszText,
  DWORD cchText
);
```

Parameters

pszText
Pointer to the buffer to receive the text string.

cchText
Size of the buffer, in characters.

Return Values

This method returns a file system error or one of the following values:

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pszText* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GetQuotaLogFlags

Retrieves the flags that control the logging of user-related quota events on the volume. Logging makes an entry in the volume server's event log.

```
HRESULT GetQuotaLogFlags(
  LPDWORD pdwFlags
);
```

Parameters

pdwFlags

Pointer to a variable to receive the volume's quota logging flags. Use the following macros to retrieve the contents of the flag value.

Flag	Meaning
DISKQUOTA_IS_LOGGED_USER_THRESHOLD	If set, an event log entry will be created when the user exceeds his assigned warning threshold.
DISKQUOTA_IS_LOGGED_USER_LIMIT	If set, an event log entry will be created when the user exceeds his assigned hard quota limit.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pdwFlags* parameter is incorrect.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GetQuotaState

Retrieves a set of flags describing the state of the quota system.

```
HRESULT GetQuotaState(
  LPDWORD pdwState
);
```

Parameters

pdwState

Pointer to a variable to receive the quota state flags. This parameter can be one or more of the following flags.

Flag	Meaning
DISKQUOTA_STATE_DISABLED	Quotas are not enabled on the volume.
DISKQUOTA_STATE_TRACK	Quotas are enabled but the limit value is not being enforced. Users may exceed their quota limit.
DISKQUOTA_STATE_ENFORCE	Quotas are enabled and the limit value is enforced. Users cannot exceed their quota limit.

(continued)

(continued)

Flag	Meaning
DISKQUOTA_FILESTATE_INCOMPLETE	The volume's quota information is out of date. Quotas are probably disabled.
DISKQUOTA_FILESTATE_REBUILDING	The volume is rebuilding its quota information.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *pdwState* parameter is incorrect.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::GiveUserNameResolutionPriority

Promotes the specified user object to the head of the queue so that it is next in line for resolution. By default, quota user objects are serviced in the order in which they were placed in the queue.

This method is applicable only when asynchronous name resolution is used.

```
HRESULT GiveUserNameResolutionPriority(
  PDISKQUOTA_USER pUser
);
```

Parameters

pUser
Pointer to the **IDiskQuotaUser** interface.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
S_FALSE	Quota user object not in the resolver queue.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Functions*, **IDiskQuotaControl**

IDiskQuotaControl::Initialize

Initializes a new **QuotaControl** object by opening the NTFS volume with the requested access rights. The return value indicates whether the volume supports NTFS disk quotas and whether the caller has sufficient access rights.

```
HRESULT Initialize(
  LPCWSTR pszPath,
  BOOL bReadWrite
);
```

Parameters

pszPath
Specifies the path to the volume root.

bReadWrite
If this value is TRUE, the volume is opened in read/write mode. If this value is FALSE, the volume is opened in read-only mode.

Return Values

This method returns one of the following values:

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_BAD_PATHNAME	The requested path name is invalid.
ERROR_FILE_NOT_FOUND	The requested file or object was not found.
ERROR_INITIALIZED	The controller object has already been initialized. Multiple initialization is not allowed.
ERROR_INVALID_NAME	The requested file path is invalid.
ERROR_NOTSUPPORTED	The file system does not support quotas.
ERROR_PATH_NOT_FOUND	The requested file path was not found.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaControl**

IDiskQuotaControl::InvalidateSidNameCache

Invalidates the contents of the system's SID-to-name cache so subsequent requests for new user objects (**IEnumDiskQuotaUsers::Next**, **IDiskQuotaControl::FindUserSid** and **IDiskQuotaControl::FindUserName**) must obtain user names from the network domain controller. As names are obtained, they are cached.

```
HRESULT InvalidateSidNameCache(void);
```

Parameters

This method has no parameters.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_OUTOFMEMORY	Insufficient memory.

E_UNEXPECTED	An unexpected exception occurred.
E_FAIL	The SID-to-name cache is not available or could not be exclusively locked.

Remarks

In general, there is no reason to call this method. It is included to provide a method for programmatically refreshing the entire SID-to-name cache.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaControl**

IDiskQuotaControl::SetDefaultQuotaLimit

Modifies the default quota limit. This limit is applied automatically to new users of the volume.

```
HRESULT SetDefaultQuotaLimit(
  LONGLONG llLimit
);
```

Parameters

llLimit
The value for the default quota limit, in bytes.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaControl**

IDiskQuotaControl::SetDefaultQuotaThreshold

Modifies the default warning threshold. This threshold is applied automatically to new users of the volume.

```
HRESULT SetDefaultQuotaThreshold(
  LONGLONG llThreshold
);
```

Parameters

llThreshold
Specifies the default warning threshold value, in bytes.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaControl**

IDiskQuotaControl::SetQuotaLogFlags

Controls the logging of user-related quota events on the volume. Logging makes an entry in the volume server system's event log.

```
HRESULT SetQuotaLogFlags(
  DWORD dwFlags
);
```

Parameters

dwFlags

Specifies the log flags to be applied to the volume. Use the following macros to set the proper bits in the *dwFlags* parameter.

Macro	Meaning
DISKQUOTA_SET_LOG_USER_THRESHOLD	Turn on/off logging of user warning threshold violations. If set, an event log entry will be created when the user exceeds his assigned warning threshold.
DISKQUOTA_SET_LOG_USER_LIMIT	Turn on/off logging of user quota limit violations. If set, an event log entry will be created when the user exceeds his assigned hard quota limit.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaControl**

IDiskQuotaControl::SetQuotaState

Sets the state of the quota system. Not all state attributes can be modified. The enable, track, and enforce attributes can be modified.

```
HRESULT SetQuotaState(
  DWORD dwState
);
```

Parameters

dwState

Specifies the state to be applied to the volume. Use the following macros to set the proper bits in the *dwState* parameter.

Macro	Enable	Track	Enforce
DISKQUOTA_SET_DISABLED	No	No	No
DISKQUOTA_SET_TRACKED	Yes	Yes	No
DISKQUOTA_SET_ENFORCED	Yes	Yes	Yes

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_NOT_READY	The DiskQuotaControl object is not initialized.
E_INVALIDARG	The *dwState* parameter is incorrect.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaControl**

IDiskQuotaControl::ShutdownNameResolution

The SID-to-name resolver translates user security identifiers (SID) to user names. It runs as a background thread. When a quota control object is destroyed, this thread automatically terminates. The final call to the **IUnknown::Release** method terminates the thread. This is normally all that is required. If you finish with the quota control object, but it is not ready to be destroyed (there are other open reference counts), call this method to terminate the background thread before the object is destroyed.

```
HRESULT ShutdownNameResolution(void);
```

Parameters

This method has no parameters.

Return Values

This method returns NOERROR.

Remarks

Asynchronous name resolution will also cease after the thread terminates. A subsequent call to the following methods may recreate the SID-to-name resolver thread:

IDiskQuotaControl::AddUserName
IDiskQuotaControl::AddUserSid
IDiskQuotaControl::CreateEnumUsers
IDiskQuotaControl::FindUserName
IDiskQuotaControl::FindUserSid

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, ***IDiskQuotaControl***

IDiskQuotaEvents

A client must implement the **IDiskQuotaEvents** interface as an event sink to receive quota-related event notifications. Its methods are called by the system whenever significant quota events have occurred. Currently, the only event supported is the asynchronous resolution of user account name information.

Virtual Function Table

IUnknown Methods

QueryInterface
AddRef
Release

IDiskQuotaEvents Methods

OnUserNameChanged

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*,

IDiskQuotaEvents::OnUserNameChanged

Notifies the client's connection sink whenever a user's SID has been asynchronously resolved. If **IDiskQuotaUser::GetAccountStatus** returns DISKQUOTA_USER_ACCOUNT_RESOLVED, the user's account container name, logon name, and display name strings are available in the quota user object.

```
HRESULT OnUserNameChanged(
  PDISKQUOTA_USER pUser
);
```

Parameters

pUser

Pointer to the **IDiskQuotaUser** interface for the quota user object. It is not necessary to call **Release** on this pointer. The **DiskQuotaControl** object controls the lifetime of the user object.

Return Values

The return value is ignored.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaEvents**

IDiskQuotaUser

The **IDiskQuotaUser** interface represents a single user quota entry in the volume quota information file. Through this interface, you can query and modify user-specific quota information on an NTFS volume. This interface is instantiated using **IEnumDiskQuotaUsers**, **IDiskQuotaControl::FindUserSid**, **IDiskQuotaControl::FindUserName**, **IDiskQuotaControl::AddUserSid** or **IDiskQuotaControl::AddUserName**.

Virtual Function Table

IUnknown Methods

QueryInterface
AddRef
Release

IDiskQuotaUser Methods

GetID
GetName
GetSidLength
GetSid
GetQuotaThreshold
GetQuotaThresholdText
GetQuotaLimit
GetQuotaLimitText
GetQuotaUsed
GetQuotaUsedText
GetQuotaInformation
SetQuotaThreshold
SetQuotaLimit
Invalidate
GetAccountStatus

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*,

IDiskQuotaUser::GetAccountStatus

Retrieves the status of the user object's account. User information is identified in the quota system by user security identifier (SID). This SID must resolve to a user account for the user's account name information to be retrieved. To determine why a user's name strings are not available, use the status information.

```
HRESULT GetAccountStatus(
  LPDWORD pdwStatus
);
```

Parameters

pdwStatus
Pointer to receive the user's account status. The status value can be one of the following.

Value	Meaning
DISKQUOTA_USER_ACCOUNT_RESOLVED	The SID was resolved to a user account. Names are available through **IDiskQuotaUser::GetName**.
DISKQUOTA_USER_ABLE	The user account is unavailable at this time. The network domain controller may not be available. Name information is not available.
DISKQUOTA_USER_ACCOUNT_DELETED	The user account was deleted from the domain. Name information is not available.
DISKQUOTA_USER_ACCOUNT_INVALID	The user account is invalid. Name information is not available.
DISKQUOTA_USER_ACCOUNT_UNKNOWN	The user account is unknown. Name information is not available.
DISKQUOTA_USER_ACCOUNT_UNRESOLVED	The SID has not been resolved to a user account.

Return Values

This method returns one of the following values:

Value	Meaning
NOERROR	Success.
E_INVALIDARG	The *pdwStatus* parameter is NULL.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetID

Retrieves a unique identifier (ID) number for the DiskQuotaUser object. This ID is unique only within the process. It can be used to identify a user object in a set of user objects if the programming language you are using does not support pointers.

```
HRESULT GetID(
  ULONG *pulID
);
```

Parameters

pulID
Pointer to the name strings associated with the disk quota user.

Return Values

This method returns NOERROR.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetName

Retrieves the name strings associated with a disk quota user.

```
HRESULT GetName(
  LPWSTR pszAccountContainer,
  DWORD cchAccountContainer,
  LPWSTR pszLogonName,
  DWORD cchLogonName,
  LPWSTR pszDisplayName,
  DWORD cchDisplayName
);
```

Parameters

pszAccountContainer
Pointer to the buffer to receive the name of the user's account container. This value can be NULL. For Windows NT 4.0 accounts, or for other accounts on without directory service information, this string is simply the domain name. For accounts with directory service information available, this string is a canonical name with the terminating object name removed.

cchAccountContainer
Size of the account container buffer, in characters. Ignored if *pszAccountContainer* is NULL.

pszLogonName
Pointer to the buffer to receive the name the user specified to log on the computer. This value can be NULL. The format of the name returned depends on whether directory service information is available.

cchLogonName
Size of the logon name buffer, in characters. Ignored if *pszLogonName* is NULL.

pszDisplayName
Pointer to the buffer to receive the display name for the quota user. This value can be NULL. If the information is not available, the string returned is of zero length.

cchDisplayName
Size of the display-name buffer, in characters. Ignored if *pszDisplayName* is NULL.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaInformation

Retrieves the values for the user's warning threshold, hard quota limit, and quota used.

```
HRESULT GetQuotaInformation(
  LPVOID pbQuotaInfo,
  DWORD cbQuotaInfo
);
```

Parameters

pbQuotaInfo
Pointer to the **DISKQUOTA_USER_INFORMATION** structure to receive the quota information.

cbQuotaInfo
Size of the quota information structure, in bytes.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pQuotaInfo* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaLimit

Retrieves the user's quota limit value on the volume. The limit is set as the maximum amount of disk space available to the volume user.

```
HRESULT GetQuotaLimit(
  PLONGLONG pllLimit
);
```

Parameters

pllLimit

Pointer to the variable to receive the limit value. If this value is –1, the user has a unlimited quota.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pllLimit* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaLimitText

Retrieves the user's quota limit for the volume. This limit is expressed as a text string, for example "10.5 MB". If the user has no quota limit, the string returned is "No Limit" (localized). If the destination buffer is too small, the string is truncated to fit the buffer.

```
HRESULT GetQuotaLimitText(
  LPWSTR pszText,
  DWORD cchText
);
```

Parameters

pszText
Pointer to the buffer to receive the text string.

cchText
Size of the buffer, in characters.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pszText* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaThreshold

Retrieves the user's warning threshold value on the volume. The threshold is an arbitrary value set by the volume's quota administrator. You can use it to identify users who are approaching their hard quota limit.

```
HRESULT GetQuotaThreshold(
  PLONGLONG pllThreshold
);
```

Parameters

pllThreshold
Pointer to a variable to receive the warning threshold value.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pllThreshold* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

! Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaThresholdText

Retrieves the user's warning threshold for the volume. This threshold is expressed as a text string, for example "10.5 MB". If the user's threshold is unlimited, the string returned is "No Limit" (localized). If the destination buffer is too small, the string is truncated to fit the buffer.

```
HRESULT GetQuotaThresholdText(
  LPWSTR pszText,
  DWORD cchText
);
```

Parameters

pszText
Pointer to the buffer to receive the text string.

cchText
Size of the destination buffer, in characters.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pszText* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaUsed

Retrieves the user's quota used value on the volume. This is the amount of information stored on the volume by the user. Note that this is the amount of uncompressed information. Therefore, the use of NTFS compression does not affect this value.

```
HRESULT GetQuotaUsed(
  PLONGLONG pllUsed
);
```

Parameters

pllUsed
Pointer to the variable to receive the quota used value.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pllUsed* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetQuotaUsedText

Retrieves the user's quota used value for the volume. This value is expressed as a text string, for example "10.5 MB". If the destination buffer is too small, the string is truncated to fit the buffer.

```
HRESULT GetQuotaUsedText(
  LPWSTR pszText,
  DWORD cchText
);
```

Parameters

pszText
Pointer to the buffer to receive the text string.

cchText
Size of the buffer, in bytes.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_INVALIDARG	The *pszText* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetSid

Retrieves the user's security identifier (SID).

```
HRESULT GetSid(
  LPBYTE pbSidBuffer,
  DWORD cbSidBuffer
);
```

Parameters

pbSidBuffer
Pointer to the buffer to receive the SID.

cbSidBuffer
Size of the buffer, in bytes. Use the **IDiskQuotaUser::GetSidLength** method to obtain the required size for the buffer.

Return Values

This method returns one of the following values:

Value	Meaning
NOERROR	Success.
E_INVALIDARG	The *pbSidBuffer* parameter is NULL.
ERROR_INVALID_SID	The SID for the user is invalid.
ERROR_INSUFFICIENT_BUFFER	Insufficient destination buffer size.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::GetSidLength

Retrieves the length of the user's security identifier (SID), in bytes. Use the return value to determine the size of the destination buffer you pass to **IDiskQuotaUser::GetSid**.

```
HRESULT GetSidLength(
  LPDWORD pdwLength
);
```

Parameters

pdwLength
Pointer to the variable to receive the SID length, in bytes.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
E_INVALIDARG	The *pdwLength* parameter is NULL.
ERROR_INVALID_SID	The SID for the user is invalid.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::Invalidate

Invalidates the quota information stored in the quota user object. The next time information is requested, it is refreshed from disk.

```
HRESULT Invalidate(void);
```

Parameters

This method has no parameters.

Return Values

This method returns NOERROR.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::SetQuotaLimit

Sets the user's quota limit value on the volume. The limit is set as the maximum amount of disk space available to the volume user.

```
HRESULT SetQuotaLimit(
  LONGLONG llLimit,
  BOOL fWriteThrough
);
```

Parameters

llLimit

Specifies the limit value, in bytes. If this value is –1, the user has an unlimited quota.

fWriteThrough

If this value is TRUE, the value is written immediately to the volume's quota file. Otherwise, the value is written only to the quota user object's local memory. This value should typically be set to TRUE. Set it to FALSE when using the **IDiskQuotaUserBatch** interface to modify multiple user quota entries at once.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_FAIL	An unexpected file system error occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUser::SetQuotaThreshold

Sets the user's warning threshold value on the volume. The threshold is an arbitrary value set by the volume's quota administrator. You can use it to identify users who are approaching their hard quota limit.

```
HRESULT SetQuotaThreshold(
  LONGLONG llThreshold,
  BOOL fWriteThrough
);
```

Parameters

llThreshold

Specifies the warning threshold value.

fWriteThrough
If this value is TRUE, the value is written immediately to the volume's quota file. Otherwise, the value is written only to the quota user object's local memory. This value should typically be set to TRUE. Set it to FALSE when using the **IDiskQuotaUserBatch** interface to modify multiple user quota entries at the same time.

Return Values

This method returns a file system error or one of the following values.

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
E_FAIL	An unexpected file system error occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IDiskQuotaUser**

IDiskQuotaUserBatch

The **IDiskQuotaUserBatch** interface enables you to add multiple quota user objects to a container that is then submitted for update in a single call. This reduces the number of calls to the underlying file system, improving update efficiency when a large number of user objects must be updated. This interface is instantiated by using the **IDiskQuotaControl::CreateUserBatch** method.

Virtual Function Table

IUnknown Methods

QueryInterface
AddRef
Release

IDiskQuotaUserBatch Methods

Add
Remove
RemoveAll
FlushToDisk

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*

IDiskQuotaUserBatch::Add

Adds an **IDiskQuotaUser** pointer to the batch list. This method calls **AddRef** on the *pUser* interface pointer. **Release** is automatically called on each contained **IDiskQuotaUser** interface pointer when the batch object is destroyed.

When setting values on a quota user object in preparation for batch processing, specify FALSE for the *fWriteThrough* parameter in the **IDiskQuotaUser::SetQuotaLimit** and **IDiskQuotaUser::SetQuotaThreshold** methods. This stores the values in memory without writing to disk. To write the changes to disk, call the **IDiskQuotaUserBatch::FlushToDisk** method.

```
HRESULT Add(
  PDISKQUOTA_USER pUser
);
```

Parameters

pUser
Pointer to the quota user object's **IDiskQuotaUser** interface.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
E_INVALIDARG	The *pUser* parameter is NULL.

E_OUTOFMEMORY	Insufficient memory.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUserBatch**

IDiskQuotaUserBatch::Remove

Removes an **IDiskQuotaUser** pointer from the batch list.

```
HRESULT Remove(
  PDISKQUOTA_USER pUser
);
```

Parameters

pUser
Pointer to the quota user object's **IDiskQuotaUser** interface.

Return Values

This method returns one of the following values.

Value	Meaning
S_OK	Success.
S_FALSE	Quota user object not found in batch.
E_INVALIDARG	The *pUser* parameter is NULL.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUserBatch**

IDiskQuotaUserBatch::RemoveAll

Removes all **IDiskQuotaUser** pointers from the batch list.

```
HRESULT RemoveAll(void);
```

Parameters

This method has no parameters.

Return Values

This method returns one of the following values.

Value	Meaning
NOERROR	Success.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUserBatch**

IDiskQuotaUserBatch::FlushToDisk

Writes user object changes to disk in a single call to the underlying file system.

```
HRESULT FlushToDisk(void);
```

Parameters

This method has no parameters.

Return Values

This method returns a file system error or one of the following values:

Value	Meaning
NOERROR	Success.
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
E_OUTOFMEMORY	Insufficient memory.
E_FAIL	An unexpected file system error occurred.
E_UNEXPECTED	An unexpected exception occurred.

Remarks

There are limitations on the amount of information that can be written to disk in a single call to the file system. The flush operation may generate multiple calls to the file system. Nonetheless, the batch operation will be more efficient than a single call for each user object.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IDiskQuotaUserBatch**

IEnumDiskQuotaUsers

The **IEnumDiskQuotaUsers** interface enumerates user quota entries on the volume. This interface is instantiated by using the **IDiskQuotaControl::CreateEnumUsers** method.

Virtual Function Table

IUnknown Methods

QueryInterface
AddRef
Release

IEnumDiskQuotaUsers Methods

Next
Skip
Reset
Clone

Remarks

To use this interface, the client directs an object that maintains a collection of items to create an enumerator object. By repeatedly calling the **Next** method, the client gets successive pointers to each item in the collection.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*

IEnumDiskQuotaUsers::Clone

Creates another enumerator that contains the same enumeration state as the current one. Using this method, a client can record a particular point in the enumeration sequence, and then return to that point at a later time. The new enumerator supports the same interface as the original one.

```
HRESULT Clone(
  IEnumDiskQuotaUsers **ppEnum
);
```

Parameters

ppEnum

Pointer to the variable to receive the **IEnumDiskQuotaUsers** interface pointer. If the method is unsuccessful, the value of this variable is undefined.

Return Values

This method returns one of the following values.

Value	Meaning
E_INVALIDARG	The *ppEnum* parameter is NULL.
E_OUTOFMEMORY	Insufficient memory.
E_UNEXPECTED	An unexpected exception occurred.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IEnumDiskQuotaUsers**

IEnumDiskQuotaUsers::Next

Retrieves the next *cUsers* items in the enumeration sequence. If there are fewer than the requested number of elements left in the sequence, it retrieves the remaining elements. The number of elements actually retrieved is returned through *pcUsersFetched* (unless the caller passed in NULL for that parameter).

```
HRESULT Next(
  DWORD cUsers,
  PDISKQUOTA_USER *rgUsers,
  LPDWORD pcUsersFetched
);
```

Parameters

cUsers
Specifies the number of elements being requested.

rgUsers
Array of size *cUsers* or larger.

pcUsersFetched
Pointer to the number of elements actually supplied in *rgUsers*. The caller can pass in NULL if *cUsers* is one.

Return Values

The return value is NOERROR if the number of elements supplied is *cUsers*; otherwise, the return value is S_FALSE.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IEnumDiskQuotaUsers**

IEnumDiskQuotaUsers::Reset

Resets the enumeration sequence to the beginning.

```
HRESULT Reset(void);
```

Parameters

This method has no parameters.

Return Values

The return value is NOERROR.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Interfaces*, **IEnumDiskQuotaUsers**

IEnumDiskQuotaUsers::Skip

Skips over the next specified number of elements in the enumeration sequence.

```
HRESULT Skip(
  DWORD cUsers
);
```

Parameters

cUsers
  Specifies the number of elements to be skipped.

Return Values

The return value is NOERROR if the number of elements skipped is *cUsers*; otherwise, the return value is S_FALSE.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

File Systems Overview, *File System Interfaces*, **IEnumDiskQuotaUsers**

File System Structures

DISKQUOTA_USER_INFORMATION

The **DISKQUOTA_USER_INFORMATION** structure represents the per-user quota information.

```
typedef struct DiskQuotaUserInformation {
  LONGLONG QuotaUsed;
  LONGLONG QuotaThreshold;
  LONGLONG QuotaLimit;
} DISKQUOTA_USER_INFORMATION, *PDISKQUOTA_USER_INFORMATION;
```

Members

QuotaUsed

Disk space charged to the user, in bytes. This is the amount of information stored, not necessarily the number of bytes used on disk.

QuotaThreshold

Warning threshold for the user, in bytes. You can use the **IDiskQuotaControl::SetQuotaLogFlags** method to configure the system to generate a system logfile entry when the disk space charged to the user exceeds this value.

QuotaLimit

Quota limit for the user, in bytes. You can use the **IDiskQuotaControl::SetQuotaLogFlags** method to configure the system to generate a system logfile entry when the disk space charged to the user exceeds this value. You can also use the **IDiskQuotaControl::SetQuotaState** method to configure the system to deny additional disk space to the user when the disk space charged to the user exceeds this value.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in dskquota.h.

See Also

File Systems Overview, *File System Structures*, **IDiskQuotaControl::SetQuotaLogFlags**, **IDiskQuotaControl::SetQuotaState**, **IDiskQuotaUser::GetQuotaInformation**

EFS_CERTIFICATE_BLOB

The **EFS_CERTIFICATE_BLOB** structure contains a certificate.

```
typedef struct _CERTIFICATE_BLOB {
  DWORD dwCertEncodingType;
  DWORD cbData;
  PBYTE pbData;
} EFS_CERTIFICATE_BLOB, *PEFS_CERTIFICATE_BLOB;
```

Members

dwCertEncodingType

Specifies the certificate encoding type. This member can be one of the following values.

CRYPT_ASN_ENCODING
CRYPT_NDR_ENCODING
X509_ASN_ENCODING
X509_NDR_ENCODING

cbData

Specifies the number of bytes in the **pbData** buffer.

pbData

A binary certificate. The format of this certificate is specified by the **dwCertEncodingType** member.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.

See Also

File Systems Overview, *File System Structures*, **ENCRYPTION_CERTIFICATE**

EFS_HASH_BLOB

The **EFS_HASH_BLOB** structure contains a certificate hash.

```
typedef struct _EFS_HASH_BLOB {
  DWORD cbData;
  PBYTE pbData;
} EFS_HASH_BLOB, *PEFS_HASH_BLOB;
```

Members

cbData
Specifies the number of bytes in the **pbData** buffer.

pbData
The certificate hash.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.

See Also

File Systems Overview, *File System Structures*, **ENCRYPTION_CERTIFICATE_HASH**

ENCRYPTION_CERTIFICATE

The **ENCRYPTION_CERTIFICATE** structure contains a certificate.

```
typedef struct _ENCRYPTION_CERTIFICATE {
  DWORD cbTotalLength;
  SID *pUserSid;
  PEFS_CERTIFICATE_BLOB pCertBlob;
} ENCRYPTION_CERTIFICATE, *PENCRYPTION_CERTIFICATE;
```

Members

cbTotalLength
The length of this structure, in bytes.

pUserSid
The **SID** of the user who owns the certificate.

pCertBlob
Pointer to an **EFS_CERTIFICATE_BLOB** structure.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.

See Also

File Systems Overview, *File System Structures*, **EFS_CERTIFICATE_BLOB**, **ENCRYPTION_CERTIFICATE_LIST**, **SetUserFileEncryptionKey**

ENCRYPTION_CERTIFICATE_HASH

The **ENCRYPTION_CERTIFICATE_HASH** structure contains a certificate hash.

```
typedef struct _ENCRYPTION_CERTIFICATE_HASH {
  DWORD cbTotalLength;
  SID *pUserSid;
  PEFS_HASH_BLOB pHash;
  LPWSTR lpDisplayInformation;
} ENCRYPTION_CERTIFICATE_HASH, *PENCRYPTION_CERTIFICATE_HASH;
```

Members

cbTotalLength
The length of this structure, in bytes.

pUserSid
The **SID** of the user who owns the certificate.

pHash
Pointer to an **EFS_HASH_BLOB** structure.

lpDisplayInformation

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.

See Also

File Systems Overview, *File System Structures*, **EFS_HASH_BLOB**, **ENCRYPTION_CERTIFICATE_HASH_LIST**

ENCRYPTION_CERTIFICATE_HASH_LIST

The **ENCRYPTION_CERTIFICATE_HASH_LIST** structure is a list of certificate hashes.

```
typedef struct _ENCRYPTION_CERTIFICATE_HASH_LIST {
  DWORD nCert_Hash;
  PENCRYPTION_CERTIFICATE_HASH *pUsers;
} ENCRYPTION_CERTIFICATE_HASH_LIST, *PENCRYPTION_CERTIFICATE_HASH_LIST;
```

Members

nCert_Hash
Specifies the number of certificate hashes in the list.

pUsers
Pointer to the first **ENCRYPTION_CERTIFICATE_HASH** structure in the list.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.

See Also

File Systems Overview, *File System Structures*, **ENCRYPTION_CERTIFICATE_HASH**, **FreeEncryptionCertificateHashList**, **QueryRecoveryAgentsOnEncryptedFile**, **QueryUsersOnEncryptedFile**, **RemoveUsersFromEncryptedFile**

ENCRYPTION_CERTIFICATE_LIST

The **ENCRYPTION_CERTIFICATE_LIST** structure is a list of certificates.

```
typedef struct _ENCRYPTION_CERTIFICATE_LIST {
  DWORD nUsers;
  PENCRYPTION_CERTIFICATE *pUsers;
} ENCRYPTION_CERTIFICATE_LIST, *PENCRYPTION_CERTIFICATE_LIST;
```

Members

nUsers
Specifies the number of certificates in the list.

pUsers
Pointer to the first **ENCRYPTION_CERTIFICATE** structure in the list.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winefs.h; include windows.h.

See Also

File Systems Overview, *File System Structures*, **AddUsersToEncryptedFile**, **ENCRYPTION_CERTIFICATE**

File System Macros

IsReparseTagHighLatency

The **IsReparseTagHighLatency** macro determines whether a reparse point tag has its high-latency bit set, indicating a file or directory for which the operating system is expected to be slow to retrieve data.

```
ULONG IsReparseTagHighLatency(
  ULONG _tag  // tag to be tested
);
```

Parameters

_tag
Reparse point tag to be tested for high latency.

Return Values

The return value is a **ULONG** that must be treated as zero or nonzero. A nonzero return value means that the tag's high-latency bit is set. A zero return value means that the tag's high-latency bit is not set.

Remarks

If the tag has its high-latency bit set, expect the operating system to be slow to retrieve the first byte of file or directory data. An example of such a file is one residing on very slow mass storage. Your application should display some indication to users that retrieval is in progress.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.

Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

File Systems Overview, *File System Macros*

IsReparseTagMicrosoft

The **IsReparseTagMicrosoft** macro determines whether a reparse point tag indicates a Microsoft reparse point.

```
ULONG IsReparseTagMicrosoft(
  ULONG _tag  // tag to be tested
);
```

Parameters

_tag
Reparse point tag to be tested.

Return Values

The return value is a **ULONG** that must be treated as zero or nonzero. A nonzero return value indicates that the tag is a Microsoft tag. A zero return value indicates that the tag is not a Microsoft tag. Only software developed by Microsoft or in partnership with Microsoft can use Microsoft tags. All other software must use non-Microsoft tags.

Remarks

If the Microsoft tag bit is set, Microsoft provides the tag. All other tags must use zero for this bit.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

File Systems Overview, *File System Macros*

IsReparseTagNameSurrogate

The **IsReparseTagNameSurrogate** macro determines whether a tag's associated reparse point is a surrogate for another named entity, for example a mount volume point.

```
ULONG IsReparseTagNameSurrogate(
  ULONG _tag  // tag to be tested
);
```

Parameters

_tag
Reparse point tag to be tested for surrogacy.

Return Values

The return value is a **ULONG** that must be treated as zero or nonzero. A nonzero return value means that the tag indicates a surrogate reparse point. A zero return value means that the tag does not indicate a surrogate reparse point.

Remarks

If the surrogacy bit is set, the file or directory represents another named entity in the system, such as a volume mounted at this directory. For more information on volume mounting, see *Volume Mount Points and Mounting Volumes*.

Requirements

Windows NT/2000: Requires Windows 2000.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winnt.h; include windows.h.

File Systems Overview, *File System Macros*

Disk Quota Interface Error Codes

The following error codes are custom **HRESULT** values defined in DSKQUOTA.H. They are returned by the disk quota interfaces, along with some of the standard **HRESULT** values.

Error	Meaning
ERROR_ACCESS_DENIED	The caller has insufficient access rights.
ERROR_BAD_PATHNAME	The requested path name is invalid.
ERROR_FILE_NOT_FOUND	The specified user account cannot be located.

ERROR_INITIALIZED	The object has already been initialized. Multiple initialization is not allowed.
ERROR_INSUFFICIENT_BUFFER	There is insufficient memory to complete the operation.
ERROR_INVALID_NAME	The requested file path is invalid.
ERROR_INVALID_SID	The SID for the user is invalid.
ERROR_LOCK_FAILED	Failure to obtain an exclusive lock.
ERROR_NO_MORE_ITEMS	There are no more items in the enumeration.
ERROR_NO_QUOTAS_FS	The file system does not support quotas. Currently, the file system must be NTFS.
ERROR_NO_QUOTAS_FSVER	The file system version does not support quotas. Currently, the file system must be Windows 2000 NTFS or later.
ERROR_NO_SID_MAPPING	There is no mapping available for the SID.
ERROR_NOT_INITIALIZED	The object has not been initialized. Initialization must be completed before the operation can be performed.
ERROR_NOT_SUPPORTED	The operation or capability is not supported.
ERROR_PATH_NOT_FOUND	The requested file path was not found.
ERROR_USER_HAS_FILES	The user owns files on the volume.
ERROR_USER_UNKNOWN	The specified user is unknown.

CHAPTER 11

Structured Exception and Error Handling

Structured Exception Handling

An *exception* is an event that occurs during the execution of a program, and requires the execution of code outside the normal flow of control. There are two kinds of exceptions: hardware exceptions and software exceptions. *Hardware exceptions* are initiated by the CPU. They can result from the execution of certain instruction sequences, such as division by zero or an attempt to access an invalid memory address. *Software exceptions* are initiated explicitly by applications or the operating system. For example, the system can detect when an invalid parameter value is specified.

Structured exception handling is a mechanism for handling both hardware and software exceptions. Therefore, your code will handle hardware and software exceptions identically. Structured exception handling enables you to have complete control over the handling of exceptions, provides support for debuggers, and is usable across all programming languages and machines.

The system also supports *termination handling*, which enables you to ensure that whenever a guarded body of code is executed, a specified block of termination code is also executed. The termination code is executed regardless of how the flow of control leaves the guarded body. For example, a termination handler can guarantee that clean-up tasks are performed even if an exception or some other error occurs while the guarded body of code is being executed.

About Structured Exception Handling

The structured exception handling and termination handling mechanisms are integral parts of the system; they enable the system to be robust. You can use these mechanisms to create consistently robust and reliable applications.

Structured exception handling is made available primarily through compiler support. For example, the Microsoft 32-bit C/C++ Optimizing Compiler supports the **__try** keyword that identifies a guarded body of code, the **__except** keyword that identifies an exception handler, and the **__finally** keyword that identifies a termination handler. Although this overview uses examples specific to the Microsoft C/C++ compiler, other compilers provide this support as well.

Exception Handling

Exceptions can be initiated by hardware or software, and can occur in kernel-mode as well as user-mode code. Structured exception handling provides a single mechanism for the handling of kernel-mode and user-mode exceptions.

The execution of certain instruction sequences can result in exceptions that are initiated by hardware. For example, an access violation is generated by the hardware when a process attempts to read from or write to a virtual address to which it does not have the appropriate access.

Events that require exception handling may also occur during execution of a software routine (for example, when an invalid parameter value is specified). When this happens, a thread can initiate an exception explicitly by calling the **RaiseException** function. This function enables the calling thread to specify information that describes the exception.

An exception can be continuable or noncontinuable. A noncontinuable exception arises when the event is not continuable in the hardware, or if continuation makes no sense. A noncontinuable exception does not terminate the application. Therefore, an application may be able to catch the exception and run. However, a noncontinuable exception typically arises as a result of a corrupted stack or other serious problem, making it difficult to recover from the exception.

Exception Dispatching

When a hardware or software exception occurs, the processor stops execution at the point at which the exception occurred and transfers control to the system. First, the system saves both the machine state of the current thread and information that describes the exception. The system then attempts to find an exception handler to handle the exception.

The machine state of the thread in which the exception occurred is saved in a **CONTEXT** structure. This information (called the *context record*) enables the system to continue execution at the point of the exception if the exception is successfully handled. The description of the exception (called the *exception record*) is saved in an **EXCEPTION_RECORD** structure. Because it stores the machine-dependent information of the context record separately from the machine-independent information of the exception record, the exception-handling mechanism is portable to different platforms.

The information in both the context and exception records is available by means of the **GetExceptionInformation** function, and can be made available to any exception handlers that are executed as a result of the exception. The exception record includes the following information.

- An exception code that identifies the type of exception.
- Flags indicating whether the exception is continuable. Any attempt to continue execution after a noncontinuable exception generates another exception.
- A pointer to another exception record. This facilitates creation of a linked list of exceptions if nested exceptions occur.

- The address at which the exception occurred.
- An array of 32-bit arguments that provide additional information about the exception.

When an exception occurs in user-mode code, the system uses the following search order to find an exception handler:

1. If the process is being debugged, the system notifies the debugger. For more information, see *Debugger Exception Handling*.
2. If the process is not being debugged, or if the associated debugger does not handle the exception, the system attempts to locate a frame-based exception handler by searching the stack frames of the thread in which the exception occurred. The system searches the current stack frame first, then searches through preceding stack frames in reverse order.
3. If no frame-based handler can be found, or no frame-based handler handles the exception, but the process is being debugged, the system notifies the debugger a second time.
4. If the process is not being debugged, or if the associated debugger does not handle the exception, the system provides default handling based on the exception type. For most exceptions, the default action is to call the **ExitProcess** function.

When an exception occurs in kernel-mode code, the system searches the stack frames of the kernel stack in an attempt to locate an exception handler. If a handler cannot be located or no handler handles the exception, the system is shut down as if the **ExitWindows** function had been called.

Debugger Exception Handling

The system's handling of user-mode exceptions provides support for sophisticated debuggers. If the process in which an exception occurs is being debugged, the system generates a debug event. If the debugger is using the **WaitForDebugEvent** function, the debug event causes that function to return with a pointer to a **DEBUG_EVENT** structure. This structure contains the process and thread identifiers the debugger can use to access the thread's context record. The structure also contains an **EXCEPTION_DEBUG_INFO** structure that includes a copy of the exception record.

When the system is searching for an exception handler, it makes two attempts to notify a process's debugger. The first notification attempt provides the debugger with an opportunity to handle breakpoint or single-step exceptions. This is known as *first-chance notification*. The user can then issue debugger commands to manipulate the process's environment before any exception handlers are executed. The second attempt to notify the debugger occurs only if the system is unable to find a frame-based exception handler that handles the exception. This is known as *last-chance notification*. If the debugger does not handle the exception after the last-chance notification, the system terminates the process being debugged.

At each notification attempt, the debugger uses the **ContinueDebugEvent** function to return control to the system. Before returning control, the debugger can handle the

exception and modify the thread state as appropriate, or it can choose not to handle the exception. Using **ContinueDebugEvent**, the debugger can indicate that it has handled the exception, in which case the machine state is restored and thread execution is continued at the point at which the exception occurred. The debugger can also indicate that it did not handle the exception, which causes the system to continue its search for an exception handler.

Floating-Point Exceptions

By default, the system has all FP exceptions turned off. Therefore, computations result in NAN or INFINITY, rather than an exception. Before you can trap floating-point (FP) exceptions using structured exception handling, you must call the **_controlfp** C run-time library function as follows:

```
// Get the default control word

int cw = _controlfp( 0, 0 );

// Set the exception masks off, turn exceptions on
cw &=~(EM_OVERFLOW | EM_UNDERFLOW | EM_INEXACT | EM_ZERODIVIDE | EM_DENORMAL

// Set the control word

_controlfp( cw, MCW_EM );
```

This turns on all possible FP exceptions To trap only particular exceptions, use only the flags that correspond to the exceptions to be trapped. Note that any handler for FP errors should call **_clearfp** as its first FP instruction. This function clears floating-point exceptions.

Frame-Based Exception Handling

A *frame-based exception handler* allows you to deal with the possibility that an exception may occur in a certain sequence of code. A frame-based exception handler consists of the following elements.

- A guarded body of code
- A filter expression
- An exception-handler block

Frame-based exception handlers are declared in language-specific syntax. For example, in the Microsoft C/C++ Optimizing Compiler, they are implemented using **__try** and **__except**. For more information, see *Handler Syntax*.

The *guarded body of code* is a set of one or more statements for which the filter expression and the exception-handler block provide exception-handling protection. The guarded body can be a block of code, a set of nested blocks, or an entire procedure or function. Using the Microsoft C/C++ Optimizing Compiler, a guarded body is enclosed by braces ({}) following the **__try** keyword.

The *filter expression* of a frame-based exception handler is an expression that is evaluated by the system when an exception occurs within the guarded body. This evaluation results in one of the following actions by the system.

- The system stops its search for an exception handler, restores the machine state, and continues thread execution at the point at which the exception occurred.
- The system continues its search for an exception handler.
- The system transfers control to the exception handler, and thread execution continues sequentially in the stack frame in which the exception handler is found. If the handler is not in the stack frame in which the exception occurred, the system unwinds the stack, leaving the current stack frame and any other stack frames until it is back to the exception handler's stack frame. Before an exception handler is executed, termination handlers are executed for any guarded bodies of code that terminated as a result of the transfer of control to the exception handler. For more information about termination handlers, refer to *Termination Handling*.

The filter expression can be a simple expression, or it can invoke a *filter function* that attempts to handle the exception. You can call the **GetExceptionCode** and **GetExceptionInformation** functions from within a filter expression to get information about the exception being filtered. **GetExceptionCode** returns a code that identifies the type of exception, and **GetExceptionInformation** returns a pointer to an **EXCEPTION_POINTERS** structure that contains pointers to **CONTEXT** and **EXCEPTION_RECORD** structures.

These functions cannot be called from within a filter function, but their return values can be passed as parameters to a filter function. **GetExceptionCode** can be used within the exception-handler block, but **GetExceptionInformation** cannot because the information it points to is typically on the stack and is destroyed when control is transferred to an exception handler. However, an application can copy the information to safe storage to make it available to the exception handler.

The advantage of a filter function is that it can handle an exception and return a value that causes the system to continue execution from the point at which the exception occurred. With an exception-handler block, in contrast, execution continues sequentially from the exception handler rather than from the point of the exception.

Handling an exception may be as simple as noting an error and setting a flag that will be examined later, printing a warning or error message, or taking some other limited action. If execution can be continued, it may also be necessary to change the machine state by modifying the context record. For an example of a filter function that handles a page fault exception, see *Using the Virtual Memory Functions*.

The **UnhandledExceptionFilter** function can be used as a filter function in a filter expression. It returns EXCEPTION_EXECUTE_HANDLER unless the process is being debugged, in which case it returns EXCEPTION_CONTINUE_SEARCH.

Termination Handling

A *termination handler* ensures that a specific block of code is executed whenever flow of control leaves a particular guarded body of code. A termination handler consists of the following elements.

- A guarded body of code
- A block of termination code to be executed when the flow of control leaves the guarded body

Termination handlers are declared in language-specific syntax. Using the Microsoft C/C++ Optimizing Compiler, they are implemented using **__try** and **__finally**. For more information, see *Handler Syntax*.

The guarded body of code can be a block of code, a set of nested blocks, or an entire procedure or function. Whenever the guarded body is executed, the block of termination code will be executed. The only exception to this is when the thread terminates during execution of the guarded body (for example, if the **ExitThread** or **ExitProcess** function is called from within the guarded body).

The termination block is executed when the flow of control leaves the guarded body, regardless of whether the guarded body terminated normally or abnormally. The guarded body is considered to have terminated normally when the last statement in the block is executed and control proceeds sequentially into the termination block. Abnormal termination occurs when the flow of control leaves the guarded body due to an exception, or due to a control statement such as **return**, **goto**, **break**, or **continue**. The **AbnormalTermination** function can be called from within the termination block to determine whether the guarded body terminated normally.

Handler Syntax

This section describes the syntax and usage of structured exception handling as implemented in the Microsoft C/C++ Optimizing Compiler. The following keywords are interpreted by the compiler as part of the structured exception-handling mechanism.

Keyword	Description
__try	Begins a guarded body of code. Used with the **__except** keyword to construct an exception handler, or with the **__finally** keyword to construct a termination handler.
__except	Begins a block of code that is executed only when an exception occurs within its associated **__try** block.
__finally	Begins a block of code that is executed whenever the flow of control leaves its associated **__try** block.
__leave	Allows for immediate termination of the **__try** block without causing abnormal termination and its performance penalty.

The compiler also interprets the **GetExceptionCode**, **GetExceptionInformation**, and **AbnormalTermination** functions as keywords, and their use outside the appropriate

exception-handling syntax generates a compiler error. The following are brief descriptions of these functions.

Function	Description
GetExceptionCode	Returns a code that identifies the type of exception. This function can be called only from within the filter expression or the exception-handler block.
GetExceptionInformation	Returns a pointer to an **EXCEPTION_POINTERS** structure containing pointers to the context record and the exception record. This function can be called only from within the filter expression of an exception handler.
AbnormalTermination	Indicates whether the flow of control left the associated **__try** block sequentially after executing the last statement in the block. This function can be called only from within the **__finally** block of a termination handler.

Exception-Handler Syntax

The **__try** and **__except** keywords are used to construct a frame-based exception handler. The following example shows the structure of an exception handler.

```
__try
{
    // guarded body of code

}
__except (filter-expression)
{
    // exception-handler block

}
```

Note that the **__try** block and the exception-handler block require braces ({}). Using a **goto** statement to jump into the body of a **__try** block or into an exception-handler block is not permitted. This rule applies to both exception handlers and termination handlers.

The **__try** block contains the guarded body of code that the exception handler protects. A function can have any number of exception handlers, and these exception-handling statements can be nested within the same function or in different functions. If an exception occurs within the **__try** block, the system takes control and begins the search for an exception handler. For a detailed description of this search, see *Exception Handling.*

The exception handler receives only exceptions that occur within a single thread. This means that if a **__try** block contains a call to the **CreateProcess** or **CreateThread** function, exceptions that occur within the new process or thread are not dispatched to this handler.

The system evaluates the filter expression of each exception handler guarding the code in which the exception occurred until either the exception is handled or there are no more handlers. A filter expression must be evaluated as one of the three following values.

Value	Meaning
EXCEPTION_EXECUTE_HANDLER	The system transfers control to the exception handler, and execution continues in the stack frame in which the handler is found.
EXCEPTION_CONTINUE_SEARCH	The system continues to search for a handler.
EXCEPTION_CONTINUE_EXECUTION	The system stops its search for a handler and returns control to the point at which the exception occurred. If the exception is noncontinuable, this results in an EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

The filter expression is evaluated in the context of the function in which the exception handler is located, even though the exception may have occurred in a different function. This means that the filter expression can access the function's local variables. Similarly, the exception-handler block can access the local variables of the function in which it is located.

For more information about filter expressions and filter functions, see *Frame-Based Exception Handling*.

Termination-Handler Syntax

The **__try** and **__finally** keywords are used to construct a termination handler. The following example shows the structure of a termination handler.

```
__try
{
    // guarded body of code

}
__finally
{
    // __finally block

}
```

As with the exception handler, both the **__try** block and the **__finally** block require braces ({}), and using a **goto** statement to jump into either block is not permitted.

The __**try** block contains the guarded body of code that is protected by the termination handler. A function can have any number of termination handlers, and these termination-handling blocks can be nested within the same function or in different functions.

The __**finally** block is executed whenever the flow of control leaves the __**try** block. However, the __**finally** block is not executed if you call any of the following functions within the __**try** block: **ExitProcess**, **ExitThread**, or **abort**.

The __**finally** block is executed in the context of the function in which the termination handler is located. This means that the __**finally** block can access that function's local variables. Execution of the __**finally** block can terminate by any of the following means.

- Execution of the last statement in the block and continuation to the next instruction
- Use of a control statement (**return**, **break**, **continue**, or **goto**)
- Use of **longjmp** or a jump to an exception handler

If execution of the __**try** block terminates because of an exception that invokes the exception-handling block of a frame-based exception handler, the __**finally** block is executed before the exception-handling block is executed. Similarly, a call to the **longjmp** C run-time library function from the __**try** block causes execution of the __**finally** block before execution resumes at the target of the **longjmp** operation. If __**try** block execution terminates due to a control statement (**return**, **break**, **continue**, or **goto**), the __**finally** block is executed.

The **AbnormalTermination** function can be used within the __**finally** block to determine whether the __**try** block terminated sequentially—that is, whether it reached the closing brace (}). Leaving the __**try** block because of a call to **longjmp**, a jump to an exception handler, or a **return**, **break**, **continue**, or **goto** statement, is considered an abnormal termination. Note that failure to terminate sequentially causes the system to search through all stack frames in reverse order to determine whether any termination handlers must be called. This can result in performance degradation due to the execution of hundreds of instructions.

To avoid abnormal termination of the termination handler, execution should continue to the end of the block. You can also execute the __**leave** statement. The __**leave** statement allows for immediate termination of the __**try** block without causing abnormal termination and its performance penalty. Check your compiler documentation to determine if the __**leave** statement is supported.

If execution of the __**finally** block terminates because of the **return** control statement, it is equivalent to a **goto** to the closing brace in the enclosing function. Therefore, the enclosing function will return.

Structured Exception Handling Reference

AbnormalTermination

The **AbnormalTermination** function indicates whether the __**try** block of a termination handler terminated normally. The function can be called only from within the __**finally** block of a termination handler.

Note The Microsoft C/C++ Optimizing Compiler interprets this function as a keyword, and its use outside the appropriate exception-handling syntax generates a compiler error.

```
BOOL AbnormalTermination(VOID);
```

Parameters

This function has no parameters.

Return Values

If the __**try** block terminated abnormally, the return value is nonzero.

If the __**try** block terminated normally, the return value is zero.

Remarks

The __**try** block terminates normally only if execution leaves the block sequentially after executing the last statement in the block. Statements (such as **return**, **goto**, **continue**, or **break**) that cause execution to leave the __**try** block result in abnormal termination of the block. This is the case even if such a statement is the last statement in the __**try** block.

Abnormal termination of a __**try** block causes the system to search backward through all stack frames to determine whether any termination handlers must be called. This can result in the execution of hundreds of instructions, so it is important to avoid abnormal termination of a __**try** block due to a **return**, **goto**, **continue**, or **break** statement. Note that these statements do not generate an exception, even though the termination is abnormal.

To avoid abnormal termination, execution should continue to the end of the block. You can also execute the __**leave** statement. The __**leave** statement allows for immediate termination of the __**try** block without causing abnormal termination and its performance penalty. Check your compiler documentation to determine if the __**leave** statement is supported.

Requirements

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

See Also

Structured Exception Handling Overview, Structured Exception Handling Functions

GetExceptionCode

The **GetExceptionCode** function retrieves a code that identifies the type of exception that occurred. The function can be called only from within the filter expression or exception-handler block of an exception handler.

Note The Microsoft C/C++ Optimizing Compiler interprets this function as a keyword, and its use outside the appropriate exception-handling syntax generates a compiler error.

```
DWORD GetExceptionCode(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value identifies the type of exception. Following are the exception codes likely to occur due to common programming errors:

Value	Meaning
EXCEPTION_ACCESS_VIOLATION	The thread attempted to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_BREAKPOINT	A breakpoint was encountered.
EXCEPTION_DATATYPE_MISALIGNMENT	The thread attempted to read or write data that is misaligned on hardware that does not provide alignment. For example, 16-bit values must be aligned on 2-byte boundaries, 32-bit values on 4-byte boundaries, and so on.
EXCEPTION_SINGLE_STEP	A trace trap or other single-instruction mechanism signaled that one instruction has been executed.

(continued)

(continued)

Value	Meaning
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	The thread attempted to access an array element that is out of bounds, and the underlying hardware supports bounds checking.
EXCEPTION_FLT_DENORMAL_OPERAND	One of the operands in a floating-point operation is denormal. A denormal value is one that is too small to represent as a standard floating-point value.
EXCEPTION_FLT_DIVIDE_BY_ZERO	The thread attempted to divide a floating-point value by a floating-point divisor of zero.
EXCEPTION_FLT_INEXACT_RESULT	The result of a floating-point operation cannot be represented exactly as a decimal fraction.
EXCEPTION_FLT_INVALID_OPERATION	This exception represents any floating-point exception not included in this list.
EXCEPTION_FLT_OVERFLOW	The exponent of a floating-point operation is greater than the magnitude allowed by the corresponding type.
EXCEPTION_FLT_STACK_CHECK	The stack overflowed or underflowed as the result of a floating-point operation.
EXCEPTION_FLT_UNDERFLOW	The exponent of a floating-point operation is less than the magnitude allowed by the corresponding type.
EXCEPTION_INT_DIVIDE_BY_ZERO	The thread attempted to divide an integer value by an integer divisor of zero.
EXCEPTION_INT_OVERFLOW	The result of an integer operation caused a carry out of the most significant bit of the result.
EXCEPTION_PRIV_INSTRUCTION	The thread attempted to execute an instruction whose operation is not allowed in the current machine mode.
EXCEPTION_NONCONTINUABLE_EXCEPTION	The thread attempted to continue execution after a noncontinuable exception occurred.

Remarks

The **GetExceptionCode** function can be called only from within the filter expression or exception-handler block of an exception handler. The filter expression is evaluated if an exception occurs during execution of the **__try** block, and it determines whether the **__except** block is executed.

The filter expression can invoke a filter function. The filter function cannot call **GetExceptionCode**. However, the return value of **GetExceptionCode** can be passed as a parameter to a filter function. The return value of the **GetExceptionInformation**

function can also be passed as a parameter to a filter function. **GetExceptionInformation** returns a pointer to a structure that includes the exception-code information.

In the case of nested handlers, each filter expression is evaluated until one is evaluated as EXCEPTION_EXECUTE_HANDLER or EXCEPTION_CONTINUE_EXECUTION. Each filter expression can invoke **GetExceptionCode** to get the exception code.

The exception code returned is the code generated by a hardware exception, or the code specified in the **RaiseException** function for a software-generated exception.

When handling the breakpoint exception, it is important to increment the instruction pointer in the context record to continue from this exception.

Requirements

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

Structured Exception Handling Overview, *Structured Exception Handling Functions*, **GetExceptionInformation**, **RaiseException**

GetExceptionInformation

The **GetExceptionInformation** function retrieves a machine-independent description of an exception, and information about the machine state that existed for the thread when the exception occurred. This function can be called only from within the filter expression of an exception handler.

Note The Microsoft C/C++ Optimizing Compiler interprets this function as a keyword, and its use outside the appropriate exception-handling syntax generates a compiler error.

```
LPEXCEPTION_POINTERS GetExceptionInformation(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is a pointer to an **EXCEPTION_POINTERS** structure that contains pointers to two other structures: an **EXCEPTION_RECORD** structure containing a description of the exception, and a **CONTEXT** structure containing the machine-state information.

Remarks

The filter expression (from which the function is called) is evaluated if an exception occurs during execution of the **__try** block, and it determines whether the **__except** block is executed.

The filter expression can invoke a filter function. The filter function cannot call **GetExceptionInformation**. However, the return value of **GetExceptionInformation** can be passed as a parameter to a filter function.

To pass the **EXCEPTION_POINTERS** information to the exception-handler block, the filter expression or filter function must copy the pointer or the data to safe storage that the handler can later access.

In the case of nested handlers, each filter expression is evaluated until one is evaluated as EXCEPTION_EXECUTE_HANDLER or EXCEPTION_CONTINUE_EXECUTION. Each filter expression can invoke **GetExceptionInformation** to get exception information.

Requirements

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

See Also

Structured Exception Handling Overview, *Structured Exception Handling Functions*, **CONTEXT**, **EXCEPTION_POINTERS**, **EXCEPTION_RECORD**, **GetExceptionCode**

RaiseException

The **RaiseException** function raises an exception in the calling thread.

```
VOID RaiseException(
  DWORD dwExceptionCode,       // exception code
  DWORD dwExceptionFlags,      // continuable exception flag
  DWORD nNumberOfArguments,    // number of arguments
  CONST ULONG_PTR *lpArguments // array of arguments
);
```

Parameters

dwExceptionCode

[in] Specifies the application-defined exception code of the exception being raised. The filter expression and exception-handler block of an exception handler can use the **GetExceptionCode** function to retrieve this value.

Note that the system will clear bit 28 of *dwExceptionCode* before displaying a message This bit is a reserved exception bit, used by the system for its own purposes.

dwExceptionFlags
[in] Specifies the exception flags. This can be either zero to indicate a continuable exception, or EXCEPTION_NONCONTINUABLE to indicate a noncontinuable exception. Any attempt to continue execution after a noncontinuable exception causes the EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

nNumberOfArguments
[in] Specifies the number of arguments in the *lpArguments* array. This value must not exceed EXCEPTION_MAXIMUM_PARAMETERS. This parameter is ignored if *lpArguments* is NULL.

lpArguments
[in] Pointer to an array of arguments. This parameter can be NULL. These arguments can contain any application-defined data that needs to be passed to the filter expression of the exception handler.

Return Values

This function does not return a value.

Remarks

The **RaiseException** function enables a process to use structured exception handling to handle private, software-generated, application-defined exceptions.

Raising an exception causes the exception dispatcher to go through the following search for an exception handler:

1. The system first attempts to notify the process's debugger, if any.
2. If the process is not being debugged, or if the associated debugger does not handle the exception, the system attempts to locate a frame-based exception handler by searching the stack frames of the thread in which the exception occurred. The system searches the current stack frame first, then proceeds backward through preceding stack frames.
3. If no frame-based handler can be found, or no frame-based handler handles the exception, the system makes a second attempt to notify the process's debugger.
4. If the process is not being debugged, or if the associated debugger does not handle the exception, the system provides default handling based on the exception type. For most exceptions, the default action is to call the **ExitProcess** function.

The values specified in the *dwExceptionCode*, *dwExceptionFlags*, *nNumberOfArguments*, and *lpArguments* parameters can be retrieved in the filter expression of a frame-based exception handler by calling the **GetExceptionInformation** function. A debugger can retrieve these values by calling the **WaitForDebugEvent** function.

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Structured Exception Handling Overview, *Structured Exception Handling Functions*, **ExitProcess**, **GetExceptionCode**, **GetExceptionInformation**, **WaitForDebugEvent**

SetUnhandledExceptionFilter

The **SetUnhandledExceptionFilter** function lets an application supersede the top-level exception handler that Win32 places at the top of each thread and process.

After calling this function, if an exception occurs in a process that is not being debugged, and the exception makes it to the Win32 unhandled exception filter, that filter will call the exception filter function specified by the *lpTopLevelExceptionFilter* parameter.

```
LPTOP_LEVEL_EXCEPTION_FILTER
SetUnhandledExceptionFilter(
  LPTOP_LEVEL_EXCEPTION_FILTER lpTopLevelExceptionFilter
);
```

Parameters

lpTopLevelExceptionFilter

[in] Pointer to a top-level exception filter function that will be called whenever the **UnhandledExceptionFilter** function gets control, and the process is not being debugged. A value of NULL for this parameter specifies default handling within **UnhandledExceptionFilter**.

The filter function has syntax congruent to that of **UnhandledExceptionFilter**: It takes a single parameter of type **LPEXCEPTION_POINTERS**, and returns a value of type **LONG**. The filter function should return one of the following values.

Value	Meaning
EXCEPTION_EXECUTE_HANDLER	Return from **UnhandledExceptionFilter** and execute the associated exception handler. This usually results in process termination.

Value	Meaning
EXCEPTION_CONTINUE_EXECUTION	Return from **UnhandledExceptionFilter** and continue execution from the point of the exception. Note that the filter function is free to modify the continuation state by modifying the exception information supplied through its **LPEXCEPTION_POINTERS** parameter.
EXCEPTION_CONTINUE_SEARCH	Proceed with normal execution of **UnhandledExceptionFilter**. That means obeying the **SetErrorMode** flags, or invoking the Application Error pop-up message box.

Return Values

The **SetUnhandledExceptionFilter** function returns the address of the previous exception filter established with the function. A NULL return value means that there is no current top-level exception handler.

Remarks

Issuing **SetUnhandledExceptionFilter** replaces the existing top-level exception filter for all existing and all future threads in the calling process.

The exception handler specified by *lpTopLevelExceptionFilter* is executed in the context of the thread that caused the fault. This can affect the exception handler's ability to recover from certain exceptions, such as an invalid stack.

! 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.

Structured Exception Handling Overview, *Structured Exception Handling Functions*, **UnhandledExceptionFilter**

UnhandledExceptionFilter

The **UnhandledExceptionFilter** function passes unhandled exceptions to the debugger, if the process is being debugged. Otherwise, it optionally displays an **Application Error** message box and causes the exception handler to be executed. This function can be called only from within the filter expression of an exception handler.

```
LONG UnhandledExceptionFilter(
  STRUCT _EXCEPTION_POINTERS *ExceptionInfo
);
```

Parameters

ExceptionInfo

[in] Pointer to an **EXCEPTION_POINTERS** structure containing a description of the exception and the processor context at the time of the exception. This pointer is the return value of a call to the **GetExceptionInformation** function.

Return Values

The function returns one of the following values.

Value	Meaning
EXCEPTION_CONTINUE_SEARCH	The process is being debugged, so the exception should be passed (as second chance) to the application's debugger.
EXCEPTION_EXECUTE_HANDLER	If the SEM_NOGPFAULTERRORBOX flag was specified in a previous call to **SetErrorMode**, no Application Error message box is displayed. The function returns control to the exception handler, which is free to take any appropriate action.

Remarks

If the process is not being debugged, the function displays an Application Error message box, depending on the current error mode. The default behavior is to display the dialog box, but this can be disabled by specifying SEM_NOGPFAULTERRORBOX in a call to the **SetErrorMode** function.

The system uses **UnhandledExceptionFilter** internally to handle exceptions that occur during process and thread creation.

Requirements

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.

Structured Exception Handling Overview, *Structured Exception Handling Functions*, **EXCEPTION_POINTERS**, **GetExceptionInformation**, **SetErrorMode**, **SetUnhandledExceptionFilter**, **UnhandledExceptionFilter**

Structured Exception Handling Structures

EXCEPTION_POINTERS

The **EXCEPTION_POINTERS** structure contains an exception record with a machine-independent description of an exception and a context record with a machine-dependent description of the processor context at the time of the exception.

```
typedef struct _EXCEPTION_POINTERS {
  PEXCEPTION_RECORD ExceptionRecord;
  PCONTEXT ContextRecord;
} EXCEPTION_POINTERS, *PEXCEPTION_POINTERS;
```

Members

ExceptionRecord
Pointer to an **EXCEPTION_RECORD** structure that contains a machine-independent description of the exception.

ContextRecord
Pointer to a **CONTEXT** structure that contains a processor-specific description of the state of the processor at the time of the exception.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winnt.h; include windows.h.

See Also

Structured Exception Handling Overview, *Structured Exception Handling Structures*, **GetExceptionInformation**, **CONTEXT**, **EXCEPTION_RECORD**

EXCEPTION_RECORD

The **EXCEPTION_RECORD** structure describes an exception.

```
typedef struct _EXCEPTION_RECORD {
  DWORD ExceptionCode;
  DWORD ExceptionFlags;
  struct _EXCEPTION_RECORD *ExceptionRecord;
  PVOID ExceptionAddress;
  DWORD NumberParameters;
  ULONG_PTR ExceptionInformation[EXCEPTION_MAXIMUM_PARAMETERS];
} EXCEPTION_RECORD, *PEXCEPTION_RECORD;
```

Members

ExceptionCode

Specifies the reason the exception occurred. This is the code generated by a hardware exception, or the code specified in the **RaiseException** function for a software-generated exception. Following are the exception codes likely to occur due to common programming errors:

Value	Meaning
EXCEPTION_ACCESS_VIOLATION	The thread tried to read from or write to a virtual address for which it does not have the appropriate access.
EXCEPTION_ARRAY_BOUNDS_EXCEEDED	The thread tried to access an array element that is out of bounds and the underlying hardware supports bounds checking.
EXCEPTION_BREAKPOINT	A breakpoint was encountered.
EXCEPTION_DATATYPE_MISALIGNMENT	The thread tried to read or write data that is misaligned on hardware that does not provide alignment. For example, 16-bit values must be aligned on 2-byte boundaries; 32-bit values on 4-byte boundaries, and so on.
EXCEPTION_FLT_DENORMAL_OPERAND	One of the operands in a floating-point operation is denormal. A denormal value is one that is too small to represent as a standard floating-point value.
EXCEPTION_FLT_DIVIDE_BY_ZERO	The thread tried to divide a floating-point value by a floating-point divisor of zero.
EXCEPTION_FLT_INEXACT_RESULT	The result of a floating-point operation cannot be represented exactly as a decimal fraction.
EXCEPTION_FLT_INVALID_OPERATION	This exception represents any floating-point exception not included in this list.
EXCEPTION_FLT_OVERFLOW	The exponent of a floating-point operation is greater than the magnitude allowed by the corresponding type.
EXCEPTION_FLT_STACK_CHECK	The stack overflowed or underflowed as the result of a floating-point operation.
EXCEPTION_FLT_UNDERFLOW	The exponent of a floating-point operation is less than the magnitude allowed by the corresponding type.

EXCEPTION_ILLEGAL_INSTRUCTION	The thread tried to execute an invalid instruction.
EXCEPTION_IN_PAGE_ERROR	The thread tried to access a page that was not present, and the system was unable to load the page. For example, this exception might occur if a network connection is lost while running a program over the network.
EXCEPTION_INT_DIVIDE_BY_ZERO	The thread tried to divide an integer value by an integer divisor of zero.
EXCEPTION_INT_OVERFLOW	The result of an integer operation caused a carry out of the most significant bit of the result.
EXCEPTION_INVALID_DISPOSITION	An exception handler returned an invalid disposition to the exception dispatcher. Programmers using a high-level language such as C should never encounter this exception.
EXCEPTION_NONCONTINUABLE_EXCEPTION	The thread tried to continue execution after a noncontinuable exception occurred.
EXCEPTION_PRIV_INSTRUCTION	The thread tried to execute an instruction whose operation is not allowed in the current machine mode.
EXCEPTION_SINGLE_STEP	A trace trap or other single-instruction mechanism signaled that one instruction has been executed.
EXCEPTION_STACK_OVERFLOW	The thread used up its stack.

Another exception code is likely to occur when debugging console processes. It does not arise because of a programming error. The DBG_CONTROL_C exception code occurs when CTRL+C is input to a console process that handles CTRL+C signals and is being debugged. This exception code is not meant to be handled by applications. It is raised only for the benefit of the debugger, and is raised only when a debugger is attached to the console process.

ExceptionFlags
Specifies the exception flags. This member can be either zero, indicating a continuable exception, or EXCEPTION_NONCONTINUABLE indicating a noncontinuable exception. Any attempt to continue execution after a noncontinuable exception causes the EXCEPTION_NONCONTINUABLE_EXCEPTION exception.

ExceptionRecord
Pointer to an associated **EXCEPTION_RECORD** structure. Exception records can be chained together to provide additional information when nested exceptions occur.

ExceptionAddress
Specifies the address where the exception occurred.

NumberParameters
Specifies the number of parameters associated with the exception. This is the number of defined elements in the **ExceptionInformation** array.

ExceptionInformation
Specifies an array of additional 32-bit arguments that describe the exception. The **RaiseException** function can specify this array of arguments. For most exception codes, the array elements are undefined. For the following exception code, the array elements are defined as follows:

Exception code	Array contents
EXCEPTION_ACCESS_VIOLATION	The first element of the array contains a read-write flag that indicates the type of operation that caused the access violation. If this value is zero, the thread attempted to read the inaccessible data. If this value is 1, the thread attempted to write to an inaccessible address. The second array element specifies the virtual address of the inaccessible data.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winnt.h; include windows.h.

See Also

Structured Exception Handling Overview, *Structured Exception Handling Structures*, **EXCEPTION_DEBUG_INFO**, **EXCEPTION_POINTERS**, **GetExceptionInformation**, **RaiseException**, **UnhandledExceptionFilter**

Error Handling

Well-written applications include error-handling code that allows them to recover gracefully from unexpected errors. When an error occurs, the application may need to request user intervention, or it may be able to recover on its own. In extreme cases, the application may log the user off or shut down the system.

About Error Handling

The Win32 API provides functions that enable you to receive and display error information for your application.

Process Error Mode

Each process has an associated error mode that indicates to the system how the application is going to respond to serious errors. Serious errors include disk failure, drive-not-ready errors, data misalignment, and unhandled exceptions. An application can let the system display a message box informing the user that an error has occurred, or it can handle the errors. To handle these errors without user intervention, use the **SetErrorMode** function. After calling **SetErrorMode** and specifying appropriate flags, the system will not display the corresponding error message boxes.

Last-Error Code

When an error occurs, most functions in the Win32 API return an error code, usually zero, NULL, or –1. Many functions also set an internal error code called the *last-error code*. When a function succeeds, the last-error code is not reset. The error code is maintained separately for each running thread; an error in one thread does not overwrite the last-error code in another thread. An application can retrieve the last-error code by using the **GetLastError** function; the error code may tell more about what actually occurred to make the function fail.

The **SetLastError** function sets the error code for the current thread. The **SetLastErrorEx** function also allows the caller to set an error type indicating the severity of the error. These functions are intended primarily for dynamic-link libraries (DLL), so they can emulate the behavior of the Win32 API.

The system defines a set of error codes that can be set as last-error codes or be returned by these functions. Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you define error codes for your application, set this bit to indicate that the error code has been defined by an application and to ensure that the error codes do not conflict with any system-defined error codes. For more information, see *Error Codes*.

Notifying the User

To notify the user that some kind of error has occurred, many applications simply produce a sound by using the **Beep** or **MessageBeep** function or flash the window by using the **FlashWindow** or **FlashWindowEx** function. An application can also use these functions to call attention to an error and then display a message box or an error message containing details about the error.

Message Tables

Message tables are special string resources used when displaying error messages. They are declared in a resource file using the **MESSAGETABLE** resource-definition statement. To access the message strings, use the **FormatMessage** function.

The system provides a message table for the Win32 error codes. To retrieve the string that corresponds to the error code, call **FormatMessage** with the FORMAT_MESSAGE_FROM_SYSTEM flag.

To provide a message table for your application, follow the instructions in *About Message Text Files*. To retrieve strings from your message table, call **FormatMessage** with the FORMAT_MESSAGE_FROM_HMODULE flag.

Fatal Application Exit

The **FatalAppExit** function displays a message box and terminates the application when the user closes the message box. This function should only be used as a last resort, because it may not free the memory or files owned by the application.

Error Message Guidelines

The guidelines presented here are intended to help you write clearer and more useful error messages.

In particular, these guidelines address three common problems:

- Users often misunderstand popup errors.
- Administrators cannot easily understand event log messages.
- Technical support receives many calls as a result of confusing error messages.

Basic Guidelines

- **Do not anthropomorphize.** Do not imply that programs or hardware can think or feel. The following table shows an example.

Correct	Incorrect
Node [*node name*] cannot use Windows protocols.	Node [*node name*] does not speak any of our protocols.

- **Avoid the word "Bad."** Try to find a more descriptive term to tell the user what is wrong. For example, avoid messages such as the following:

```
bad link          // Avoid!
bad size          // Avoid!
bad argument      // Avoid!
bad address       // Avoid!
bad CRC           // Avoid!
```

- **Use full sentences.** For example, use "Binding is too long." instead of "Binding too long."
- **Place a word that is in the index at the start of each message.** Avoid starting a message with an article (*the*, *a*, or *an*). Never put a placeholder variable, such as *%1* or *%2* at the beginning of a message, because these are very difficult for a user to look up. Instead, write the message so that a word that is in the index is at the beginning and the placeholder is embedded in the message. The following table shows some examples.

Correct	Incorrect
Log file %1 is full.	%1 log file is full.
Listen failed in %1.	%1: Listen failed.
Computer name %1 is a domain controller of domain %2.	%1 is a domain controller of domain %2.

Searchable Messages

Place words that are both in the index and relevant to the central meaning at the beginning of the message string. The following table shows some examples.

Correct	Incorrect
Recycle Bin cannot store some of the items you are about to delete.	Some of the items you are about to delete cannot be stored in the Recycle Bin.
Endpoint was not found.	No endpoint was found.
Move you specified requires moving text cards.	That move requires moving text cards.
Computer name as entered is not valid.	The computer name you typed is invalid. Or Invalid computer name.
Chat application cannot start.	The application cannot start in Chat component.

Style Considerations

Users almost always prefer simple sentences that use simple present or past tense and active voice. The following table shows some examples.

Correct	Incorrect
Registry Editor cannot create the subkey.	Could not create the subkey.
Setup cannot start Program Manager.	Setup was unable to activate Program Manager.
Floppy disk sector ID field does not match floppy disk controller track address.	Mismatch between the floppy disk sector ID field and the floppy disk controller track address.
CHKDSK encountered an error during ...	An error was encountered during ...
Cannot find %1.	Could not find %1.
Modem does not respond.	Modem not responding.

However, using active voice and simple constructions is less important than making your message searchable. Use passive voice and more complex constructions if necessary to place search-relevant words at the beginning of the message. The following table shows some examples.

Correct	Incorrect
Log file size cannot be adjusted.	Cannot adjust the size of the log file. Or Was unable to adjust the size of the log file.
Printing of %1 cannot resume.	Cannot resume printing %1. Or Could not resume printing %1.

Message Length

Wordiness or verbosity of messages should depend on the component. Consider the following guidelines on message length:

- Strings from components that have the potential to destroy an installation or cause the system to crash should include more explanation and user guidance. For example, Setup and STOP messages can contain more text.
- Strings from components that are displayed often to less technical users (as opposed to administrators or technical support) can also be longer. For example, Print Manager and IIS messages can be longer.
- Strings that display mostly to technical users should be shorter. Explanation and user action fields in the Messages Help file can go into more detail for these users. For example, WINS and TCP/IP messages should be brief and to the point.

Dialog Box and Popup Message Guidelines

Users are often confused by messages that appear as dialog boxes or popup messages. When these appear on the screen and do not close until the user clicks the **OK** or **Cancel** button, users often feel they have "broken" something on the computer and are not sure what to do next.

When writing this type of message, consider the following:

- Does the message give the user a clear, non-technical explanation of the problem?
- Does the message include steps or a clear explanation of how to prevent the problem from occurring again?
- If a long explanation is needed, point the user to a help file topic to search for more information.
- If it is a critical problem, be sure to use the event log and write out the error.

Error Handling Reference

Error Handling Functions

Beep

The **Beep** function generates simple tones on the speaker. The function is synchronous; it does not return control to its caller until the sound finishes.

```
BOOL Beep(
  DWORD dwFreq,      // sound frequency
  DWORD dwDuration   // sound duration
);
```

Parameters

dwFreq

Windows NT/2000: [in] Specifies the frequency, in hertz, of the sound. This parameter must be in the range 37 through 32,767 (0x25 through 0x7FFF).

dwDuration

Windows NT/2000: [in] Specifies the duration, in milliseconds, of the sound.

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

Windows 95: The **Beep** function ignores the *dwFreq* and *dwDuration* parameters. On computers with a sound card, the function plays the default sound event. On computers without a sound card, the function plays the standard system beep.

Requirements

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.

See Also

Error Handling Overview, *Error Handling Functions*, **MessageBeep**

FatalAppExit

The **FatalAppExit** function displays a message box and terminates the application when the message box is closed. If the system is running with a kernel debugger, the message box gives the user the opportunity to terminate the application or to cancel the message box and return to the application that called **FatalAppExit**.

```
VOID FatalAppExit(
  UINT uAction,           // reserved
  LPCTSTR lpMessageText   // display string
);
```

Parameters

uAction

Reserved; must be zero.

lpMessageText

[in] Pointer to a null-terminated string that is displayed in the message box. The message is displayed on a single line. To accommodate low-resolution screens, the string should be no more than 35 characters in length.

Return Values

This function does not return a value.

Remarks

An application calls **FatalAppExit** only when it is not capable of terminating any other way. **FatalAppExit** may not always free an application's memory or close its files, and it may cause a general failure of the system. An application that encounters an unexpected error should terminate by freeing all its memory and returning from its main message loop.

Requirements

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

Error Handling Overview, *Error Handling Functions*, **FatalExit**

FlashWindow

The **FlashWindow** function flashes the specified window one time. It does not change the active state of the window.

To flash the window a specified number of times, use the **FlashWindowEx** function.

```
BOOL FlashWindow(
  HWND hWnd,      // handle to window
  BOOL bInvert    // flash status
);
```

Parameters

hWnd

[in] Handle to the window to be flashed. The window can be either open or minimized.

bInvert

[in] Specifies whether the window is to be flashed or returned to its original state. The window is flashed from one state to the other if this parameter is TRUE. If it is FALSE, the window is returned to its original state (either active or inactive). When an application is minimized, if this parameter is TRUE, the taskbar window button flashes active/inactive. If it is FALSE, the taskbar window button flashes inactive, meaning that it does not change colors. It flashes, as if it were being redrawn, but it does not provide the visual invert clue to the user.

Return Values

The return value specifies the window's state before the call to the **FlashWindow** function. If the window caption was drawn as active before the call, the return value is nonzero. Otherwise, the return value is zero.

Remarks

Flashing a window means changing the appearance of its caption bar as if the window were changing from inactive to active status, or vice versa. (An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.)

Typically, a window is flashed to inform the user that the window requires attention but that it does not currently have the keyboard focus.

The **FlashWindow** function flashes the window only once; for repeated flashing, the application should create a system timer.

Requirements

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.

Error Handling Overview, *Error Handling Functions*

FlashWindowEx

The **FlashWindowEx** function flashes the specified window. It does not change the active state of the window.

```
BOOL FlashWindowEx(
  PFLASHWINFO pfwi  // flash status information
);
```

Parameters

pfwi

[in] Pointer to the **FLASHWINFO** structure.

Return Values

The return value specifies the window's state before the call to the **FlashWindowEx** function. If the window caption was drawn as active before the call, the return value is nonzero. Otherwise, the return value is zero.

Remarks

Typically, you flash a window to inform the user that the window requires attention but does not currently have the keyboard focus. When a window flashes, it appears to change from inactive to active status. An inactive caption bar changes to an active caption bar; an active caption bar changes to an inactive caption bar.

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.

Error Handling Overview, *Error Handling Functions*, **FLASHWINFO**

FormatMessage

The **FormatMessage** function formats a message string. The function requires a message definition as input. The message definition can come from a buffer passed into the function. It can come from a message table resource in an already-loaded module. Or the caller can ask the function to search the system's message table resource(s) for the message definition. The function finds the message definition in a message table resource based on a message identifier and a language identifier. The function copies the formatted message text to an output buffer, processing any embedded insert sequences if requested.

```
DWORD FormatMessage(
  DWORD dwFlags,         // source and processing options
  LPCVOID lpSource,      // message source
  DWORD dwMessageId,     // message identifier
  DWORD dwLanguageId,    // language identifier
  LPTSTR lpBuffer,       // message buffer
  DWORD nSize,           // maximum size of message buffer
  va_list *Arguments     // array of message inserts
);
```

Parameters

dwFlags

[in] Specifies aspects of the formatting process and how to interpret the *lpSource* parameter. The low-order byte of *dwFlags* specifies how the function handles line breaks in the output buffer. The low-order byte can also specify the maximum width of a formatted output line.

You can specify a combination of the following values.

Value	Meaning
FORMAT_MESSAGE_ALLOCATE_BUFFER	Specifies that the *lpBuffer* parameter is a pointer to a **PVOID** pointer, and that the *nSize* parameter specifies the minimum number of **TCHARs** to allocate for an output message buffer. The function allocates a buffer large enough to hold the formatted message, and places a pointer to the allocated buffer at the address specified by *lpBuffer*. The caller should use the **LocalFree** function to free the buffer when it is no longer needed.
FORMAT_MESSAGE_IGNORE_INSERTS	Specifies that insert sequences in the message definition are to be ignored and passed through to the output buffer unchanged. This flag is useful for fetching a message for later formatting. If this flag is set, the *Arguments* parameter is ignored.

(continued)

(continued)

Value	Meaning
FORMAT_MESSAGE_FROM_STRING	Specifies that *lpSource* is a pointer to a null-terminated message definition. The message definition may contain insert sequences, just as the message text in a message table resource may. Cannot be used with FORMAT_MESSAGE_FROM_HMODULE or FORMAT_MESSAGE_FROM_SYSTEM.
FORMAT_MESSAGE_FROM_HMODULE	Specifies that *lpSource* is a module handle containing the message-table resource(s) to search. If this *lpSource* handle is NULL, the current process's application image file will be searched. Cannot be used with FORMAT_MESSAGE_FROM_STRING.
FORMAT_MESSAGE_FROM_SYSTEM	Specifies that the function should search the system message-table resource(s) for the requested message. If this flag is specified with FORMAT_MESSAGE_FROM_HMODULE, the function searches the system message table if the message is not found in the module specified by *lpSource*. Cannot be used with FORMAT_MESSAGE_FROM_STRING. If this flag is specified, an application can pass the result of the **GetLastError** function to retrieve the message text for a system-defined error.
FORMAT_MESSAGE_ARGUMENT_ARRAY	Specifies that the *Arguments* parameter is *not* a **va_list** structure, but instead is just a pointer to an array of values that represent the arguments.

The low-order byte of *dwFlags* can specify the maximum width of a formatted output line. Use the FORMAT_MESSAGE_MAX_WIDTH_MASK constant and bitwise Boolean operations to set and retrieve this maximum width value. The following table shows how **FormatMessage** interprets the value of the low-order byte.

Value	Meaning
0	There are no output line width restrictions. The function stores line breaks that are in the message definition text into the output buffer.
A nonzero value other than FORMAT_MESSAGE_MAX_WIDTH_MASK	The nonzero value is the maximum number of characters in an output line. The function ignores regular line breaks in the message definition text. The function never splits a string delimited by white space across a line break.

Value	Meaning
	The function stores hard-coded line breaks in the message definition text into the output buffer. Hard-coded line breaks are coded with the %n escape sequence.
FORMAT_MESSAGE_MAX_WIDTH_MASK	The function ignores regular line breaks in the message definition text. The function stores hard-coded line breaks in the message definition text into the output buffer. The function generates no new line breaks.

lpSource

[in] Specifies the location of the message definition. The type of this parameter depends upon the settings in the *dwFlags* parameter.

***dwFlags* Setting**	**Parameter Type**
FORMAT_MESSAGE_FROM_HMODULE	This parameter is a handle to the module that contains the message table to search.
FORMAT_MESSAGE_FROM_STRING	This parameter is a pointer to a string that consists of unformatted message text. It will be scanned for inserts and formatted accordingly.

If neither of these flags is set in *dwFlags*, then *lpSource* is ignored.

dwMessageId

[in] Specifies the message identifier for the requested message. This parameter is ignored if *dwFlags* includes FORMAT_MESSAGE_FROM_STRING.

dwLanguageId

[in] Specifies the language identifier for the requested message. This parameter is ignored if *dwFlags* includes FORMAT_MESSAGE_FROM_STRING.

If you pass a specific **LANGID** in this parameter, **FormatMessage** will return a message for that **LANGID** only. If the function cannot find a message for that **LANGID**, it returns ERROR_RESOURCE_LANG_NOT_FOUND. If you pass in zero, **FormatMessage** looks for a message for **LANGIDs** in the following order:

1. Language neutral
2. Thread **LANGID**, based on the thread's locale value
3. User default **LANGID**, based on the user's default locale value
4. System default **LANGID**, based on the system default locale value
5. US English

If **FormatMessage** doesn't find a message for any of the preceding **LANGIDs**, it returns any language message string that is present. If that fails, it returns ERROR_RESOURCE_LANG_NOT_FOUND.

lpBuffer

[out] Pointer to a buffer for the formatted (and null-terminated) message. If *dwFlags* includes FORMAT_MESSAGE_ALLOCATE_BUFFER, the function allocates a buffer using the **LocalAlloc** function, and places the pointer to the buffer at the address specified in *lpBuffer*.

nSize

[in] If the FORMAT_MESSAGE_ALLOCATE_BUFFER flag is not set, this parameter specifies the maximum number of **TCHARs** that can be stored in the output buffer. If FORMAT_MESSAGE_ALLOCATE_BUFFER is set, this parameter specifies the minimum number of **TCHARs** to allocate for an output buffer.

Arguments

[in] Pointer to an array of values that are used as insert values in the formatted message. A %1 in the format string indicates the first value in the *Arguments* array; a %2 indicates the second argument; and so on.

The interpretation of each value depends on the formatting information associated with the insert in the message definition. The default is to treat each value as a pointer to a null-terminated string.

By default, the *Arguments* parameter is of type **va_list***, which is a language- and implementation-specific data type for describing a variable number of arguments. If you do not have a pointer of type **va_list***, then specify the FORMAT_MESSAGE_ARGUMENT_ARRAY flag and pass a pointer to an array of values; those values are input to the message formatted as the insert values. Each insert must have a corresponding element in the array.

Windows 95: No single insertion string may exceed 1023 characters in length.

Return Values

If the function succeeds, the return value is the number of **TCHARs** stored in the output buffer, excluding the terminating null character.

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

Remarks

The **FormatMessage** function can be used to obtain error message strings for the system error codes returned by **GetLastError**.

Within the message text, several escape sequences are supported for dynamically formatting the message. These escape sequences and their meanings are shown in the following table. All escape sequences start with the percent character (%).

Escape sequence	Meaning
%0	Terminates a message text line without a trailing newline character. This escape sequence can be used to build up long lines or to terminate the message itself without a trailing newline character. It is useful for prompt messages.
%*n*!*printf format string*!	Identifies an insert. The value of *n* can be in the range 1 through 99. The **printf** format string (which must be bracketed by exclamation marks) is optional and defaults to **!s!** if not specified. The **printf** format string can contain the * specifier for either the precision or the width component. If * is specified for one component, the **FormatMessage** function uses insert %*n*+1; it uses %*n*+2 if * is specified for both components. Floating-point **printf** format specifiers—e, E, f, and g—are not supported. The workaround is to use the **sprintf** function to format the floating-point number into a temporary buffer, then use that buffer as the insert string.

Any other nondigit character following a percent character is formatted in the output message without the percent character. Following are some examples:

Format string	Resulting output
%%	A single percent sign in the formatted message text.
%n	A hard line break when the format string occurs at the end of a line. This format string is useful when **FormatMessage** is supplying regular line breaks so the message fits in a certain width.
%*space*	A space in the formatted message text. This format string can be used to ensure the appropriate number of trailing spaces in a message text line.
%.	A single period in the formatted message text. This format string can be used to include a single period at the beginning of a line without terminating the message text definition.
%!	A single exclamation point in the formatted message text. This format string can be used to include an exclamation point immediately after an insert without its being mistaken for the beginning of a **printf** format 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 winbase.h; include windows.h.
Library: Use kernel32.lib.
Unicode: Implemented as Unicode and ANSI versions on Windows NT/2000.

Error Handling Overview, *Error Handling Functions*, MESSAGETABLE Resource, Message Compiler

GetLastError

The **GetLastError** function returns the calling thread's last-error code value. The last-error code is maintained on a per-thread basis. Multiple threads do not overwrite each other's last-error code.

```
DWORD GetLastError(VOID);
```

Parameters

This function has no parameters.

Return Values

The return value is the calling thread's last-error code value. Functions set this value by calling the **SetLastError** function. The **Return Value** section of each reference page notes the conditions under which the function sets the last-error code.

Windows 95/98: Because **SetLastError** is a 32-bit function only, Win32 functions that are actually implemented in 16-bit code do not set the last-error code. You should ignore the last-error code when you call these functions. They include window management functions, GDI functions, and Multimedia functions.

Remarks

To obtain an error string for system error codes, use the **FormatMessage** function. For a complete list of error codes, see *Error Codes*.

You should call the **GetLastError** function immediately when a function's return value indicates that such a call will return useful data. That is because some functions call **SetLastError**(0) when they succeed, wiping out the error code set by the most recently failed function.

Most functions in the Win32 API that set the thread's last error code value set it when they fail; a few functions set it when they succeed. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call **SetLastError** under conditions of success; those cases are noted in each function's reference page.

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you are defining an error code for your application, set this bit to one. That indicates that the error code has been defined by an application, and ensures that your error code does not conflict with any error codes defined by the system.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development.*

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Error Handling Overview, *Error Handling Functions*, **FormatMessage**, **SetLastError**, **SetLastErrorEx**

MessageBeep

The **MessageBeep** function plays a waveform sound. The waveform sound for each sound type is identified by an entry in the registry.

```
BOOL MessageBeep(
  UINT uType   // sound type
);
```

Parameters

uType

[in] Specifies the sound type, as identified by an entry in the registry. This parameter can be one of the following values.

Value	Sound
–1	Standard beep using the computer speaker
MB_ICONASTERISK	SystemAsterisk
MB_ICONEXCLAMATION	SystemExclamation
MB_ICONHAND	SystemHand
MB_ICONQUESTION	SystemQuestion
MB_OK	SystemDefault

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

After queuing the sound, the **MessageBeep** function returns control to the calling function and plays the sound asynchronously.

If it cannot play the specified alert sound, **MessageBeep** attempts to play the system default sound. If it cannot play the system default sound, the function produces a standard beep sound through the computer speaker.

The user can disable the warning beep by using the Sound control panel application.

Requirements

Windows NT/2000: Requires Windows 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

Error Handling Overview, *Error Handling Functions*, **FlashWindow**, **MessageBox**

SetErrorMode

The **SetErrorMode** function controls whether the system will handle the specified types of serious errors, or whether the process will handle them.

```
UINT SetErrorMode(
  UINT uMode    // process error mode
);
```

Parameters

uMode

[in] Specifies the process error mode. This parameter can be one or more of the following values.

Value	Action
SEM_FAILCRITICALERRORS	The system does not display the critical-error-handler message box. Instead, the system sends the error to the calling process.
SEM_NOALIGNMENTFAULTEXCEPT	**RISC:** The system automatically fixes memory alignment faults and makes them invisible to the application. It does this for the calling process and any descendant processes. This flag has no effect on x86 processors.
SEM_NOGPFAULTERRORBOX	The system does not display the general-protection-fault message box. This flag should *only* be set by debugging applications that handle general protection (GP) faults themselves with an exception handler.
SEM_NOOPENFILEERRORBOX	The system does not display a message box when it fails to find a file. Instead, the error is returned to the calling process.

Return Values

The return value is the previous state of the error-mode bit flags.

Remarks

Each process has an associated error mode that indicates to the system how the application is going to respond to serious errors. A child process inherits the error mode of its parent process.

RISC: On some non-x86 processors misaligned memory references cause an alignment fault exception. The SEM_NOALIGNMENTFAULTEXCEPT flag lets you control whether the system automatically fixes such alignment faults, or makes them visible to an application.

MIPS: On a MIPS computer, an application must explicitly call **SetErrorMode** with the SEM_NOALIGNMENTFAULTEXCEPT flag to have the system automatically fix alignment faults. The default setting is for the system to make alignment faults visible to an application.

Alpha: On an ALPHA computer, you control the alignment fault behavior by setting the **EnableAlignmentFaultExceptions** value in the **HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Session Manager** registry key as follows.

Value	Meaning
0	Automatically fix alignment faults. This is the default.
1	Make alignment faults visible to the application. You must call **SetErrorMode** with SEM_NOALIGNMENTFAULTEXCEPT to have the system automatically fix alignment faults.
2	**Windows 2000:** Alignment faults are visible only when the process is running under the debugger.

x86: On an x86 computer, the system does not make alignment faults visible to an application. Therefore, specifying the SEM_NOALIGNMENTFAULTEXCEPT flag on an x86 computer is not an error, but the system is free to silently ignore and not properly preserve the flag. This means that code sequences such as the following are not always valid on x86 computers:

```
SetErrorMode(SEM_NOALIGNMENTFAULTEXCEPT);
fuOldErrorMode = SetErrorMode(0);
ASSERT(fuOldErrorMode == SEM_NOALIGNMENTFAULTEXCEPT);
```

Requirements

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.

See Also

Error Handling Overview, *Error Handling Functions*

SetLastError

The **SetLastError** function sets the last-error code for the calling thread.

```
VOID SetLastError(
  DWORD dwErrCode   // per-thread error code
);
```

Parameters

dwErrCode

[in] Specifies the last-error code for the thread.

Return Values

This function does not return a value.

Remarks

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no system error code has this bit set. If you are defining an error code for your application, set this bit to indicate that the error code has been defined by your application and to ensure that your error code does not conflict with any system-defined error codes.

This function is intended primarily for dynamic-link libraries (DLL). Calling this function after an error occurs lets the DLL emulate the behavior of the Win32 API.

Most Win32 functions call **SetLastError** when they fail. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call **SetLastError** under conditions of success; those cases are noted in each function's reference topic.

Applications can retrieve the value saved by this function by using the **GetLastError** function. The use of **GetLastError** is optional; an application can call it to find out the specific reason for a function failure.

The last-error code is kept in thread local storage so that multiple threads do not overwrite each other's 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 winbase.h; include windows.h.
Library: Use kernel32.lib.

See Also

Error Handling Overview, *Error Handling Functions*, **GetLastError**, **SetLastErrorEx**

SetLastErrorEx

The **SetLastErrorEx** function sets the last-error code.

Currently, this function is identical to the **SetLastError** function. The second parameter is ignored.

```
VOID SetLastErrorEx(
  DWORD dwErrCode,  // per-thread error code
  DWORD dwType      // error type
);
```

Parameters

dwErrCode

[in] Specifies the last-error code for the thread.

dwType

This parameter is ignored.

Return Values

This function does not return a value.

Remarks

Error codes are 32-bit values (bit 31 is the most significant bit). Bit 29 is reserved for application-defined error codes; no Win32 API error code has this bit set. If you are defining an error code for your application, set this bit to indicate that the error code has been defined by the application and to ensure that your error code does not conflict with any system-defined error codes.

This function is intended primarily for dynamic-link libraries (DLL). Calling this function after an error occurs allows the DLL to emulate the behavior of the Win32 API.

Most Win32 functions call **SetLastError** when they fail. Function failure is typically indicated by a return value error code such as zero, NULL, or –1. Some functions call **SetLastError** under conditions of success; those cases are noted in each function's reference topic.

Applications can retrieve the value saved by this function by using the **GetLastError** function.

The last-error code is kept in thread local storage so that multiple threads do not overwrite each other's values.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in winuser.h; include windows.h.
Library: Use user32.lib.

See Also

Error Handling Overview, *Error Handling Functions*, **GetLastError**

Error Handling Structures

FLASHWINFO

The **FLASHWINFO** structure contains the flash status for a window and the number of times the system should flash the window.

```
typedef struct {
  UINT  cbSize;
  HWND  hwnd;
  DWORD dwFlags;
  UINT  uCount;
  DWORD dwTimeout;
} FLASHWINFO, *PFLASHWINFO;
```

Members

cbSize
Specifies the size, in bytes, of the structure.

hwnd
Handle to the window to be flashed. The window can be either opened or minimized.

dwFlags
Specifies the flash status. This parameter can be one or more of the following flags.

Flag	Meaning
FLASHW_STOP	Stop flashing. The system restores the window to its original state.
FLASHW_CAPTION	Flash the window caption.
FLASHW_TRAY	Flash the taskbar button.
FLASHW_ALL	Flash both the window caption and taskbar button. This is equivalent to setting the FLASHW_CAPTION I FLASHW_TRAY flags.
FLASHW_TIMER	Flash continuously, until the FLASHW_STOP flag is set.
FLASHW_TIMERNOFG	Flash continuously until the window comes to the foreground.

uCount
Specifies the number of times to flash the window.

dwTimeout
Specifies the rate, in milliseconds, at which the window will be flashed. If **dwTimeout** is zero, the function uses the default cursor blink rate.

CHAPTER 12

Unicode

The Microsoft Win32 API provides support for the many different written languages of the international marketplace through Unicode and traditional character sets. Unicode is a worldwide character-encoding standard that uses 16-bit character values to represent all the characters used in modern computing, including technical symbols and special characters used in publishing. Traditional character sets are previous character-encoding standards—such as the Windows ANSI character set—that use 8-bit character values or combinations of 8-bit values to represent the characters used in a specific language or geographical region.

About Unicode and Character Sets

The world's character-based data was developed using both Unicode and traditional character sets. Because of this, the Microsoft Win32 API provides character-set functions that help Win32-based applications convert the character-based data from its original character set to Unicode or another traditional character set. These character-set functions also help Win32-based applications create character-based data that can be transferred to and used on any operating system, including those that do not support Unicode.

For details about Unicode beyond the scope of this overview, see *The Unicode Standard: Worldwide Character Encoding, Version 2.0* (Addison-Wesley, 1996).

Character Sets

A *character set* is a mapping of characters to their identifying numeric values. Most of the character sets commonly used in computers are single-byte character sets in which each character is identified by a value one byte wide. The large number of characters in Asian languages led to the development of multibyte character sets, in particular the double-byte character set (DBCS). Microsoft Windows NT/Windows 2000 incorporates a new global standard for character encoding: Unicode.

Single-Byte Character Sets

A single-byte character set is a mapping of 256 individual characters to their identifying numeric values. The character codes 0x20 through 0x7E represent standardized displayable characters, but the characters represented by the remaining codes vary among character sets. The ASCII character set covers the range 0x00 through 0x7F.

The ANSI character set is used in the window manager (User) and graphical device interface (GDI), but the Microsoft MS-DOS file allocation table (FAT) file system uses the

original equipment manufacturer (OEM) character set. Variations on the character sets, called *code pages*, include different special characters, typically customized for a language or group of languages. The OEM code page 437 is generally used in the United States.

Win32-based applications can use Unicode to avoid the inconsistencies of varied code pages and as an aid in developing easily localized applications.

An application can use the **GetACP** function to retrieve the ANSI code-page identifier for the system or use the **GetOEMCP** function to retrieve the OEM code-page identifier.

The **OemToChar** and **OemToCharBuff** functions allow an application to convert a character or string from the OEM code page to either the ANSI code page or Unicode. To convert in the other direction, you can use either the **CharToOem** or **CharToOemBuff** function. In addition, an application can use the **MultiByteToWideChar** and **WideCharToMultiByte** functions to map single-byte character set (SBCS) strings to Unicode and map Unicode strings to SBCS strings.

The **GetCPInfo** function fills a **CPINFO** structure with information that includes the size, in bytes, of the largest character in the code page and the default character used when a character code is entered that has no corresponding entry in the code page.

Double-Byte Character Sets

The double-byte character set (DBCS) is called an expanded 8-bit character set because its smallest unit is a byte. It can be thought of as the ANSI character set for some Asian versions of Microsoft Windows NT/
Windows 2000, Windows 95, and Windows 98 (particularly the Japanese versions). Win32 functions for the Japanese versions of Windows NT/Windows 2000, Windows 95, and Windows 98 accept DBCS strings for the ANSI versions of the functions. However, unlike the handling of Unicode, DBCS character handling requires detailed changes in the character-processing algorithms throughout an application's source code.

To help identify double-byte character sets, an application can use the **IsDBCSLeadByte** function to determine whether a given character is the first byte in a 2-byte character. In addition, an application can use the **MultiByteToWideChar** and **WideCharToMultiByte** functions to map DBCS strings to Unicode and map Unicode strings to DBCS strings.

Unicode

Unicode is a worldwide character-encoding standard. Windows NT/
Windows 2000 uses it exclusively at the system level for character and string manipulation. Unicode simplifies localization of software and improves multilingual text processing. By implementing it in your applications, you can enable the application with universal data exchange capabilities for global marketing, using a single binary file for every possible character code.

Unicode defines semantics for each character, standardizes script behavior, provides a standard algorithm for bidirectional text, and defines cross-mappings to other standards.

Among the scripts supported by Unicode are Latin, Greek, Han, Hiragana, and Katakana. Supported languages include, but are not limited to, German, French, English, Greek, Chinese, and Japanese.

Unicode can represent all the world's characters in modern computer use, including technical symbols and special characters used in publishing. Because each Unicode character is 16 bits wide, it is possible to have separate values for up to 65,536 characters. Unicode-enabled functions are often referred to as "wide-character" functions.

Win32 functions support applications that use either Unicode or the regular ANSI character set. Mixed use in the same application is also possible. Adding Unicode support to an application is easy, and you can even maintain a single set of sources from which to compile an application that supports either Unicode or the Windows ANSI character set.

Win32 functions support Unicode by assigning its strings a specific data type and providing a separate set of entry points and messages to support this new data type. A series of macros and naming conventions make transparent migration to Unicode, or even compiling both non-Unicode and Unicode versions of an application from the same set of sources, a straightforward matter.

Implementing Unicode as a separate data type also enables the compiler's type checking to ensure that only Unicode parameters are used with functions expecting Unicode strings.

For a list of functions that support Unicode under Windows 95/98, see *Windows 95/98 General Limitations*.

Surrogates

There is a need to support more characters than currently fit in the Unicode character set. For example, the Chinese speaking community uses at least 55,000 characters. To answer this need, the Unicode Standard defines surrogates. A *surrogate* or *surrogate pair* is a pair of 16-bit Unicode characters that represent a single character or glyph. The first (high) surrogate is a 16-bit character in the range U+D800 to U+DBFF. The second (low) surrogate is a 16-bit character in the range U+DC00 to U+DFFF. Using surrogates, Unicode can support over one million characters. For more details about surrogates, refer to the Unicode Standard, version 2.0.

Windows 2000 provides support for basic input, output, and simple sorting of surrogates. However, not all Windows 2000 system components are surrogate compatible. Surrogates ar not supported in Windows 95/98 or in Windows NT 4.0.

Windows 2000 supports surrogates in the following ways:

- The cmap 12 OpenType font format is introduced, which directly supports the 4-byte character code. Refer to the OpenType font specification for more detail.
- Windows USER supports surrogate-enabled IMEs.
- Windows GDI APIs support cmap 12 so surrogates can be displayed correctly.

- Uniscribe APIs support surrogates.
- Windows controls, including Edit and Rich Edit, support surrogates.
- HTML engine supports HTML page that includes surrogates for display, editing (through Outlook Express), and forms submission.
- System sorting table supports surrogates.
- Planes two and three (defined in ISO/IEC 10646) are reserved for ideographic characters. These planes fall in the high surrogate range of U+D840 to U+D8BF.

General Guidelines for Software Development

Windows 2000 handles surrogates as pairs of 16-bit characters. The system processes surrogate pairs in a way similar to the way it processes nonspacing marks. At display time, the surrogate pair display as one glyph. (This conforms to the requirements in the Unicode Standard, Version 2.0)

Applications automatically support surrogates if they use system controls and standard APIs, such as **ExtTextOut** and **DrawText**. Thus, if your code uses standard system controls or uses general **ExtTextOut**-type calls to display, surrogate pairs should work without any changes necessary.

Applications implementing their own editing support by working out glyph positions for themselves may use Uniscribe for all text processing. Uniscribe has separate APIs to deal with complex script processing (such as line service, hit testing, and cursor movement). The application must call the Uniscribe APIs specifically to get these advanced features. Applications written to the Uniscribe API are fully multilingual.

You can also write your own code to handle surrogate text processing. When a program encounters a separated Unicode value from either the lower reserved range or the upper reserved range, it must be one half of a surrogate pair. Thus, to detect a surrogate pair, do simple range checking. If you encounter a Unicode value in the lower or upper range, then you need to track backward or forward one 16-bit width to get the rest of the character. Keep in mind that **CharNext** and **CharPrev** move by 16-bit code points, not by surrogates.

For sorting, note that all surrogate pairs are treated as two Unicode code points. Surrogates are sorted after other Unicode code points, but before the PUA (private user area).

If you are a font or IME provider, note that Windows 2000 disables surrogate support by default. If you provide a font and IME package that requires surrogate support, you must set the following registry values:

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT \CurrentVersion\LanguagePack]

SURROGATE=(REG_DWORD)0x00000002

[HKEY_CURRENT_USER\Software\Microsoft\Internet Explorer\International\Scripts\42]

IEFixedFontName=[Surrogate Font Face Name]

IEPropFontName=[Surrogate Font Face Name]

Unicode in the Win32 API

Win32 API elements that use characters are generally implemented in one of three formats:

- A generic version that can be compiled for either ANSI or Unicode
- An ANSI version
- A Unicode version

The following topics discuss Unicode data types and how they are used in functions and messages; the use of resources, file names, and command-line arguments; and methods of translating between different types of strings.

Win32 Data Types
Win32 Function Prototypes
Message Translation
String Functions
Standard C Functions
Character Sets Used in Filenames
Translation Between String Types
Command-line Arguments

For a list of functions that support Unicode under Windows 95/98, see *Windows 95/98 General Limitations*.

Win32 Data Types

Most string operations for Unicode can be written by using the same logic used for handling the Windows ANSI character set, except that the basic unit of operation is a 16-bit character instead of an 8-bit byte character. The Platform SDK header files provide several type definitions that make it easy to create sources that can be compiled for Unicode or the ANSI character set.

The following example shows the method used in the Platform SDK header files to define three sets of data types: a set of generic type definitions that can compile for either ANSI or Unicode, and two sets of specific type definitions. The first set of specific type definitions is for use with the existing Windows (ANSI) character set, and the other is for use with Unicode (or wide) characters.

```
// Generic types

#ifdef UNICODE
    typedef wchar_t TCHAR;
```

(continued)

(continued)

```
#else
    typedef unsigned char TCHAR;
#endif

typedef TCHAR * LPTSTR, *LPTCH;

// 8-bit character specific

typedef unsigned char CHAR;
typedef CHAR *LPSTR, *LPCH;

// Unicode specific (wide characters)

typedef unsigned wchar_t WCHAR;
typedef WCHAR *LPWSTR, *LPWCH;
```

The letter *T* in a type definition designates a generic type that can be compiled for either ANSI or Unicode. The letter *W* in a type definition designates a wide-character (Unicode) type. For the actual implementation of this method, see the WinNT.h header file.

An application using generic data types can be compiled for Unicode simply by defining UNICODE before the include statements for the header files, or during compilation. To compile the code for ANSI, omit the UNICODE definition. It is best to use the generic data types, but the specific types exist for applications that require mixed types.

Win32 Function Prototypes

Win32 function prototypes are provided in generic, ANSI, and Unicode versions. The documentation provides generic function prototypes, which can be compiled to produce either ANSI or Unicode prototypes. As an example, all three prototypes are shown in the following code sample for the **SetWindowText** function.

Generic Prototype:

```
BOOL SetWindowText(
  HWND hwnd,
  LPCTSTR lpText
);
```

The header file provides the generic function name implemented as a macro.

```
#ifdef UNICODE
#define SetWindowText SetWindowTextW
#else
#define SetWindowText SetWindowTextA
#endif // !UNICODE
```

The preprocessor expands the macro into either the ANSI or Unicode function names, depending on whether UNICODE is defined. The letter *A* (ANSI) or *W* (wide) is added at

the end of the generic function name, as appropriate. The header file then provides ANSI and Unicode function prototypes, as shown in the following examples.

ANSI Prototype:

```
BOOL
WINAPI
SetWindowTextA(
    HWND hwnd,
    LPCSTR lpText );
```

Unicode Prototype:

```
BOOL
WINAPI
SetWindowTextW(
    HWND hwnd,
    LPCWSTR lpText );
```

Note that the generic function prototype uses the generic **LPCTSTR** for the text parameter, but the ANSI prototype uses **LPCSTR,** and the Unicode prototype uses **LPCWSTR**.

You can call the generic function in your application, then define UNICODE when you compile the code to use the Unicode function. To default to the ANSI function, do not define UNICODE. You can mix function calls by using the explicit function names ending with *A* and *W*.

This approach applies to all functions with text arguments. Always use a generic function prototype with the generic string and character types. All function names that end with an uppercase *W* take wide-character arguments. Some functions exist only in wide-character versions and can be used only with the appropriate data types.

The Requirements section in the documentation for each function provides information on the function versions implemented by the system. If there is a line that begins with **Unicode**, either the function has separate Unicode and ANSI versions, or the function accepts only Unicode strings. Otherwise, the function does not accept strings, or accepts only ANSI strings.

Note Whenever a function has a length parameter for a character string, the length should be documented as a count of **TCHAR** values in the string. However, functions that require or return pointers to untyped memory blocks, such as the **GlobalAlloc** function, are exceptions.

Message Translation

Although applications generally use the same window class, messages between windows of different classes are transparently translated by the system.

Even though a window function is implemented to receive messages in Unicode or ANSI format, the window procedure can still send messages or call functions of either type.

The following messages have text arguments and are subject to automatic text translation. (For information about automatic translation, see *Subclassing and Automatic Message Translation.*)

CB_ADDSTRING
CB_DIR
CB_FINDSTRING
CB_GETLBTEXT
CB_INSERTSTRING
CB_SELECTSTRING
EM_GETLINE
EM_REPLACESEL
EM_SETPASSWORDCHAR
LB_ADDFILE
LB_ADDSTRING
LB_DIR
LB_FINDSTRING
LB_GETTEXT
LB_INSERTSTRING
LB_SELECTSTRING
WM_ASKCBFORMATNAME
WM_CHAR
WM_CHARTOITEM
WM_CREATE
WM_DEADCHAR
WM_DEVMODECHANGE
WM_GETTEXT
WM_MDICREATE
WM_MENUCHAR
WM_NCCREATE
WM_SETTEXT
WM_SYSCHAR
WM_SYSDEADCHAR
WM_WININICHANGE

String Functions

All of the string functions listed in this section exist in ANSI and Unicode implementations to support ANSI and Unicode arguments. There are, however, subtle differences among some of them.

The following string functions do not require special comment; their ANSI and Unicode implementations work identically.

CharNext
CharPrev
lstrcat
lstrcpy
lstrlen

The value returned by the **lstrlen** function is always the number of characters, regardless of whether the ANSI or Unicode form is used.

The following string functions are sensitive to the locale of the current thread (as derived from the locale the user selects in Control Panel). The **lstrcmp** and **lstrcmpi** functions do not perform byte comparisons like their ANSI C namesakes; they compare strings according to the rules of the selected locale.

CharLower
CharLowerBuff
CharUpper

CharUpperBuff
lstrcmp
lstrcmpi

The following functions convert between the OEM character set and either ANSI or Unicode, depending on which version is used.

CharToOem
CharToOemBuff
OemToChar
OemToCharBuff

The print function **wsprintf** supports Unicode by providing the following new and changed data types in its format specifications. These format specifications affect the way the **wsprintf** function interprets the corresponding passed-in parameter.

Format specification	ANSI version	Unicode version
c	**CHAR**	**WCHAR**
C	**WCHAR**	**CHAR**
hc, **hC**	**CHAR**	**CHAR**
hs, **hS**	**LPSTR**	**LPSTR**
lc, **lC**	**WCHAR**	**WCHAR**
ls, **lS**	**LPWSTR**	**LPWSTR**
s	**LPSTR**	**LPWSTR**
S	**LPWSTR**	**LPSTR**

The data type for the output text always depends on the version of the function. Where the data type of the passed-in parameter and of the output text do not agree, **wsprintf** will perform a conversion from Unicode to ANSI, or vice versa, as required.

For the Unicode version of **wsprintf**, the format string is Unicode, as is the output text.

Standard C Functions

The standard C run-time libraries contain wide-character versions of the ANSI string functions that begin with the letters *str*. The wide-character versions of the functions start with the letters *wcs* (or sometimes *_wcs*). The Unicode data type is compatible with the wide-character data type **wchar_t** in ANSI C; this allows access to the wide-character string functions.

Generic functions exist for all standard C string functions. They start with the letters *_tcs* and are listed in the Tchar.h header file. These functions use the generic data types **TCHAR** and **TCHAR***

An application must add the following lines to its program in order to use the generic functions and compile for Unicode:

```
#define _UNICODE

#include <tchar.h>
#include <wchar.h>
```

Note that both the Tchar.h and Wchar.h are required, and that the leading underscore on the _UNICODE variable is also required.

The **wcstombs** and **mbstowcs** functions can convert from the character set supported by the standard C library to Unicode and back, with some limitations. For more information about translating strings to and from Unicode, see *Translation Between String Types*.

The **printf** function defined in Tchar.h supports the same format specifications as **wsprintf**; for details, see *String Functions*. Similarly, Tchar.h contains a **wprintf** function, in which the format string itself is a Unicode string.

Character Sets Used in Filenames

Window manager (User) and graphical device interface (GDI) use the ANSI character set; the MS-DOS FAT file system uses the OEM character set. Applications that create MS-DOS files sometimes have to use the **CharToOem** and **OemToChar** functions to translate between these character sets. However, NTFS is capable of storing file names in Unicode; no translation is necessary with NTFS.

With Unicode implementations of the file-system functions, it is not necessary to perform translations to and from ANSI and OEM character sets. Instead, you can use a single source file to compile non-Unicode versions of an application by providing macros for functions that are not invoked when compiling for Unicode, such as **CharToOem** and **OemToChar**.

The special file name characters in MS-DOS are unchanged in Unicode file names:

"\," "/," ".," "?," "*."

These special characters are in the ASCII range of characters (0x00 through 0x7F) and their Unicode equivalents are simply the same values in a 2-byte form: 0x0000 through 0x007F.

Translation Between String Types

The following Win32 functions translate character strings from one string type to another.

Function	Description
FoldString	Translates one character string to another.
LCMapString	Maps a character string by locale.
ToUnicode	Translates a virtual-key code into a Unicode character.
MultiByteToWideChar	Maps a multibyte string into a wide-character string.
WideCharToMultiByte	Maps a wide-character string into a multibyte string.

The **WideCharToMultiByte** and **MultiByteToWideChar** functions are particularly useful for applications that support several string types. ANSI C also defines the conversion functions **wcstombs** and **mbstowcs**, but they can only convert to and from the character set supported by the standard C library.

Command-Line Arguments

An application can use the **GetCommandLine** function to retrieve Unicode command-line arguments by calling it as a Unicode function.

Unicode and Character Set Reference

Unicode and Character Set Functions

GetTextCharset

The **GetTextCharset** function obtains a character-set identifier for the font that is currently selected into a specified device context.

The function call **GetTextCharset**(*hdc*) is equivalent to the function call **GetTextCharsetInfo**(*hdc*, NULL, 0).

```
int GetTextCharset(
  HDC hdc,   // handle to DC
);
```

Parameters

hdc

[in] Handle to a device context. The function obtains a character-set identifier for the font that is selected into this device context.

Return Values

If the function succeeds, the return value identifies the character set of the font that is currently selected into the specified device context. The following character-set identifiers are defined:

ANSI_CHARSET
BALTIC_CHARSET
CHINESEBIG5_CHARSET
DEFAULT_CHARSET
EASTEUROPE_CHARSET
GB2312_CHARSET
GREEK_CHARSET
HANGUL_CHARSET

MAC_CHARSET
OEM_CHARSET
RUSSIAN_CHARSET
SHIFTJIS_CHARSET
SYMBOL_CHARSET
TURKISH_CHARSET

Korean Windows:
JOHAB_CHARSET

Middle-Eastern Windows:
HEBREW_CHARSET
ARABIC_CHARSET

Thai Windows:
THAI_CHARSET

If the function fails, the return value is DEFAULT_CHARSET.

Requirements

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

See Also

Unicode and Character Sets Overview, *Unicode and Character Set Functions*, **GetTextCharsetInfo**

GetTextCharsetInfo

The **GetTextCharsetInfo** function obtains information about the character set of the font that is currently selected into a specified device context.

```
int GetTextCharsetInfo(
  HDC hdc,                  // handle to DC
  LPFONTSIGNATURE lpSig,    // data buffer
  DWORD dwFlags             // reserved; must be zero
);
```

Parameters

hdc

[in] Handle to a device context. The function obtains information about the font that is selected into this device context.

lpSig

[out] Pointer to a **FONTSIGNATURE** data structure that receives font-signature information.

If a TrueType font is currently selected into the device context, the **FONTSIGNATURE** structure receives information that identifies the code page and Unicode subranges for which the font provides glyphs.

If a font other than TrueType is currently selected into the device context, the **FONTSIGNATURE** structure receives zeroes. In this case, use the **TranslateCharsetInfo** function to obtain generic font-signature information for the character set.

The *lpSig* parameter can be NULL if you do not need the **FONTSIGNATURE** information. In this case, you can also call the **GetTextCharset** function, which is equivalent to calling **GetTextCharsetInfo** with *lpSig* set to NULL.

dwFlags

This parameter is reserved for future use. It must be set to zero.

Return Values

If the function succeeds, the return value identifies the character set of the font currently selected into the specified device context. The following character-set identifiers are defined:

ANSI_CHARSET
BALTIC_CHARSET
CHINESEBIG5_CHARSET
DEFAULT_CHARSET
EASTEUROPE_CHARSET
GB2312_CHARSET
GREEK_CHARSET
HANGUL_CHARSET
MAC_CHARSET
OEM_CHARSET
RUSSIAN_CHARSET
SHIFTJIS_CHARSET
SYMBOL_CHARSET
TURKISH_CHARSET

Korean Windows:

JOHAB_CHARSET

Middle-Eastern Windows:

HEBREW_CHARSET
ARABIC_CHARSET

Thai Windows:

THAI_CHARSET

If the function fails, the return value is DEFAULT_CHARSET.

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 wingdi.h; include windows.h.
Library: Use gdi32.lib.

See Also

Unicode and Character Sets Overview, *Unicode and Character Set Functions*, **FONTSIGNATURE**, **GetTextCharset**, **TranslateCharsetInfo**

IsDBCSLeadByte

The **IsDBCSLeadByte** function determines whether a character is a lead byte—that is, the first byte of a character in a double-byte character set (DBCS).

```
BOOL IsDBCSLeadByte(
  BYTE TestChar   // character to test
);
```

Parameters

TestChar
[in] Specifies the character to be tested.

Return Values

If the character is a lead byte, it returns a nonzero value.

If the character is not a lead byte, the return value is zero. To get extended error information, call **GetLastError**.

Remarks

Lead bytes are unique to double-byte character sets. A lead byte introduces a double-byte character. Lead bytes occupy a specific range of byte values. The **IsDBCSLeadByte** function uses the ANSI code page to check lead-byte ranges. To specify a different code page, use the **IsDBCSLeadByteEx** 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 winnls.h; include windows.h.
Library: Use kernel32.lib.

Unicode and Character Sets Overview, *Unicode and Character Set Functions*, **IsDBCSLeadByteEx**, **MultiByteToWideChar**

IsDBCSLeadByteEx

The **IsDBCSLeadByteEx** function determines whether a character is a lead byte that is, the first byte of a character in a double-byte character set (DBCS).

```
BOOL IsDBCSLeadByteEx(
  UINT CodePage,   // identifier of code page
  BYTE TestChar    // character to test
);
```

Parameters

CodePage

[in] Identifier of the code page to use to check lead-byte ranges. Can be one of the values given 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.

TestChar

[in] Character to test.

Return Values

If the function succeeds, it returns a nonzero value.

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

Requirements

Windows NT/2000: Requires Windows NT 3.51 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winnls.h; include windows.h.
Library: Use kernel32.lib.

Unicode and Character Sets Overview, *Unicode and Character Set Functions*

IsTextUnicode

The **IsTextUnicode** function determines whether a buffer is likely to contain a form of Unicode text. The function uses various statistical and deterministic methods to make its determination, under the control of flags passed via *lpi*. When the function returns, the results of such tests are reported via *lpi*.

```
DWORD IsTextUnicode(
  CONST LPVOID lpBuffer,  // input buffer to be examined
  int cb,                 // size of input buffer
  LPINT lpi               // options
);
```

Parameters

lpBuffer
[in] Pointer to the input buffer to be examined.

cb
[in] Specifies the size, in bytes, of the input buffer pointed to by *lpBuffer*.

lpi
[in/out] Pointer to an **int** variable that, upon entry to the function, contains a set of flags that specify the tests to be applied to the input buffer text. Upon exit from the function, that same variable contains a set of bit flags indicating the results of the specified tests: 1 if the contents of the buffer pass a test, 0 for failure. Only flags that are set upon entry to the function are significant upon exit.

If *lpi* is NULL, the function uses all available tests to determine whether the data in the buffer is likely to be Unicode text.

The following table shows the constants used with **lpi*'s bit flags.

Value	Meaning
IS_TEXT_UNICODE_ASCII16	The text is Unicode, and contains only zero-extended ASCII values/characters.
IS_TEXT_UNICODE_REVERSE_ASCII16	Same as the preceding, except that the Unicode text is byte-reversed.
IS_TEXT_UNICODE_STATISTICS	The text is probably Unicode, with the determination made by applying statistical analysis. Absolute certainty is not guaranteed. See the following Remarks section.
IS_TEXT_UNICODE_REVERSE_STATISTICS	Same as the preceding, except that the probably-Unicode text is byte-reversed.

IS_TEXT_UNICODE_CONTROLS	The text contains Unicode representations of one or more of these nonprinting characters: RETURN, LINEFEED, SPACE, CJK_SPACE, TAB.
IS_TEXT_UNICODE_REVERSE_CONTROLS	Same as the preceding, except that the Unicode characters are byte-reversed.
IS_TEXT_UNICODE_BUFFER_TOO_SMALL	There are too few characters in the buffer for meaningful analysis (fewer than two bytes).
IS_TEXT_UNICODE_SIGNATURE	The text contains the Unicode byte-order mark (BOM) 0xFEFF as its first character.
IS_TEXT_UNICODE_REVERSE_SIGNATURE	The text contains the Unicode byte-reversed byte-order mark (Reverse BOM) 0xFFFE as its first character.
IS_TEXT_UNICODE_ILLEGAL_CHARS	The text contains one of these Unicode-illegal characters: embedded Reverse BOM, UNICODE_NUL, CRLF (packed into one **WORD**), or 0xFFFF.
IS_TEXT_UNICODE_ODD_LENGTH	The number of characters in the string is odd. A string of odd length cannot (by definition) be Unicode text.
IS_TEXT_UNICODE_NULL_BYTES	The text contains null bytes, which indicate non-ASCII text.
IS_TEXT_UNICODE_UNICODE_MASK	This flag constant is a combination of IS_TEXT_UNICODE_ASCII16, IS_TEXT_UNICODE_STATISTICS, IS_TEXT_UNICODE_CONTROLS, IS_TEXT_UNICODE_SIGNATURE.
IS_TEXT_UNICODE_REVERSE_MASK	This flag constant is a combination of IS_TEXT_UNICODE_REVERSE_ASCII16, IS_TEXT_UNICODE_REVERSE_STATISTICS, IS_TEXT_UNICODE_REVERSE_CONTROLS, IS_TEXT_UNICODE_REVERSE_SIGNATURE.
IS_TEXT_UNICODE_NOT_UNICODE_MASK	This flag constant is a combination of IS_TEXT_UNICODE_ILLEGAL_CHARS, IS_TEXT_UNICODE_ODD_LENGTH, and two currently unused bit flags.
IS_TEXT_UNICODE_NOT_ASCII_MASK	This flag constant is a combination of IS_TEXT_UNICODE_NULL_BYTES and three currently unused bit flags.

Return Values

The function returns nonzero if the data in the buffer passes the specified tests.

The function returns zero if the data in the buffer does not pass the specified tests.

In either case, the **int** variable pointed to by *lpi* contains the results of the specific tests the function applied to make its determination.

Remarks

As noted in the preceding table of flag constants, the IS_TEXT_UNICODE_STATISTICS and IS_TEXT_UNICODE_REVERSE_STATISTICS tests use statistical analysis. These tests are not foolproof. The statistical tests assume certain amounts of variation between low and high bytes in a string, and some ASCII strings can slip through. For example, if *lpBuffer* points to the ASCII string 0x41, 0x0A, 0x0D, 0x1D (A\n\r^Z), the string passes the IS_TEXT_UNICODE_STATISTICS test, though failure would be preferable.

! Requirements

Windows NT/2000: Requires Windows NT 3.5 or later.
Windows 95/98: Unsupported.
Windows CE: Unsupported.
Header: Declared in winbase.h; include windows.h.
Library: Use advapi32.lib.

+ See Also

Unicode and Character Sets Overview, *Unicode and Character Set Functions*

MultiByteToWideChar

The **MultiByteToWideChar** function maps a character string to a wide-character (Unicode) string. The character string mapped by this function is not necessarily from a multibyte character set.

```
int MultiByteToWideChar(
  UINT CodePage,         // code page
  DWORD dwFlags,         // character-type options
  LPCSTR lpMultiByteStr, // string to map
  int cbMultiByte,       // number of bytes in string
  LPWSTR lpWideCharStr,  // wide-character buffer
  int cchWideChar        // size of buffer
);
```

Parameters

CodePage

[in] Specifies the code page to be used to perform the conversion. This parameter can be given the value of any code page that is installed or available in the system. You can also specify one of the values shown in the following table.

Value	Meaning
CP_ACP	ANSI code page
CP_MACCP	Macintosh code page
CP_OEMCP	OEM code page
CP_SYMBOL	**Windows 2000:** Symbol code page (42)
CP_THREAD_ACP	The current thread's ANSI code page
CP_UTF7	**Windows NT 4.0 and Windows 2000:** Translate using UTF-7
CP_UTF8	**Windows NT 4.0 and Windows 2000:** Translate using UTF-8. When this is set, *dwFlags* must be zero.

dwFlags

[in] Indicates whether to translate to precomposed or composite-wide characters (if a composite form exists), whether to use glyph characters in place of control characters, and how to deal with invalid characters. You can specify a combination of the following flag constants.

Value	Meaning
MB_PRECOMPOSED	Always use precomposed characters—that is, characters in which a base character and a nonspacing character have a single character value. This is the default translation option. Cannot be used with MB_COMPOSITE.
MB_COMPOSITE	Always use composite characters—that is, characters in which a base character and a nonspacing character have different character values. Cannot be used with MB_PRECOMPOSED.
MB_ERR_INVALID_CHARS	If the function encounters an invalid input character, it fails and **GetLastError** returns ERROR_NO_UNICODE_TRANSLATION.
MB_USEGLYPHCHARS	Use glyph characters instead of control characters.

A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a base/nonspacing character combination. In the character *è*, the *e* is the base character and the accent grave mark is the nonspacing character.

The function's default behavior is to translate to the precomposed form. If a precomposed form does not exist, the function attempts to translate to a composite form.

The flags MB_PRECOMPOSED and MB_COMPOSITE are mutually exclusive. The MB_USEGLYPHCHARS flag and the MB_ERR_INVALID_CHARS can be set regardless of the state of the other flags.

lpMultiByteStr

[in] Points to the character string to be converted.

cbMultiByte
[in] Specifies the size in bytes of the string pointed to by the *lpMultiByteStr* parameter. If this value is —1, the string is assumed to be null terminated and the length is calculated automatically. The length will include the null terminator.

lpWideCharStr
[out] Points to a buffer that receives the translated string.

cchWideChar
[in] Specifies the size, in wide characters, of the buffer pointed to by the *lpWideCharStr* parameter. If this value is zero, the function returns the required buffer size, in wide characters, and makes no use of the *lpWideCharStr* buffer.

Return Values

If the function succeeds, and *cchWideChar* is nonzero, the return value is the number of wide characters written to the buffer pointed to by *lpWideCharStr*.

If the function succeeds, and *cchWideChar* is zero, the return value is the required size, in wide characters, for a buffer that can receive the translated 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
ERROR_NO_UNICODE_TRANSLATION

Remarks

The *lpMultiByteStr* and *lpWideCharStr* pointers must not be the same. If they are the same, the function fails, and **GetLastError** returns the value ERROR_INVALID_PARAMETER.

The function fails if MB_ERR_INVALID_CHARS is set and encounters an invalid character in the source string. An invalid character is either, a) a character that is not the default character in the source string but translates to the default character when MB_ERR_INVALID_CHARS is *not* set, or b) for DBCS strings, a character which has a lead byte but no valid trailing byte. When an invalid character is found, and MB_ERR_INVALID_CHARS is set, the function returns 0 and sets **GetLastError** with the error ERROR_NO_UNICODE_TRANSLATION.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

! Requirements

Windows NT/2000: Requires Windows 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 and Character Sets Overview, *Unicode and Character Set Functions*, **WideCharToMultiByte**

TranslateCharsetInfo

The **TranslateCharsetInfo** function translates based on the specified character set, code page, or font signature value, setting all members of the destination structure to appropriate values.

```
BOOL TranslateCharsetInfo(
  DWORD *lpSrc,           // information
  LPCHARSETINFO lpCs,     // character set information
  DWORD dwFlags           // translation option
);
```

Parameters

lpSrc

[in/out] If *dwFlags* is TCI_SRCFONTSIG, this parameter is the address of the **fsCsb** member of a **FONTSIGNATURE** structure. Otherwise, this parameter is a **DWORD** value.

lpCs

[out] Pointer to a **CHARSETINFO** structure that receives the translated character set information.

dwFlags

[in] Specifies how to perform the translation. This parameter can be one of the following values.

Value	Meaning
TCI_SRCCHARSET	Source contains the character set value in the low word, and zero in the high word.
TCI_SRCCODEPAGE	Source is a code-page identifier in the low word and zero in the high word.

(continued)

(continued)

Value	Meaning
TCI_SRCFONTSIG	Source is the code-page bitfield portion of a **FONTSIGNATURE** structure. On input this should have only one Windows code-page bit set, either for an ANSI code-page value or for a common ANSI and OEM value (for OEM values, bits 32-63 must be clear.). On output this will have only one bit set. If the TCI_SRCFONTSIG value is given, the *lpSrc* parameter must be the address of the code-page bitfield. If any other TCI_ value is given, the *lpSrc* parameter must be a value not an address.

Return Values

If the function succeeds, it returns a nonzero value.

If the function fails, it returns zero. 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 2.10 or later.
Header: Declared in wingdi.h; include windows.h.
Library: Use gdi32.lib.

Unicode and Character Sets Overview, *Unicode and Character Set Functions*, **CHARSETINFO**, **FONTSIGNATURE**

WideCharToMultiByte

The **WideCharToMultiByte** function maps a wide-character string to a new character string. The new character string is not necessarily from a multibyte character set.

```
int WideCharToMultiByte(
  UINT CodePage,            // code page
  DWORD dwFlags,            // performance and mapping flags
  LPCWSTR lpWideCharStr,    // wide-character string
  int cchWideChar,          // number of chars in string
  LPSTR lpMultiByteStr,     // buffer for new string
  int cbMultiByte,          // size of buffer
  LPCSTR lpDefaultChar,     // default for unmappable chars
  LPBOOL lpUsedDefaultChar  // set when default char used
);
```

Parameters

CodePage

[in] Specifies the code page used to perform the conversion. This parameter can be given the value of any code page that is installed or available in the system. You can also specify one of the following values.

Value	Meaning
CP_ACP	ANSI code page
CP_MACCP	Macintosh code page
CP_OEMCP	OEM code page
CP_SYMBOL	**Windows 2000:** Symbol code page (42)
CP_THREAD_ACP	Current thread's ANSI code page
CP_UTF7	**Windows NT 4.0 and Windows 2000:** Translate using UTF-7
CP_UTF8	**Windows NT 4.0 and Windows 2000:** Translate using UTF-8. When this is set, *dwFlags* must be zero.

dwFlags

[in] Specifies the handling of unmapped characters. The function performs more quickly when none of these flags is set. The following flag constants are defined.

Value	Meaning
WC_NO_BEST_FIT_CHARS	**Windows 2000:** Any Unicode characters that do not translate directly to multibyte equivalents will be translated to the default character (see *lpDefaultChar* parameter). In other words, if translating from Unicode to multibyte and back to Unicode again does not yield the exact same Unicode character, the default character is used. This flag may be used by itself or in combination with the other *dwFlag* options.
WC_COMPOSITECHECK	Convert composite characters to precomposed characters.
WC_DISCARDNS	Discard nonspacing characters during conversion.
WC_SEPCHARS	Generate separate characters during conversion. This is the default conversion behavior.
WC_DEFAULTCHAR	Replace exceptions with the default character during conversion.

When WC_COMPOSITECHECK is specified, the function converts composite characters to precomposed characters. A composite character consists of a base character and a nonspacing character, each having different character values. A precomposed character has a single character value for a base/nonspacing character

combination. In the character è, the *e* is the base character, and the accent grave mark is the nonspacing character.

When an application specifies WC_COMPOSITECHECK, it can use the last three flags in this list (WC_DISCARDNS, WC_SEPCHARS, and WC_DEFAULTCHAR) to customize the conversion to precomposed characters. These flags determine the function's behavior when there is no precomposed mapping for a base/nonspace character combination in a wide-character string. These last three flags can only be used if the WC_COMPOSITECHECK flag is set.

The function's default behavior is to generate separate characters (WC_SEPCHARS) for unmapped composite characters.

lpWideCharStr
[in] Points to the wide-character string to be converted.

cchWideChar
[in] Specifies the number of wide characters in the string pointed to by the *lpWideCharStr* parameter. If this value is —1, the string is assumed to be null-terminated and the length is calculated automatically. The length will include the null-terminator.

lpMultiByteStr
[out] Points to the buffer to receive the translated string.

cbMultiByte
[in] Specifies the size, in bytes, of the buffer pointed to by the *lpMultiByteStr* parameter. If this value is zero, the function returns the number of bytes required for the buffer. (In this case, the *lpMultiByteStr* buffer is not used.)

lpDefaultChar
[in] Points to the character used if a wide character cannot be represented in the specified code page. If this parameter is NULL, a system default value is used. The function is faster when both *lpDefaultChar* and *lpUsedDefaultChar* are NULL.

lpUsedDefaultChar
[in] Points to a flag that indicates whether a default character was used. The flag is set to TRUE if one or more wide characters in the source string cannot be represented in the specified code page. Otherwise, the flag is set to FALSE. This parameter may be NULL. The function is faster when both *lpDefaultChar* and *lpUsedDefaultChar* are NULL.

Return Values

If the function succeeds, and *cbMultiByte* is nonzero, the return value is the number of bytes written to the buffer pointed to by *lpMultiByteStr*. The number includes the byte for the null terminator.

If the function succeeds, and *cbMultiByte* is zero, the return value is the required size, in bytes, for a buffer that can receive the translated 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 *lpMultiByteStr* and *lpWideCharStr* pointers must not be the same. If they are the same, the function fails, and **GetLastError** returns ERROR_INVALID_PARAMETER.

If *CodePage* is CP_SYMBOL and *cbMultiByte* is less than *cchWideChar,* no characters are written to *lpMultiByte.* Otherwise, if *cbMultiByte* is less than *cchWideChar*, *cbMultiByte* characters are copied to the buffer pointed to by *lpMultiByte.*

An application can use the *lpDefaultChar* parameter to change the default character used for the conversion.

As noted earlier, the **WideCharToMultiByte** function operates most efficiently when both *lpDefaultChar* and *lpUsedDefaultChar* are NULL. The following table shows the behavior of **WideCharToMultiByte** for the four combinations of *lpDefaultChar* and *lpUsedDefaultChar*.

lpDefaultChar	*lpUsedDefaultChar*	Result
NULL	NULL	No default checking. This is the most efficient way to use this function.
non-NULL	NULL	Uses the specified default character, but does not set *lpUsedDefaultChar*.
NULL	non-NULL	Uses the system default character and sets *lpUsedDefaultChar* if necessary.
non-NULL	non-NULL	Uses the specified default character and sets *lpUsedDefaultChar* if necessary.

MAPI: For more information, see *Syntax and Limitations for Win32 Functions Useful in MAPI Development*.

Requirements

Windows NT/2000: Requires Windows 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.

See Also

Unicode and Character Sets Overview, *Unicode and Character Set Functions*, **MultiByteToWideChar**

Unicode and Character Set Structures

CHARSETINFO

The **CHARSETINFO** structure contains information about a character set.

```
typedef struct {
    UINT ciCharset;
    UINT ciACP;
    FONTSIGNATURE fs;
} CHARSETINFO, *PCHARSETINFO;
```

Members

ciCharset
Character set value.

ciACP
ANSI code-page identifier.

fs
A **FONTSIGNATURE** structure that identifies the Unicode and code page—font signature values. Only one code page will be set when this structure is set by the 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 wingdi.h; include windows.h.

See Also

Unicode and Character Sets Overview, *Unicode and Character Set Structures*, **FONTSIGNATURE**, **TranslateCharsetInfo**

FONTSIGNATURE

The **FONTSIGNATURE** structure contains information identifying the code pages and Unicode subranges for which a given font provides glyphs.

```
typedef struct tagFONTSIGNATURE {
    DWORD  fsUsb[4];
    DWORD  fsCsb[2];
} FONTSIGNATURE, *PFONTSIGNATURE;
```

Members

fsUsb

A 128-bit Unicode subset bitfield (USB) identifying up to 126 Unicode subranges. Each bit, except the two most significant bits, represents a single subrange. The most significant bit is always 1 and identifies the bitfield as a font signature; the second most significant bit is reserved and must be 0. Unicode subranges are numbered in accordance with the ISO 10646 standard. For more information, see *Unicode Subset Bitfields.*

fsCsb

A 64-bit, code-page bitfield (CPB) that identifies a specific character set or code page. Code pages are in the lower 32 bits of this bitfield. The high 32 are used for non-Windows code pages. For more information, see Code-Page Bitfields.

Remarks

GDI relies on Windows code pages fitting within a 32-bit value. Furthermore, the highest two bits within this value are reserved for GDI internal use and may not be assigned to code pages.

! Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

+ See Also

Unicode and Character Sets Overview, *Unicode and Character Set Structures*, **LOCALESIGNATURE**

LOCALESIGNATURE

The **LOCALESIGNATURE** structure contains extended font-signature information, including two code-page bitfields (CPBs) that define the default and supported character sets and code pages. This structure is typically used to represent the relationships between font coverage and locales.

```
typedef struct tagLOCALESIGNATURE {
    DWORD  lsUsb[4];
    DWORD  lsCsbDefault[2];
    DWORD  lsCsbSupported[2];
} LOCALESIGNATURE, *PLOCALESIGNATURE;
```

Members

lsUsb

A 128-bit Unicode subset bitfield (USB) identifying up to 126 Unicode subranges. Each bit, except the two most significant bits, represents a single subrange. The most significant bit is always 1 and identifies the bitfield as a font signature; the second most significant is reserved and must be 0. Unicode subranges are numbered in accordance with the ISO 10646 standard.

lsCsbDefault

A code-page bitfield that indicates the default OEM and ANSI code pages for a locale. The code pages may be identified by separate bits or a single bit representing a common ANSI and OEM code page. For a list of possible bitfield values, see *Code-Page Bitfields*.

lsCsbSupported

A code-page bitfield that indicates all the code pages in which the locale can be supported. For a list of possible bitfield values, see *Code-Page Bitfields*.

Requirements

Windows NT/2000: Requires Windows NT 4.0 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Unsupported.
Header: Declared in wingdi.h; include windows.h.

See Also

Unicode and Character Sets Overview, Unicode and Character Set Structures

Unicode and Character Set Macros

The following macro is used in Unicode applications.

TEXT

The **TEXT** macro identifies a string as Unicode when the UNICODE is defined during compilation. Otherwise, it identifies a string as an ANSI string.

```
TEXT(
  LPTSTR string  // ANSI or Unicode string
);
```

Parameters

string

Pointer to the string to be interpreted as either Unicode or ANSI.

Requirements

Windows NT/2000: Requires Windows NT 3.1 or later.
Windows 95/98: Requires Windows 95 or later.
Windows CE: Requires version 1.0 or later.
Header: Declared in winnt.h; include windows.h.

See Also

Unicode and Character Sets Overview, Unicode and Character Set Macros

Unicode and Character Set Constants

The following groups of constants are used with the character set functions.

- ANSI Code-Page Identifiers
- OEM Code-Page Identifiers
- Code-Page Identifiers
- Code-Page Bitfields

ANSI Code-Page Identifiers

Identifier	Meaning
874	Thai
932	Japan
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1200	Unicode (BMP of ISO 10646)
1250	Windows 3.1 Eastern European
1251	Windows 3.1 Cyrillic
1252	Windows 3.1 Latin 1 (US, Western Europe)
1253	Windows 3.1 Greek
1254	Windows 3.1 Turkish
1255	Hebrew
1256	Arabic
1257	Baltic

OEM Code-Page Identifiers

Identifier	Meaning
437	MS-DOS United States

(continued)

(continued)

Identifier	Meaning
708	Arabic (ASMO 708)
709	Arabic (ASMO 449+, BCON V4)
710	Arabic (Transparent Arabic)
720	Arabic (Transparent ASMO)
737	Greek (formerly 437G)
775	Baltic
850	MS-DOS Multilingual (Latin I)
852	MS-DOS Slavic (Latin II)
855	IBM Cyrillic (primarily Russian)
857	IBM Turkish
860	MS-DOS Portuguese
861	MS-DOS Icelandic
862	Hebrew
863	MS-DOS Canadian-French
864	Arabic
865	MS-DOS Nordic
866	MS-DOS Russian (former USSR)
869	IBM Modern Greek
874	Thai
932	Japan
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1361	Korean (Johab)

Code-Page Identifiers

Identifier	Meaning
037	EBCDIC
437	MS-DOS United States
500	EBCDIC "500V1"
708	Arabic (ASMO 708)
709	Arabic (ASMO 449+, BCON V4)
710	Arabic (Transparent Arabic)
720	Arabic (Transparent ASMO)
737	Greek (formerly 437G)
775	Baltic

Identifier	Meaning
850	MS-DOS Multilingual (Latin I)
852	MS-DOS Slavic (Latin II)
855	IBM Cyrillic (primarily Russian)
857	IBM Turkish
860	MS-DOS Portuguese
861	MS-DOS Icelandic
862	Hebrew
863	MS-DOS Canadian-French
864	Arabic
865	MS-DOS Nordic
866	MS-DOS Russian
869	IBM Modern Greek
874	Thai
875	EBCDIC
932	Japan
936	Chinese (PRC, Singapore)
949	Korean
950	Chinese (Taiwan; Hong Kong SAR, PRC)
1026	EBCDIC
1200	Unicode (BMP of ISO 10646)
1250	Windows 3.1 Eastern European
1251	Windows 3.1 Cyrillic
1252	Windows 3.1 US (ANSI)
1253	Windows 3.1 Greek
1254	Windows 3.1 Turkish
1255	Hebrew
1256	Arabic
1257	Baltic
1361	Korean (Johab)
10000	Macintosh Roman
10001	Macintosh Japanese
10006	Macintosh Greek I
10007	Macintosh Cyrillic
10029	Macintosh Latin 2
10079	Macintosh Icelandic
10081	Macintosh Turkish

Code-Page Bitfields

Bit	Code page	Description
ANSI		
0	1252	Latin 1
1	1250	Latin 2: Eastern Europe
2	1251	Cyrillic
3	1253	Greek
4	1254	Turkish
5	1255	Hebrew
6	1256	Arabic
7	1257	Baltic
8 - 15		Reserved for ANSI
ANSI and OEM		
16	874	Thai
17	932	Japanese, Shift-JIS
18	936	Chinese: Simplified chars—PRC and Singapore
19	949	Korean Unified Hangeul Code (Hangeul TongHabHyung Code)
20	950	Chinese: Traditional chars—Taiwan and Hong Kong SAR, PRC
21	1361	Korean (Johab)
22 - 29		Reserved for alternate ANSI and OEM
30 - 31		Reserved by system.
OEM		
32 - 47		Reserved for OEM
48	869	IBM Greek
49	866	MS-DOS Russian
50	865	MS-DOS Nordic
51	864	Arabic
52	863	MS-DOS Canadian French
53	862	Hebrew
54	861	MS-DOS Icelandic
55	860	MS-DOS Portuguese
56	857	IBM Turkish
57	855	IBM Cyrillic; primarily Russian

Bit	Code page	Description
58	852	Latin 2
59	775	Baltic
60	737	Greek; former 437 G
61	708	Arabic; ASMO 708
62	850	Western European/Latin 1
63	437	US

Unicode Subset Bitfields

Bit	Description
0	Basic Latin
1	Latin-1 Supplement
2	Latin Extended-A
3	Latin Extended-B
4	IPA Extensions
5	Spacing Modifier Letters
6	Combining Diacritical Marks
7	Basic Greek
8	Greek Symbols and Coptic
9	Cyrillic
10	Armenian
11	Basic Hebrew
12	Hebrew Extended
13	Basic Arabic
14	Arabic Extended
15	Devanagari
16	Bengali
17	Gurmukhi
18	Gujarati
19	Oriya
20	Tamil
21	Telugu
22	Kannada
23	Malayalam
24	Thai
25	Lao
26	Basic Georgian

(continued)

(continued)

Bit	Description
27	Georgian Extended
28	Hangul Jamo
29	Latin Extended Additional
30	Greek Extended
31	General Punctuation
32	Subscripts and Superscripts
33	Currency Symbols
34	Combining Diacritical Marks for Symbols
35	Letter-like Symbols
36	Number Forms
37	Arrows
38	Mathematical Operators
39	Miscellaneous Technical
40	Control Pictures
41	Optical Character Recognition
42	Enclosed Alphanumerics
43	Box Drawing
44	Block Elements
45	Geometric Shapes
46	Miscellaneous Symbols
47	Dingbats
48	Chinese, Japanese, and Korean (CJK) Symbols and Punctuation
49	Hiragana
50	Katakana
51	Bopomofo
52	Hangul Compatibility Jamo
53	CJK Miscellaneous
54	Enclosed CJK
55	CJK Compatibility
56	Hangul
57	Reserved for Unicode Subranges
58	Reserved for Unicode Subranges
59	CJK Unified Ideographs
60	Private Use Area

Bit	Description
61	CJK Compatibility Ideographs
62	Alphabetic Presentation Forms
63	Arabic Presentation Forms-A
64	Combining Half Marks
65	CJK Compatibility Forms
66	Small Form Variants
67	Arabic Presentation Forms-B
68	Halfwidth and Fullwidth Forms
69	Specials
70-127	Reserved for Unicode Subranges

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.

Atom Reference 346
- Atom Functions 346
  - AddAtom
  - DeleteAtom
  - FindAtom
  - GetAtomName
  - GlobalAddAtom
  - GlobalDeleteAtom
  - GlobalFindAtom
  - GlobalGetAtomName
  - InitAtomTable
- Atom Macros 356
  - MAKEINTATOM

Clipboard Reference 363
- Clipboard Functions 363
  - ChangeClipboardChain
  - CloseClipboard
  - CountClipboardFormats
  - EmptyClipboard
  - EnumClipboardFormats
  - GetClipboardData
  - GetClipboardFormatName
  - GetClipboardOwner
  - GetClipboardSequenceNumber
  - GetClipboardViewer
  - GetOpenClipboardWindow
  - GetPriorityClipboardFormat
  - IsClipboardFormatAvailable
  - OpenClipboard
  - RegisterClipboardFormat
  - SetClipboardData
  - SetClipboardViewer
- Clipboard Structures 378
  - METAFILEPICT
- Clipboard Messages 380
  - WM_ASKCBFORMATNAME
  - WM_CHANGECBCHAIN
  - WM_CLEAR
  - WM_COPY
  - WM_CUT

Clipboard Messages *(continued)*
WM_DESTROYCLIPBOARD
WM_DRAWCLIPBOARD
WM_HSCROLLCLIPBOARD
WM_PAINTCLIPBOARD
WM_PASTE
WM_RENDERALLFORMATS
WM_RENDERFORMAT
WM_SIZECLIPBOARD
WM_VSCROLLCLIPBOARD
Dynamic Link Library Reference 217
Dynamic Link Library Functions 217
DisableThreadLibraryCalls
DllMain
FreeLibrary
FreeLibraryAndExitThread
GetModuleFileName
GetModuleHandle
GetProcAddress
LoadLibrary
LoadLibraryEx
Error Handling Reference 767
Error Handling Functions 767
Beep
FatalAppExit
FlashWindow
FlashWindowEx
FormatMessage
GetLastError
MessageBeep
SetErrorMode
SetLastError
SetLastErrorEx
Error Handling Structures 783
FLASHWINFO
File I/O Reference 481
File I/O Functions 481
AreFileApisANSI

File I/O Reference *(continued)*
File I/O Functions *(continued)*
CancelIo
CopyFile
CopyFileEx
CopyProgressRoutine
CreateDirectory
CreateDirectoryEx
CreateFile
CreateIoCompletionPort
DefineDosDevice
DeleteFile
FileIOCompletionRoutine
FindClose
FindCloseChangeNotification
FindFirstChangeNotification
FindFirstFile
FindFirstFileEx
FindNextChangeNotification
FindNextFile
FlushFileBuffers
GetBinaryType
GetCurrentDirectory
GetDiskFreeSpace
GetDiskFreeSpaceEx
GetDriveType
GetFileAttributes
GetFileAttributesEx
GetFileInformationByHandle
GetFileSize
GetFileSizeEx
GetFileType
GetFullPathName
GetLogicalDrives
GetLogicalDriveStrings
GetLongPathName
GetQueuedCompletionStatus
GetShortPathName
GetTempFileName

File I/O Reference *(continued)*
File I/O Functions *(continued)*
GetTempPath
Int32x32To64
Int64ShllMod32
Int64ShraMod32
Int64ShrlMod32
LockFile
LockFileEx
MoveFile
MoveFileEx
MoveFileWithProgress
MulDiv
PostQueuedCompletionStatus
QueryDosDevice
ReadDirectoryChangesW
ReadFile
ReadFileEx
ReadFileScatter
RemoveDirectory
ReplaceFile
SearchPath
SetCurrentDirectory
SetEndOfFile
SetFileApisToANSI
SetFileApisToOEM
SetFileAttributes
SetFilePointer
SetFilePointerEx
SetVolumeLabel
UInt32x32To64
UnlockFile
UnlockFileEx
WriteFile
WriteFileEx
WriteFileGather
File I/O Structures 606
BY_HANDLE_FILE_INFORMATION
FILE_NOTIFY_INFORMATION
File I/O Structures *(continued)*
LARGE_INTEGER
OFSTRUCT
ULARGE_INTEGER
WIN32_FILE_ATTRIBUTE_DATA
WIN32_FIND_DATA
File I/O Enumeration Types 617
FINDEX_INFO_LEVELS
FINDEX_SEARCH_OPS
GET_FILEEX_INFO_LEVELS
AddUsersToEncryptedFile
CreateHardLink
DecryptFile
DeleteVolumeMountPoint
EncryptFile
EncryptionDisable
FileEncryptionStatus
FindFirstVolume
FindFirstVolumeMountPoint
FindNextVolume
FindNextVolumeMountPoint
FindVolumeClose
FindVolumeMountPointClose
FreeEncryptionCertificateHashList
GetCompressedFileSize
GetVolumeInformation
GetVolumeNameForVolume MountPoint
GetVolumePathName
QueryRecoveryAgentsOn EncryptedFile
QueryUsersOnEncryptedFile
RemoveUsersFromEncryptedFile
SetUserFileEncryptionKey
SetVolumeMountPoint
File System Interfaces 683
IDiskQuotaControl
IDiskQuotaControl::AddUserName
IDiskQuotaControl::AddUserSid
File System Interfaces 683

File System Interfaces *(continued)*
IDiskQuotaControl::CreateEnumUsers
IDiskQuotaControl::CreateUserBatch
IDiskQuotaControl::DeleteUser
IDiskQuotaControl::FindUserName
IDiskQuotaControl::FindUserSid
IDiskQuotaControl::GetDefaultQuotaLimit
IDiskQuotaControl::GetDefaultQuotaLimitText
IDiskQuotaControl::GetDefaultQuotaThreshold
IDiskQuotaControl::GetDefaultQuotaThresholdText
IDiskQuotaControl::GetQuotaLogFlags
IDiskQuotaControl::GetQuotaState
IDiskQuotaControl::GiveUserNameResolutionPriority
IDiskQuotaControl::Initialize
IDiskQuotaControl::InvalidateSidNameCache
IDiskQuotaControl::SetDefaultQuotaLimit
IDiskQuotaControl::SetDefaultQuotaThreshold
IDiskQuotaControl::SetQuotaLogFlags
IDiskQuotaControl::SetQuotaState
IDiskQuotaControl::ShutdownNameResolution
IDiskQuotaEvents
IDiskQuotaEvents::OnUserNameChanged
IDiskQuotaUser
IDiskQuotaUser::GetAccountStatus
IDiskQuotaUser::GetID
IDiskQuotaUser::GetName

File System Interfaces *(continued)*
IDiskQuotaUser::GetQuotaInformation
IDiskQuotaUser::GetQuotaLimit
IDiskQuotaUser::GetQuotaLimitText
IDiskQuotaUser::GetQuotaThreshold
IDiskQuotaUser::GetQuotaThresholdText
IDiskQuotaUser::GetQuotaUsed
IDiskQuotaUser::GetQuotaUsedText
IDiskQuotaUser::GetSid
IDiskQuotaUser::GetSidLength
IDiskQuotaUser::Invalidate
IDiskQuotaUser::SetQuotaLimit
IDiskQuotaUser::SetQuotaThreshold
IDiskQuotaUserBatch
IDiskQuotaUserBatch::Add
IDiskQuotaUserBatch::Remove
IDiskQuotaUserBatch::RemoveAll
IDiskQuotaUserBatch::FlushToDisk
IEnumDiskQuotaUsers
IEnumDiskQuotaUsers::Clone
IEnumDiskQuotaUsers::Next
IEnumDiskQuotaUsers::Reset
IEnumDiskQuotaUsers::Skip

File System Structures 731
DISKQUOTA_USER_INFORMATION
EFS_CERTIFICATE_BLOB
EFS_HASH_BLOB
ENCRYPTION_CERTIFICATE
ENCRYPTION_CERTIFICATE_HASH
ENCRYPTION_CERTIFICATE_HASH_LIST
ENCRYPTION_CERTIFICATE_LIST

File System Macros 736
- IsReparseTagHighLatency
- IsReparseTagMicrosoft
- IsReparseTagNameSurrogate

Handle and Object Reference 404
- Handle and Object Functions 404
  - CloseHandle
  - DuplicateHandle
  - GetHandleInformation
  - SetHandleInformation

Hook Reference 420
- Hook Functions 420
  - CallMsgFilter
  - CallNextHookEx
  - CallWndProc
  - CallWndRetProc
  - CBTProc
  - DebugProc
  - ForegroundIdleProc
  - GetMsgProc
  - JournalPlaybackProc
  - JournalRecordProc
  - KeyboardProc
  - LowLevelKeyboardProc
  - LowLevelMouseProc
  - MessageProc
  - MouseProc
  - SetWindowsHookEx
  - ShellProc
  - SysMsgProc
  - UnhookWindowsHookEx
- Hook Structures 456
  - CBT_CREATEWND
  - CBTACTIVATESTRUCT
  - CWPRETSTRUCT
  - CWPSTRUCT
  - DEBUGHOOKINFO
  - EVENTMSG
  - KBDLLHOOKSTRUCT
- Hook Structures *(continued)*
  - MOUSEHOOKSTRUCT
  - MOUSEHOOKSTRUCTEX
  - MSLLHOOKSTRUCT
- Hook Messages 465
  - WM_CANCELJOURNAL
  - WM_QUEUESYNC

Interprocess Communications Reference 343
- Interprocess Communications Structures 343
  - COPYDATASTRUCT
- Interprocess Communications Messages 343
  - WM_COPYDATA

Memory Management Reference 261
  - AllocateUserPhysicalPages
  - CopyMemory
  - FillMemory
  - FreeUserPhysicalPages
  - GetProcessHeap
  - GetProcessHeaps
  - GetWriteWatch
  - GlobalMemoryStatus
  - HeapAlloc
  - HeapCompact
  - HeapCreate
  - HeapDestroy
  - HeapFree
  - HeapLock
  - HeapReAlloc
  - HeapSize
  - HeapUnlock
  - HeapValidate
  - HeapWalk
  - IsBadCodePtr
  - IsBadReadPtr
  - IsBadStringPtr
  - IsBadWritePtr
  - MapUserPhysicalPages

Memory Management Reference *(continued)*
MapUserPhysicalPagesScatter
MoveMemory
ResetWriteWatch
VirtualAlloc
VirtualAllocEx
VirtualFree
VirtualFreeEx
VirtualLock
VirtualProtect
VirtualProtectEx
VirtualQuery
VirtualQueryEx
VirtualUnlock
ZeroMemory
Memory Management Structures 328
MEMORY_BASIC_INFORMATION
MEMORYSTATUS
PROCESS_HEAP_ENTRY
Process and Thread Reference 74
Process and Thread Functions 74
AssignProcessToJobObject
AttachThreadInput
BindIoCompletionCallback
CommandLineToArgvW
ConvertThreadToFiber
CreateFiber
CreateJobObject
CreateProcess
CreateProcessAsUser
CreateProcessWithLogonW
CreateRemoteThread
CreateThread
DeleteFiber
ExitProcess
ExitThread
FiberProc
FreeEnvironmentStrings
Process and Thread Reference *(continued)*
Process and Thread Functions *(continued)*
GetCommandLine
GetCurrentProcess
GetCurrentProcessId
GetCurrentThread
GetCurrentThreadId
GetEnvironmentStrings
GetEnvironmentVariable
GetExitCodeProcess
GetExitCodeThread
GetGuiResources
GetPriorityClass
GetProcessAffinityMask
GetProcessIoCounters
GetProcessPriorityBoost
GetProcessShutdownParameters
GetProcessTimes
GetProcessVersion
GetProcessWorkingSetSize
GetStartupInfo
GetThreadPriority
GetThreadPriorityBoost
GetThreadTimes
OpenJobObject
OpenProcess
OpenThread
QueryInformationJobObject
QueueUserWorkItem
ResumeThread
SetEnvironmentVariable
SetInformationJobObject
SetPriorityClass
SetProcessAffinityMask
SetProcessPriorityBoost
SetProcessShutdownParameters
SetProcessWorkingSetSize
SetThreadAffinityMask

Process and Thread Reference *(continued)*
Process and Thread Functions *(continued)*
SetThreadIdealProcessor
SetThreadPriority
SetThreadPriorityBoost
Sleep
SleepEx
SuspendThread
SwitchToFiber
SwitchToThread
TerminateJobObject
TerminateProcess
TerminateThread
ThreadProc
TlsAlloc
TlsFree
TlsGetValue
TlsSetValue
UserHandleGrantAccess
WaitForInputIdle
Process and Thread Structures 184
IO_COUNTERS
JOBOBJECT_ASSOCIATE_COMPLETION_PORT
JOBOBJECT_BASIC_ACCOUNTING_INFORMATION
JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION
JOBOBJECT_BASIC_LIMIT_INFORMATION
JOBOBJECT_BASIC_PROCESS_ID_LIST
JOBOBJECT_BASIC_UI_RESTRICTIONS
Process and Thread Structures *(continued)*
JOBOBJECT_END_OF_JOB_TIME_INFORMATION
JOBOBJECT_EXTENDED_LIMIT_INFORMATION
JOBOBJECT_SECURITY_LIMIT_INFORMATION
PROCESS_INFORMATION
STARTUPINFO
Process and Thread Macros 207
GetCurrentFiber
GetFiberData
Structured Exception Handling Reference 750
AbnormalTermination
GetExceptionCode
GetExceptionInformation
RaiseException
SetUnhandledExceptionFilter
UnhandledExceptionFilter
Structured Exception Handling Structures 759
EXCEPTION_POINTERS
EXCEPTION_RECORD
Unicode and Character Set Reference 795
Unicode and Character Set Functions 795
GetTextCharset
GetTextCharsetInfo
IsDBCSLeadByte
IsDBCSLeadByteEx
IsTextUnicode
MultiByteToWideChar
TranslateCharsetInfo
WideCharToMultiByte
Unicode and Character Set

Structures 810
CHARSETINFO
FONTSIGNATURE
LOCALESIGNATURE
Unicode and Character Set
Macros 812
TEXT

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
CancelIo 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
FiberProc ... 116
FILE_NOTIFY_INFORMATION ... 609
FileEncryptionStatus ... 662
FileIOCompletionRoutine ... 507
FillMemory ... 264
FindAtom ... 348
FindClose ... 509
FindCloseChangeNotification ... 510
FINDEX_INFO_LEVELS ... 617
FINDEX_SEARCH_OPS ... 618
FindFirstChangeNotification ... 511
FindFirstFile ... 513
FindFirstFileEx ... 514
FindFirstVolume ... 663
FindFirstVolumeMountPoint ... 665
FindNextChangeNotification ... 517
FindNextFile ... 518
FindNextVolume ... 666
FindNextVolumeMountPoint ... 667
FindVolumeClose ... 668
FindVolumeMountPointClose ... 669
FlashWindow ... 769
FlashWindowEx ... 770
FLASHWINFO ... 783
FlushFileBuffers ... 519
FONTSIGNATURE ... 810
ForegroundIdleProc ... 432
FormatMessage ... 771
FreeEncryptionCertificateHashList ... 670
FreeEnvironmentStrings ... 117
FreeLibrary ... 222
FreeLibraryAndExitThread ... 223
FreeUserPhysicalPages ... 265

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
GetDiskFreeSpace ... 523
GetDiskFreeSpaceEx ... 525
GetDriveType ... 526
GetEnvironmentStrings ... 122
GetEnvironmentVariable ... 123
GetExceptionCode ... 751
GetExceptionInformation ... 753
GetExitCodeProcess ... 124
GetExitCodeThread ... 125
GetFiberData ... 207
GetFileAttributes ... 527
GetFileAttributesEx ... 530
GetFileInformationByHandle ... 531
GetFileSize ... 532
GetFileSizeEx ... 533
GetFileType ... 534
GetFullPathName ... 535
GetGuiResources ... 126
GetHandleInformation ... 413
GetLastError ... 776
GetLogicalDrives ... 536
GetLogicalDriveStrings ... 537
GetLongPathName ... 538
GetModuleFileName ... 224
GetModuleHandle ... 225
GetMsgProc ... 433
GetOpenClipboardWindow ... 371
GetPriorityClass ... 127
GetPriorityClipboardFormat ... 372
GetProcAddress ... 226
GetProcessAffinityMask ... 128
GetProcessHeap ... 266
GetProcessHeaps ... 267
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

GlobalFindAtom.. 353
GlobalGetAtomName 354
GlobalMemoryStatus.................................. 269

H

HeapAlloc 271
HeapCompact 273
HeapCreate 275
HeapDestroy.. 277
HeapFree... 278
HeapLock 280
HeapReAlloc 281
HeapSize 284
HeapUnlock 286
HeapValidate 287
HeapWalk 289

I

IDiskQuotaControl 683
AddUserName 684
AddUserSid 686
CreateEnumUsers 688
CreateUserBatch 690
DeleteUser...................................... 691
FindUserName 692
FindUserSid.................................... 693
GetDefaultQuotaLimit...................... 694
GetDefaultQuotaLimitText................ 695
GetDefaultQuotaThreshold 696
GetDefaultQuotaThresholdText 697
GetQuotaLogFlags 698
GetQuotaState................................. 699
GiveUserNameResolutionPriority 700
Initialize... 701
InvalidateSidNameCache................ 702
SetDefaultQuotaLimit 703
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
GetQuotaUsedText..........................718
GetSid 719
GetSidLength 720
Invalidate...721
SetQuotaLimit 721
SetQuotaThreshold..........................722
IDiskQuotaUserBatch 723
Add..724
Remove...725
RemoveAll.......................................726
FlushToDisk 726
IEnumDiskQuotaUsers 727
Clone...728
Next...729
Reset...730
Skip 730
InitAtomTable...355
Int32x32To64 546
Int64ShllMod32 547
Int64ShraMod32 548
Int64ShrlMod32...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

JOBOBJECT_ASSOCIATE_COMPLETION_PORT 85
JOBOBJECT_BASIC_ACCOUNTING_INFORMATION...188
JOBOBJECT_BASIC_AND_IO_ACCOUNTING_INFORMATION...190
JOBOBJECT_BASIC_LIMIT_INFORMATION...191
JOBOBJECT_BASIC_PROCESS_ID_LIST 195
JOBOBJECT_BASIC_UI_RESTRICTIONS 196
JOBOBJECT_END_OF_JOB_TIME_INFORMATION...197
JOBOBJECT_EXTENDED_LIMIT_INFORMATION...199
JOBOBJECT_SECURITY_LIMIT_INFORMATION...200

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_CHANGECBCHAIN 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_GETHORIZONTALEXTENT 96
CB_GETITEMDATA 97
CB_GETITEMHEIGHT 98
CB_GETLBTEXT 99
CB_GETLBTEXTLEN 100
CB_GETLOCALE 101
CB_GETTOPINDEX 102
CB_INITSTORAGE 103
CB_INSERTSTRING 104
CB_LIMITTEXT 105
CB_RESETCONTENT 106
CB_SELECTSTRING 106
CB_SETCURSEL 108
CB_SETDROPPEDWIDTH 108
CB_SETEDITSEL 109
CB_SETEXTENDEDUI 110
CB_SETHORIZONTALEXTENT 111
CB_SETITEMDATA 112
CB_SETITEMHEIGHT 113
CB_SETLOCALE 114
CB_SETTOPINDEX 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
CreateIcon 219
CreateIconFromResource 221
CreateIconFromResourceEx 222
CreateIconIndirect 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
GetMenuDefaultItem 261
GetMenuInfo 262
GetMenuItemCount 263
GetMenuItemID 264
GetMenuItemInfo 264
GetMenuItemRect 266
GetMenuState 267
GetMenuString 269
GetMessage 618
GetMessageExtraInfo 620
GetMessagePos 621
GetMessageTime 622
GetMouseMovePointsEx 374
GetNextDlgGroupItem 562
GetNextDlgTabItem 563
GetProp 689
GetQueueStatus 622
GetScrollBarInfo 136
GetScrollInfo 137
GetScrollPos 139
GetScrollRange 140
GetStringTypeA 338
GetStringTypeEx 342
GetStringTypeW 346
GetSubMenu 270
GetSystemMenu 271

H

HARDWAREINPUT 509
HideCaret 195
HiliteMenuItem 272

I

ICONINFO 239
ICONMETRICS 240
INPUT 510
InSendMessage 624
InSendMessageEx 625
InsertMenu 273
InsertMenuItem 276
IsCharAlpha 350
IsCharAlphaNumeric 351
IsCharLower 352
IsCharUpper 352
IsDialogMessage 564
IsDlgButtonChecked 55
IsMenu 278
IsWindowEnabled 483

K

keybd_event 483
KEYBDINPUT 511
KillTimer 674

L

LASTINPUTINFO 513
LoadAccelerators 449
LoadCursor 208
LoadCursorFromFile 209
LoadIcon 235
LoadKeyboardLayout 485
LoadMenu 278
LoadMenuIndirect 279
LoadString 353
LookupIconIdFromDirectory 236
LookupIconIdFromDirectoryEx 238
lstrcat 354
lstrcmp 355
lstrcmpi 356
lstrcpy 358
lstrcpyn 359
lstrlen 360

M

MapDialogRect 566
MapVirtualKey 487
MapVirtualKeyEx 489
MDICREATESTRUCT 659
MDINEXTMENU 297
MEASUREITEMSTRUCT 82
MENUBARINFO 297
MENUEX_TEMPLATE_HEADER 298
MENUEX_TEMPLATE_ITEM 299
MENUGETOBJECTINFO 301
MENUINFO 302
MenuItemFromPoint 280
MENUITEMINFO 304
MENUITEMTEMPLATE 309
MENUITEMTEMPLATEHEADER 310
MessageBox 567
MessageBoxEx 572
MessageBoxIndirect 577
ModifyMenu 281
mouse_event 376
MOUSEINPUT 514
MOUSEMOVEPOINT 385
MSG 645
MSGBOXPARAMS 593

O

OemKeyScan 491
OemToChar 361
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
WindowProc.......685
WM_ACTIVATE.......517
WM_APP.......646
WM_APPCOMMAND.......387
WM_CAPTURECHANGED.......390
WM_CHANGEUISTATE.......453
WM_CHAR.......518
WM_COMMAND.......311
WM_COMPAREITEM.......126
WM_CONTEXTMENU.......312
WM_CTLCOLORBTN.......68
WM_CTLCOLORDLG.......597
WM_CTLCOLORSCROLLBAR.......165
WM_CTLCOLORSTATIC.......180
WM_DEADCHAR.......520
WM_DRAWITEM.......127
WM_ENTERIDLE.......599
WM_ENTERMENULOOP.......314
WM_ERASEBKGND.......241
WM_EXITMENULOOP.......315
WM_GETDLGCODE.......600
WM_GETFONT.......50
WM_GETHOTKEY.......522
WM_HOTKEY.......523
WM_HSCROLL.......166
WM_ICONERASEBKGND.......242
WM_INITDIALOG.......601
WM_INITMENU.......455
WM_INITMENUPOPUP.......456
WM_KEYDOWN.......524
WM_KEYUP.......526
WM_KILLFOCUS.......527
WM_LBUTTONDBLCLK.......391
WM_LBUTTONDOWN.......392
WM_LBUTTONUP.......394
WM_MBUTTONDBLCLK.......395
WM_MBUTTONDOWN.......397
WM_MBUTTONUP.......398
WM_MDIACTIVATE.......661
WM_MDICASCADE.......662
WM_MDICREATE.......663
WM_MDIDESTROY.......665
WM_MDIGETACTIVE.......666
WM_MDIICONARRANGE.......667
WM_MDIMAXIMIZE.......667
WM_MDINEXT.......668
WM_MDIREFRESHMENU.......669
WM_MDIRESTORE.......670
WM_MDISETMENU.......671
WM_MDITILE.......672
WM_MEASUREITEM.......128
WM_MENUCHAR.......456
WM_MENUCOMMAND.......316
WM_MENUDRAG.......316
WM_MENUGETOBJECT.......317
WM_MENURBUTTONUP.......318
WM_MENUSELECT.......458
WM_MOUSEACTIVATE.......399
WM_MOUSEHOVER.......401
WM_MOUSELEAVE.......402
WM_MOUSEMOVE.......403
WM_MOUSEWHEEL.......404
WM_NCHITTEST.......407
WM_NCLBUTTONDBLCLK.......409
WM_NCLBUTTONDOWN.......410
WM_NCLBUTTONUP.......411
WM_NCMBUTTONDBLCLK.......412
WM_NCMBUTTONDOWN.......414
WM_NCMBUTTONUP.......415
WM_NCMOUSEHOVER.......416
WM_NCMOUSELEAVE.......417
WM_NCMOUSEMOVE.......418
WM_NCRBUTTONDBLCLK.......419
WM_NCRBUTTONDOWN.......420
WM_NCRBUTTONUP.......421
WM_NCXBUTTONDBLCLK.......423
WM_NCXBUTTONDOWN.......424
WM_NCXBUTTONUP.......426
WM_NEXTDLGCTL.......602
WM_NEXTMENU.......319
WM_PAINTICON.......243
WM_QUERYUISTATE.......459
WM_RBUTTONDBLCLK.......427
WM_RBUTTONDOWN.......429
WM_RBUTTONUP.......430
WM_SETCURSOR.......217
WM_SETFOCUS.......528
WM_SETFONT.......51
WM_SETHOTKEY.......529
WM_SYSCHAR.......460
WM_SYSCOMMAND.......462
WM_SYSDEADCHAR.......530
WM_SYSKEYDOWN.......532
WM_SYSKEYUP.......534
WM_TIMER.......679
WM_UNINITMENUPOPUP.......320
WM_UPDATEUISTATE.......464
WM_USER.......647
WM_VSCROLL.......168
WM_XBUTTONDBLCLK.......431
WM_XBUTTONDOWN.......433
WM_XBUTTONUP.......435
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
BitBlt 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
EMRCREATEPALETTE 436
EMRCREATEPEN 437
EMRELLIPSE
EMRRECTANGLE 437
EMREOF 438
EMREXCLUDECLIPRECT 439
EMRINTERSECTCLIPRECT 439
EMREXTCREATEFONTINDIRECTW 439
EMREXTCREATEPEN 440
EMREXTFLOODFILL 441
EMREXTSELECTCLIPRGN 442
EMREXTTEXTOUTA 443
EMREXTTEXTOUTW 443
EMRFILLPATH
EMRSTROKEANDFILLPATH 444
EMRSTROKEPATH 444
EMRFILLRGN 444
EMRFORMAT 445
EMRFRAMERGN 446
EMRGDICOMMENT 447
EMRGLSBOUNDEDRECORD 448
EMRGLSRECORD 449
EMRGRADIENTFILL 450
EMRINVERTRGN 451
EMRPAINTRGN 451
EMRLINETO 452
EMRMOVETOEX 452
EMRMASKBLT 452
EMRMODIFYWORLDTRANSFORM 455
EMROFFSETCLIPRGN 455
EMRPIXELFORMAT 456
EMRPLGBLT 457
EMRPOLYDRAW 459
EMRPOLYDRAW16 460
EMRPOLYLINE 461
EMRPOLYBEZIER 461
EMRPOLYGON 461
EMRPOLYBEZIERTO 461
EMRPOLYLINETO 461
EMRPOLYLINE16 462
EMRPOLYBEZIER16 462
EMRPOLYGON16 462
EMRPOLYBEZIERTO16 462
EMRPOLYLINETO16 462
EMRPOLYPOLYLINE 463
EMRPOLYPOLYGON 463
EMRPOLYPOLYLINE16 464
EMRPOLYPOLYGON16 464
EMRPOLYTEXTOUTA 464
EMRPOLYTEXTOUTW 464
EMRRESIZEPALETTE 466
EMRRESTOREDC 466
EMRROUNDRECT 467
EMRSCALEVIEWPORTEXTEX 468
EMRSCALEWINDOWEXTEX 468
EMRSELECTOBJECT 469
EMRDELETEOBJECT 469
EMRSELECTPALETTE 470
EMRSETARCDIRECTION 471
EMRSETBKCOLOR 471
EMRSETTEXTCOLOR 471
EMRSETCOLORADJUSTMENT 472
EMRSETCOLORSPACE 469
EMRSELECTCOLORSPACE 469
EMRDELETECOLORSPACE 469
EMRSETDIBITSTODEVICE 472
EMRSETICMPROFILE 474
EMRSETMAPPERFLAGS 475
EMRSETMITERLIMIT 476
EMRSETPALETTEENTRIES 476
EMRSETPIXELV 477
EMRSETVIEWPORTEXTEX 478
EMRSETWINDOWEXTEX 478
EMRSETVIEWPORTORGEX 479
EMRSETWINDOWORGEX 479
EMRSETBRUSHORGEX 479
EMRSETWORLDTRANSFORM 479
EMRSTRETCHBLT 480
EMRSTRETCHDIBITS 482
EMRTEXT 484
EMRTRANSPARENTBLT 485
EndPaint 526
EndPath 590
Enhanced Metafile Records with No Parameters 487
Enhanced Metafile Records with One Parameter 487
EnhMetaFileProc 402
ENHMETAHEADER 488
ENHMETARECORD 491
EnumDisplayDevices 310
EnumDisplaySettings 311
EnumDisplaySettingsEx 313
EnumEnhMetaFile 403
EnumObjects 316
EnumObjectsProc 317
EqualRect 619
ExcludeClipRect 177
ExcludeUpdateRgn 526
ExtCreatePen 608
ExtFloodFill 80
EXTLOGPEN 611
ExtSelectClipRgn 178

F

FillPath 591
FillRect 357
FlattenPath 592

FrameRect 358

G

GdiComment 404
GdiFlush 527
GdiGetBatchLimit 529
GdiSetBatchLimit 530
GetArcDirection 376
GetBitmapDimensionEx 82
GetBkColor 531
GetBkMode 531
GetBoundsRect 532
GetBrushOrgEx 164
GetBValue 226
GetClipBox 180
GetClipRgn 181
GetColorAdjustment 205
GetCurrentObject 318
GetCurrentPositionEx 255
GetDC 319
GetDCBrushColor 320
GetDCEx 321
GetDCOrgEx 323
GetDCPenColor 324
GetDeviceCaps 325
GetDIBColorTable 83
GetDIBits 84
GetEnhMetaFile 407
GetEnhMetaFileBits 408
GetEnhMetaFileHeader 411
GetEnhMetaFilePaletteEntries 412
GetGraphicsMode 256
GetGValue 226
GetMapMode 257
GetMetaRgn 182
GetMiterLimit 593
GetNearestColor 206
GetNearestPaletteIndex 207
GetObject 331
GetObjectType 333
GetPaletteEntries 208
GetPath 594
GetPixel 87
GetRandomRgn 183
GetROP2 533
GetRValue 227
GetStockObject 334
GetStretchBltMode 88
GetSysColorBrush 165
GetSystemPaletteEntries 209
GetSystemPaletteUse 210
GetUpdateRect 535
GetUpdateRgn 536
GetViewportExtEx 258
GetViewportOrgEx 259
GetWindowDC 537
GetWindowExtEx 260
GetWindowOrgEx 261
GetWindowRgn 539
GetWinMetaFileBits 413
GetWorldTransform 262
GRADIENT_RECT 146
GRADIENT_TRIANGLE 147
GradientFill 88
GrayString 540

H

HANDLETABLE 491
HTUI_ColorAdjustment 211

I

InflateRect 620
IntersectClipRect 184
IntersectRect 621
InvalidateRect 542
InvalidateRgn 543
InvertRect 359
IsRectEmpty 622

L

LineDDA 377
LineDDAProc 378
LineTo 379
LoadBitmap 90
LockWindowUpdate 544
LOGBRUSH 169
LOGBRUSH32 172
LOGPALETTE 224
LOGPEN 615
LPtoDP 263

M

MAKEPOINTS 631
MAKEROP4 152
MapWindowPoints 264
MaskBlt 92
ModifyWorldTransform 265
MoveToEx 381

O

OffsetClipRgn 185
OffsetRect 623

OffsetViewportOrgEx 267
OffsetWindowOrgEx 268
OutputProc 546

P

PaintDesktop 547
PAINTSTRUCT 561
PALETTEENTRY 224
PALETTEINDEX 228
PALETTERGB 229
PatBlt 166
PathToRegion 596
Pie 360
PlayEnhMetaFile 415
PlayEnhMetaFileRecord 417
PlgBlt 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 133
ACM_PLAY 134
ACM_STOP 135
ACN_START 142
ACN_STOP 142
AddPropSheetPageProc 441
Animate_Close 136
Animate_Create 136
Animate_Open 137
Animate_OpenEx 138
Animate_Play 139
Animate_Seek 140
Animate_Stop 141

C

CBEM_DELETEITEM 151
CBEM_GETCOMBOCONTROL 152
CBEM_GETEDITCONTROL 152
CBEM_GETEXTENDEDSTYLE 153
CBEM_GETIMAGELIST 153
CBEM_GETITEM 154
CBEM_GETUNICODEFORMAT 155
CBEM_HASEDITCHANGED 155
CBEM_INSERTITEM 156
CBEM_SETEXTENDEDSTYLE 157
CBEM_SETIMAGELIST 157
CBEM_SETITEM 158
CBEM_SETUNICODEFORMAT 159
CBEN_BEGINEDIT 160
CBEN_DELETEITEM 160
CBEN_DRAGBEGIN 161
CBEN_ENDEDIT 161
CBEN_GETDISPINFO 162
CBEN_INSERTITEM 163
CCM_GETUNICODEFORMAT 86
CCM_GETVERSION 87
CCM_SETUNICODEFORMAT 88
CCM_SETVERSION 89
COLORSCHEME 104
COMBOBOXEXITEM 164
CreatePropertySheetPage 441
CreateStatusWindow 568
CreateUpDownControl 741

D

DateTime_GetMonthCal 211
DateTime_GetMonthCalColor 211
DateTime_GetMonthCalFont 213
DateTime_GetRange 213
DateTime_GetSystemtime 214
DateTime_SetFormat 215
DateTime_SetMonthCalColor 216
DateTime_SetMonthCalFont 217
DateTime_SetRange 217
DateTime_SetSystemtime 218
DestroyPropertySheetPage 442
DL_BEGINDRAG 234
DL_CANCELDRAG 235
DL_DRAGGING 236
DL_DROPPED 236
DRAGLISTINFO 237
DrawInsert 232
DrawStatusText 569
DTM_GETMCCOLOR 203
DTM_GETMCFONT 204
DTM_GETMONTHCAL 204
DTM_GETRANGE 205
DTM_GETSYSTEMTIME 206
DTM_SETFORMAT 206
DTM_SETMCCOLOR 207
DTM_SETMCFONT 208
DTM_SETRANGE 209
DTM_SETSYSTEMTIME 210
DTN_CLOSEUP 219
DTN_DATETIMECHANGE 220
DTN_DROPDOWN 221
DTN_FORMAT 222
DTN_FORMATQUERY 222
DTN_USERSTRING 223
DTN_WMKEYDOWN 224

E

ExtensionPropSheetPageProc 443

F

FIRST_IPADDRESS 331
FlatSB_EnableScrollBar 242
FlatSB_GetScrollInfo 243
FlatSB_GetScrollPos 244
FlatSB_GetScrollProp 245
FlatSB_GetScrollRange 247
FlatSB_SetScrollInfo 248
FlatSB_SetScrollPos 249
FlatSB_SetScrollProp 250
FlatSB_SetScrollRange 253
FlatSB_ShowScrollBar 254
FORWARD_WM_NOTIFY 92
FOURTH_IPADDRESS 332

G

GetEffectiveClientRect 81
GetMUILanguage 82

H

HANDLE_WM_NOTIFY 93
HDHITTESTINFO 307
HDITEM 309
HDLAYOUT 312
HDM_CLEARFILTER 264
HDM_CREATEDRAGIMAGE 265
HDM_DELETEITEM 265
HDM_EDITFILTER 266
HDM_GETBITMAPMARGIN 267
HDM_GETIMAGELIST 267
HDM_GETITEM 268
HDM_GETITEMCOUNT 268
HDM_GETITEMRECT 269
HDM_GETORDERARRAY 270
HDM_GETUNICODEFORMAT 271
HDM_HITTEST 271
HDM_INSERTITEM 272
HDM_LAYOUT 272
HDM_ORDERTOINDEX 273
HDM_SETBITMAPMARGIN 274
HDM_SETFILTERCHANGETIMEOUT 274
HDM_SETHOTDIVIDER 275
HDM_SETIMAGELIST 276
HDM_SETITEM 277
HDM_SETORDERARRAY 277
HDM_SETUNICODEFORMAT 278, 279
HDN_BEGINDRAG 297
HDN_BEGINTRACK 298
HDN_DIVIDERDBLCLICK 298
HDN_ENDDRAG 299
HDN_ENDTRACK 299
HDN_FILTERBTNCLICK 300
HDN_FILTERCHANGE 301
HDN_GETDISPINFO 301
HDN_ITEMCHANGED 302
HDN_ITEMCHANGING 303
HDN_ITEMCLICK 303
HDN_ITEMDBLCLICK 304
HDN_TRACK 304
HDTEXTFILTER Structure 312
Header_ClearFilter 280
Header_CreateDragImage 281
Header_DeleteItem 281
Header_EditFilter 282
Header_GetBitmapMargin 283
Header_GetImageList 284
Header_GetItem 284
Header_GetItemCount 285
Header_GetItemRect 286
Header_GetOrderArray 287
Header_GetUnicodeFormat 288
Header_InsertItem 288
Header_Layout 289
Header_OrderToIndex 290
Header_SetBitmapMargin 291
Header_SetFilterChangeTimeout 292
Header_SetHotDivider 292
Header_SetImageList 293
Header_SetItem 294
Header_SetOrderArray 295
Header_SetUnicodeFormat 296
HKM_GETHOTKEY 321
HKM_SETHOTKEY 322
HKM_SETRULES 323

I

INDEXTOSTATEIMAGEMASK 94
InitCommonControls 83
InitCommonControlsEx 83
INITCOMMONCONTROLSEX 104
InitializeFlatSB 241
InitMUILanguage 84
IPM_CLEARADDRESS 326
IPM_GETADDRESS 327
IPM_ISBLANK 328
IPM_SETADDRESS 328
IPM_SETFOCUS 329
IPM_SETRANGE 329
IPN_FIELDCHANGED 330

L

LBItemFromPt 233

M

MakeDragList 234
MAKEIPADDRESS 332
MAKEIPRANGE 333
MCHITTESTINFO 388
MCM_GETCOLOR 345
MCM_GETCURSEL 346
MCM_GETFIRSTDAYOFWEEK 347
MCM_GETMAXSELCOUNT 348
MCM_GETMAXTODAYWIDTH 348
MCM_GETMINREQRECT 349
MCM_GETMONTHDELTA 350
MCM_GETMONTHRANGE 351
MCM_GETRANGE 352
MCM_GETSELRANGE 353
MCM_GETTODAY 353
MCM_GETUNICODEFORMAT 354
MCM_HITTEST 355
MCM_SETCOLOR 357
MCM_SETCURSEL 358
MCM_SETDAYSTATE 359
MCM_SETFIRSTDAYOFWEEK 360
MCM_SETMAXSELCOUNT 360
MCM_SETMONTHDELTA 361
MCM_SETRANGE 362
MCM_SETSELRANGE 363
MCM_SETTODAY 364
MCM_SETUNICODEFORMAT 364
MCN_GETDAYSTATE 385
MCN_SELCHANGE 386
MCN_SELECT 386
MenuHelp 570
MonthCal_GetColor 365
MonthCal_GetCurSel 366
MonthCal_GetFirstDayOfWeek 367
MonthCal_GetMaxSelCount 368
MonthCal_GetMaxTodayWidth 369
MonthCal_GetMinReqRect 369
MonthCal_GetMonthDelta 370
MonthCal_GetMonthRange 371
MonthCal_GetRange 372
MonthCal_GetSelRange 373
MonthCal_GetToday 374
MonthCal_GetUnicodeFormat 374
MonthCal_HitTest 375
MonthCal_SetColor 376
MonthCal_SetCurSel 377
MonthCal_SetDayState 378
MonthCal_SetFirstDayOfWeek 379
MonthCal_SetMaxSelCount 380
MonthCal_SetMonthDelta 381
MonthCal_SetRange 382
MonthCal_SetSelRange 383
MonthCal_SetToday 383
MonthCal_SetUnicodeFormat 384
MONTHDAYSTATE 391

N

NM_CHAR 95
NM_CLICK 95
NM_CLICK (status bar) 584
NM_CLICK (tab) 645
NM_CUSTOMDRAW 117
NM_CUSTOMDRAW (header) 305
NM_CUSTOMDRAW (rebar) 541
NM_CUSTOMDRAW (Tooltip) 693
NM_CUSTOMDRAW (trackbar) 734
NM_DBLCLK 96
NM_DBLCLK (status bar) 585
NM_HOVER 96
NM_KEYDOWN 97
NM_KILLFOCUS 98
NM_KILLFOCUS (date time) 225
NM_NCHITTEST 98
NM_NCHITTEST (rebar) 542
NM_OUTOFMEMORY 99
NM_RCLICK 99
NM_RCLICK (header) 306
NM_RCLICK (status bar) 585
NM_RCLICK (tab) 645
NM_RDBLCLK 100
NM_RDBLCLK (status bar) 586
NM_RELEASEDCAPTURE 101
NM_RELEASEDCAPTURE (header) 307
NM_RELEASEDCAPTURE (monthcal) 387
NM_RELEASEDCAPTURE (pager) 414
NM_RELEASEDCAPTURE (rebar) 543
NM_RELEASEDCAPTURE (tab) 646
NM_RELEASEDCAPTURE (trackbar) 735
NM_RELEASEDCAPTURE (up-down) 752
NM_RETURN 101
NM_SETCURSOR 102
NM_SETCURSOR (ComboBoxEx) 163
NM_SETFOCUS 102
NM_SETFOCUS (date time) 225
NM_TOOLTIPSCREATED 103
NMCBEDRAGBEGIN 167
NMCBEENDEDIT 166
NMCHAR 105
NMCOMBOBOXEX 167
NMCUSTOMDRAW 119
NMDATETIMECHANGE 226
NMDATETIMEFORMAT 227
NMDATETIMEFORMATQUERY 228

NMDATETIMESTRING 229
NMDATETIMEWMKEYDOWN 230
NMDAYSTATE 390
NMHDDISPINFO 313
NMHDFILTERBTNCLICK Structure 314
NMHDR 106
NMHEADER 315
NMIPADDRESS 335
NMKEY 107
NMMOUSE 107
NMOBJECTNOTIFY 108
NMPGCALCSIZE 416
NMPGSCROLL 417
NMRBAUTOSIZE 550
NMREBAR 551
NMREBARCHEVRON 552
NMREBARCHILDSIZE 553
NMSELCHANGE 390
NMTCKEYDOWN 650
NMTOOLTIPSCREATED 109
NMTTCUSTOMDRAW 697
NMTTDISPINFO 697
NMUPDOWN 753

P

Pager_ForwardMouse 405
Pager_GetBkColor 405
Pager_GetBorder 406
Pager_GetButtonSize 407
Pager_GetButtonState 407
Pager_GetDropTarget 408
Pager_GetPos 409
Pager_RecalcSize 409
Pager_SetBkColor 410
Pager_SetBorder 411
Pager_SetButtonSize 412
Pager_SetChild 412
Pager_SetPos 413
PBM_DELTAPOS 423
PBM_GETPOS 423
PBM_GETRANGE 424
PBM_SETBARCOLOR 425
PBM_SETBKCOLOR 425
PBM_SETPOS 426
PBM_SETRANGE 427
PBM_SETRANGE32 427
PBM_SETSTEP 428
PBM_STEPIT 429
PBRANGE 429
PGM_FORWARDMOUSE 396
PGM_GETBKCOLOR 397
PGM_GETBORDER 397
PGM_GETBUTTONSIZE 398
PGM_GETBUTTONSTATE 398
PGM_GETDROPTARGET 399
PGM_GETPOS 400
PGM_RECALCSIZE 401
PGM_SETBKCOLOR 401
PGM_SETBORDER 402
PGM_SETBUTTONSIZE 402
PGM_SETCHILD 403
PGM_SETPOS 404
PGN_CALCSIZE 415
PGN_SCROLL 415
PropertySheet 444
PropSheet_AddPage 467
PropSheet_Apply 467
PropSheet_CancelToClose 468
PropSheet_Changed 469
PropSheet_GetCurrentPageHwnd 470
PropSheet_GetTabControl 471
PropSheet_HwndToIndex 471
PropSheet_IdToIndex 472
PropSheet_IndexToHwnd 473
PropSheet_IndexToId 473
PropSheet_IndexToPage 474
PropSheet_InsertPage 475
PropSheet_IsDialogMessage 476
PropSheet_PageToIndex 477
PropSheet_PressButton 478
PropSheet_QuerySiblings 479
PropSheet_RebootSystem 480
PropSheet_RemovePage 480
PropSheet_RestartWindows 481
PropSheet_SetCurSel 482
PropSheet_SetCurSelByID 483
PropSheet_SetFinishText 483
PropSheet_SetHeaderSubTitle 484
PropSheet_SetHeaderTitle 485
PropSheet_SetTitle 486
PropSheet_SetWizButtons 487
PropSheet_UnChanged 488
PROPSHEETHEADER 499
PROPSHEETPAGE 505
PropSheetPageProc 445
PropSheetProc 446
PSHNOTIFY 509
PSM_ADDPAGE 447
PSM_APPLY 448
PSM_CANCELTOCLOSE 448
PSM_CHANGED 449
PSM_GETCURRENTPAGEHWND 450
PSM_GETTABCONTROL 451
PSM_HWNDTOINDEX 451
PSM_IDTOINDEX 452
PSM_INDEXTOHWND 452
PSM_INDEXTOID 453
PSM_INDEXTOPAGE 453
PSM_INSERTPAGE 454

PSM_ISDIALOGMESSAGE........ 455
PSM_PAGETOINDEX........ 456
PSM_PRESSBUTTON........ 457
PSM_QUERYSIBLINGS........ 457
PSM_REBOOTSYSTEM........ 458
PSM_REMOVEPAGE........ 459
PSM_RESTARTWINDOWS........ 459
PSM_SETCURSEL........ 460
PSM_SETCURSELID........ 461
PSM_SETFINISHTEXT........ 462
PSM_SETHEADERSUBTITLE........ 462
PSM_SETHEADERTITLE........ 463
PSM_SETTITLE........ 464
PSM_SETWIZBUTTONS........ 465
PSM_UNCHANGED........ 466
PSN_APPLY........ 489
PSN_GETOBJECT........ 490
PSN_HELP........ 490
PSN_KILLACTIVE........ 491
PSN_QUERYCANCEL........ 492
PSN_QUERYINITIALFOCUS........ 493
PSN_RESET........ 494
PSN_SETACTIVE........ 495
PSN_TRANSLATEACCELERATOR........ 495
PSN_WIZBACK........ 496
PSN_WIZFINISH........ 497
PSN_WIZNEXT........ 498

R

RB_BEGINDRAG........ 516
RB_DELETEBAND........ 517
RB_DRAGMOVE........ 517
RB_ENDDRAG........ 518
RB_GETBANDBORDERS........ 518
RB_GETBANDCOUNT........ 519
RB_GETBANDINFO........ 520
RB_GETBARHEIGHT........ 521
RB_GETBARINFO........ 521
RB_GETBKCOLOR........ 522
RB_GETCOLORSCHEME........ 522
RB_GETDROPTARGET........ 523
RB_GETPALETTE........ 524
RB_GETRECT........ 524
RB_GETROWCOUNT........ 525
RB_GETROWHEIGHT........ 525
RB_GETTEXTCOLOR........ 526
RB_GETTOOLTIPS........ 526
RB_GETUNICODEFORMAT........ 527
RB_HITTEST........ 528
RB_IDTOINDEX........ 528
RB_INSERTBAND........ 529
RB_MAXIMIZEBAND........ 530
RB_MINIMIZEBAND........ 530
RB_MOVEBAND........ 531
RB_PUSHCHEVRON........ 532
RB_SETBANDINFO........ 533
RB_SETBARINFO........ 533
RB_SETBKCOLOR........ 534
RB_SETCOLORSCHEME........ 535
RB_SETPALETTE........ 535
RB_SETPARENT........ 536
RB_SETTEXTCOLOR........ 537
RB_SETTOOLTIPS........ 538
RB_SETUNICODEFORMAT........ 538
RB_SHOWBAND........ 539
RB_SIZETORECT........ 540
RBHITTESTINFO........ 554
RBN_AUTOSIZE........ 543
RBN_BEGINDRAG........ 544
RBN_CHEVRONPUSHED........ 545
RBN_CHILDSIZE........ 545
RBN_DELETEDBAND........ 546
RBN_DELETINGBAND........ 547
RBN_ENDDRAG........ 547
RBN_GETOBJECT........ 548
RBN_HEIGHTCHANGE........ 549
RBN_LAYOUTCHANGED........ 549
REBARBANDINFO........ 554
REBARINFO........ 558

S

SB_GETBORDERS........ 571
SB_GETICON........ 572
SB_GETPARTS........ 572
SB_GETRECT........ 573
SB_GETTEXT........ 573
SB_GETTEXTLENGTH........ 575
SB_GETTIPTEXT........ 576
SB_GETUNICODEFORMAT........ 576
SB_ISSIMPLE........ 577
SB_SETBKCOLOR........ 578
SB_SETICON........ 578
SB_SETMINHEIGHT........ 579
SB_SETPARTS........ 580
SB_SETTEXT........ 580
SB_SETTIPTEXT........ 581
SB_SETUNICODEFORMAT........ 582
SB_SIMPLE........ 583
SBN_SIMPLEMODECHANGE........ 586
SECOND_IPADDRESS........ 334
ShowHideMenuCtl........ 85

T

TabCtrl_AdjustRect........ 625
TabCtrl_DeleteAllItems........ 626
TabCtrl_DeleteItem........ 626

TabCtrl_DeselectAll 627
TabCtrl_GetCurFocus 628
TabCtrl_GetCurSel 628
TabCtrl_GetExtendedStyle 629
TabCtrl_GetImageList 629
TabCtrl_GetItem 630
TabCtrl_GetItemCount 631
TabCtrl_GetItemRect 631
TabCtrl_GetRowCount 632
TabCtrl_GetToolTips 633
TabCtrl_GetUnicodeFormat 633
TabCtrl_HighlightItem 634
TabCtrl_HitTest 635
TabCtrl_InsertItem 635
TabCtrl_RemoveImage 636
TabCtrl_SetCurFocus 637
TabCtrl_SetCurSel 638
TabCtrl_SetExtendedStyle 638
TabCtrl_SetImageList 639
TabCtrl_SetItem 640
TabCtrl_SetItemExtra 640
TabCtrl_SetItemSize 641
TabCtrl_SetMinTabWidth 642
TabCtrl_SetPadding 643
TabCtrl_SetToolTips 643
TabCtrl_SetUnicodeFormat 644
TBM_CLEARSEL 711
TBM_CLEARTICS 712
TBM_GETBUDDY 712
TBM_GETCHANNELRECT 713
TBM_GETLINESIZE 714
TBM_GETNUMTICS 714
TBM_GETPAGESIZE 715
TBM_GETPOS 716
TBM_GETPTICS 716
TBM_GETRANGEMAX 717
TBM_GETRANGEMIN 717, 718
TBM_GETSELEND 718
TBM_GETSELSTART 719
TBM_GETTHUMBLENGTH 719
TBM_GETTHUMBRECT 720
TBM_GETTIC 721
TBM_GETTICPOS 721
TBM_GETTOOLTIPS 722
TBM_GETUNICODEFORMAT 722
TBM_SETBUDDY 723
TBM_SETLINESIZE 724
TBM_SETPAGESIZE 725
TBM_SETPOS 725
TBM_SETRANGE 726
TBM_SETRANGEMAX 727
TBM_SETRANGEMIN 728
TBM_SETSEL 728
TBM_SETSELEND 729
TBM_SETSELSTART 730
TBM_SETTHUMBLENGTH 731
TBM_SETTIC 731
TBM_SETTIPSIDE 732
TBM_SETTOOLTIPS 733
TCHITTESTINFO 650
TCITEM 651
TCITEMHEADER 653
TCM_ADJUSTRECT 607
TCM_DELETEALLITEMS 607
TCM_DELETEITEM 608
TCM_DESELECTALL 608
TCM_GETCURFOCUS 609
TCM_GETCURSEL 610
TCM_GETEXTENDEDSTYLE 610
TCM_GETIMAGELIST 611
TCM_GETITEM 611
TCM_GETITEMCOUNT 612
TCM_GETITEMRECT 612
TCM_GETROWCOUNT 613
TCM_GETTOOLTIPS 613
TCM_GETUNICODEFORMAT 614
TCM_HIGHLIGHTITEM 615
TCM_HITTEST 615
TCM_INSERTITEM 616
TCM_REMOVEIMAGE 617
TCM_SETCURFOCUS 617
TCM_SETCURSEL 618
TCM_SETEXTENDEDSTYLE 619
TCM_SETIMAGELIST 620
TCM_SETITEM 620
TCM_SETITEMEXTRA 621
TCM_SETITEMSIZE 622
TCM_SETMINTABWIDTH 622
TCM_SETPADDING 623
TCM_SETTOOLTIPS 623
TCM_SETUNICODEFORMAT 624
TCN_FOCUSCHANGE 646
TCN_GETOBJECT 647
TCN_KEYDOWN 648
TCN_SELCHANGE 648
TCN_SELCHANGING 649
THIRD_IPADDRESS 335
TOOLINFO 699
TTHITTESTINFO 701
TTM_ACTIVATE 672
TTM_ADDTOOL 672
TTM_ADJUSTRECT 673
TTM_DELTOOL 674
TTM_ENUMTOOLS 675
TTM_GETBUBBLESIZE 675
TTM_GETCURRENTTOOL 676
TTM_GETDELAYTIME 677
TTM_GETMARGIN 677
TTM_GETMAXTIPWIDTH 678
TTM_GETTEXT 679

TTM_GETTIPBKCOLOR 680
TTM_GETTOOLCOUNT 680
TTM_GETTOOLINFO 681
TTM_HITTEST 681
TTM_NEWTOOLRECT 682
TTM_POP 683
TTM_RELAYEVENT 683
TTM_SETDELAYTIME 684
TTM_SETMARGIN 685
TTM_SETMAXTIPWIDTH 686
TTM_SETTIPBKCOLOR 687
TTM_SETTIPTEXTCOLOR 687
TTM_SETTITLE 688
TTM_SETTOOLINFO 689
TTM_TRACKACTIVATE 689
TTM_TRACKPOSITION 690
TTM_UPDATE 691
TTM_UPDATETIPTEXT 692
TTM_WINDOWFROMPOINT 692
TTN_GETDISPINFO 694
TTN_POP 695
TTN_SHOW 696

U

UDACCEL 754
UDM_GETACCEL 743
UDM_GETBASE 744
UDM_GETBUDDY 744
UDM_GETPOS 744
UDM_GETRANGE 745
UDM_GETRANGE32 746
UDM_GETUNICODEFORMAT 746
UDM_SETACCEL 747
UDM_SETBASE 748
UDM_SETBUDDY 748
UDM_SETPOS 749
UDM_SETRANGE 749
UDM_SETRANGE32 750
UDM_SETUNICODEFORMAT 751
UDN_DELTAPOS 752
UninitializeFlatSB 255

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
CPlApplet 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
Skip ... 235
IExtractIcon
Extract ... 237
GetIconLocation ... 238
IExtractImage
Extract ... 241
GetLocation ... 241
IExtractImage2
GetDateStamp ... 244
IFileViewer
PrintTo ... 245
Show ... 246
ShowInitialize ... 247
IFileViewerSite
GetPinnedWindow ... 248
SetPinnedWindow ... 249
IInputObject
HasFocusIO ... 250
TranslateAcceleratorIO ... 251
UIActivateIO ... 251
IInputObjectSite
OnFocusChangeIS ... 253
InetIsOffline ... 421
INewShortcutHook
GetExtension ... 254
GetFolder ... 255
GetName ... 256
GetReferent ... 256
SetFolder ... 257
SetReferent ... 258
INotifyReplica
YouAreAReplica ... 259
IntlStrEqN ... 576
IntlStrEqNI ... 577
IntlStrEqWorker ... 578
IObjMgr
Append ... 260
Remove ... 261
IPersistFileSystemFolder
GetFolderTargetInfo ... 265
InitializeEx ... 266
IPersistFolder
Initialize ... 262
IPersistFolder2
GetCurFolder ... 263
IProgressDialog
HasUserCancelled ... 269
SetAnimation ... 269
SetCancelMsg ... 270
SetLine ... 271
SetProgress ... 272
SetProgress64 ... 273
SetTitle ... 274
StartProgressDialog ... 274
StopProgressDialog ... 276
Timer ... 276
IQueryAssociations
GetData ... 279
GetEnum ... 280
GetKey ... 280
GetString ... 281
Init ... 282
IQueryInfo
GetInfoFlags ... 284
GetInfoTip ... 285
IReconcilableObject
GetProgressFeedbackMax
Estimate ... 286
Reconcile ... 287
IReconcileInitiator
SetAbortCallback ... 292
SetProgressFeedback ... 293
IRemoteComputer
Initialize ... 294
IResolveShellLink
ResolveShellLink ... 296
IRunnableTask
IsRunning ... 298
Kill ... 299
Resume ... 299
Run ... 300
Suspend ... 300
IShellBrowser
BrowseObject ... 302
EnableModelessSB ... 304
GetControlWindow ... 304
GetViewStateStream ... 306
InsertMenusSB ... 307
OnViewWindowActive ... 308
QueryActiveShellView ... 309
RemoveMenusSB ... 310
SendControlMsg ... 311
SetMenuSB ... 312
SetStatusTextSB ... 313
SetToolbarItems ... 314
TranslateAcceleratorSB ... 315
IShellChangeNotify
OnChange ... 316
IShellDetails
ColumnClick ... 319
GetDetailsOf ... 320
IShellExecuteHook
Execute ... 323
IShellExtInit
Initialize ... 324
IShellFolder
BindToObject ... 327
BindToStorage ... 328
CompareIDs ... 329

CreateViewObject 331
EnumObjects 332
GetAttributesOf 333
GetDisplayNameOf 335
GetUIObjectOf 337
ParseDisplayName 338
SetNameOf 342
IShellFolder2
EnumSearches 344
GetDefaultColumn 345
GetDefaultColumnState 346
GetDefaultSearchGUID 347
GetDetailsEx 347
GetDetailsOf 348
MapNameToSCID 349
IShellIcon
GetIconOf 351
IShellIconOverlay
GetOverlayIconIndex 353
GetOverlayIndex 354
IShellIconOverlayIdentifier
GetOverlayInfo 356
GetPriority 357
IsMemberOf 358
IShellLink
GetArguments 360
GetDescription 361
GetHotkey 361
GetIconLocation 362
GetIDList 363
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
DestroyViewWindow 387
EnableModeless 387
EnableModelessSV 388
GetCurrentInfo 388
GetItemObject 389
Refresh 390
SaveViewState 391
SelectItem 392
TranslateAccelerator 393
UIActivate 394
IShellView2
CreateViewWindow2 396
GetView 397
HandleRename 397
SelectAndPositionItem 398
ITaskbarList
ActivateTab 400
AddTab 400
DeleteTab 401
HrInit 402
SetActiveAlt 402
IUniformResourceLocator
GetURL 403
InvokeCommand 405
SetURL 406
IURL_SETURL_FLAGS 566
IURL_SETURL_INVOKECOMMAND_FLAGS 567
IURLSearchHook
Translate 407

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
PathGetArgs 623
PathGetCharType 623
PathGetDriveNumber 624
PathIsContentType 625
PathIsDirectory 625
PathIsDirectoryEmpty 626
PathIsFileSpec 627
PathIsHTMLFile 627
PathIsLFNFileSpec 628
PathIsNetworkPath 629
PathIsPrefix 630
PathIsRelative 630
PathIsRoot 631
PathIsSameRoot 632
PathIsSystemFolder 632
PathIsUNC 633
PathIsUNCServer 634
PathIsUNCServerShare 634
PathIsURL 635
PathMakePretty 636
PathMakeSystemFolder 636
PathMatchSpec 637
PathParseIconLocation 638
PathQuoteSpaces 639
PathRelativePathTo 639
PathRemoveArgs 641
PathRemoveBackslash 641
PathRemoveBlanks 642
PathRemoveExtension 642
PathRemoveFileSpec 643
PathRenameExtension 644
PathSearchAndQualify 644
PathSetDlgItemPath 645
PathSkipRoot 646
PathStripPath 647
PathStripToRoot 647
PathUndecorate 648
PathUnExpandEnvStrings 649
PathUnmakeSystemFolder 650
PathUnquoteSpaces 651

R

RegisterDialogClasses 423
REGSAM 669

S

ScreenSaverConfigureDialog 424
ScreenSaverProc 425
SetMenuContextHelpId 426
SetWindowContextHelpId 427
SHAddToRecentDocs 428
SHAppBarMessage 429
SHAutoComplete 712
SHBindToParent 430
SHBrowseForFolder 431
SHChangeNotify 432
SHCONTF 568
SHCopyKey 675
SHCreateDirectoryEx 437
SHCreateProcessAsUser 438
SHCreateShellPalette 709
SHCreateStreamOnFile 714
SHCreateThread 714
SHDeleteEmptyKey 676
SHDeleteKey 677
SHDeleteValue 678
Shell_NotifyIcon 439
ShellAbout 441
ShellExecute 442
ShellExecuteEx 445
SHEmptyRecycleBin 447
SHEnumKeyEx 679
SHEnumValue 680
SHFileOperation 448
SHFreeNameMappings 449
SHGetDataFromIDList 450
SHGetDesktopFolder 451
SHGetDiskFreeSpace 452
SHGetFileInfo 453
SHGetFolderLocation 457
SHGetFolderPath 458
SHGetIconOverlayIndex 461
SHGetInstanceExplorer 462
SHGetMalloc 463
SHGetNewLinkInfo 464
SHGetPathFromIDList 466
SHGetSettings 466
SHGetSpecialFolderLocation 468
SHGetSpecialFolderPath 469
SHGetThreadRef 716
SHGetValue 681
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

SHRegEnumUSKey 690
SHRegEnumUSValue 691
SHRegGetBoolUSValue 692
SHRegGetPath 693
SHRegGetUSValue 694
SHRegOpenUSKey 696
SHRegQueryInfoUSKey 697
SHRegQueryUSValue 698
SHRegSetPath 700
SHRegSetUSValue 701
SHRegWriteUSValue 702
SHSetThreadRef 719
SHSetValue 704
SHStrDup 580
SOANGLETENTHS 573
SoftwareUpdateMessageBox 473
SOPALETTEINDEX 573
SOPALETTERGB 573
SORGB 574
SOSETRATIO 574
StrCat 581
StrCatBuff 581
StrChr 582
StrChrI 583
StrCmp 584
StrCmpI 585
StrCmpN 585
StrCmpNI 586
StrCpy 587
StrCpyN 588
StrCSpn 589
StrCSpnI 590
StrDup 591
StrFormatByteSize 592
StrFormatByteSize64A 593
StrFormatKBSize 594
StrFromTimeInterval 595
StrIsIntlEqual 596
StrNCat 597
StrPBrk 598
StrRChr 598
StrRChrI 599
StrRetToBuf 600
StrRetToStr 601
StrRStrI 602
StrSpn 603
StrStr 604
StrStrI 604
StrToInt 605
StrToIntEx 606
StrTrim 607

T

TranslateURL 475
TRANSLATEURL_IN_FLAGS 570

U

UndeleteFile 484
UrlApplyScheme 651
URLAssociationDialog 476
URLASSOCIATIONDIALOG_IN_FLAGS 571
UrlCanonicalize 653
UrlCombine 654
UrlCompare 655
UrlCreateFromPath 656
UrlEscape 657
UrlEscapeSpaces 658
UrlGetLocation 659
UrlGetPart 660
UrlHash 661
UrlIs 662
UrlIsFileUrl 663
UrlIsNoHistory 664
UrlIsOpaque 665
UrlUnEscape 666
UrlUnEscapeInPlace 667

W

WinHelp 477
WM_CPL_LAUNCH 747
WM_CPL_LAUNCHED 747
WM_DROPFILES 748
WM_HELP 749
WM_TCARD 749
wnsprintf 608
wvnsprintf 609

MICROSOFT LICENSE AGREEMENT

Book Companion CD

IMPORTANT—READ CAREFULLY: This Microsoft End-User License Agreement ("EULA") is a legal agreement between you (either an individual or an entity) and Microsoft Corporation for the Microsoft product identified above, which includes computer software and may include associated media, printed materials, and "online" or electronic documentation ("SOFTWARE PRODUCT"). Any component included within the SOFTWARE PRODUCT that is accompanied by a separate End-User License Agreement shall be governed by such agreement and not the terms set forth below. By installing, copying, or otherwise using the SOFTWARE PRODUCT, you agree to be bound by the terms of this EULA. If you do not agree to the terms of this EULA, you are not authorized to install, copy, or otherwise use the SOFTWARE PRODUCT; you may, however, return the SOFTWARE PRODUCT, along with all printed materials and other items that form a part of the Microsoft product that includes the SOFTWARE PRODUCT, to the place you obtained them for a full refund.

SOFTWARE PRODUCT LICENSE

The SOFTWARE PRODUCT is protected by United States copyright laws and international copyright treaties, as well as other intellectual property laws and treaties. The SOFTWARE PRODUCT is licensed, not sold.

1. **GRANT OF LICENSE.** This EULA grants you the following rights:

   a. **Software Product.** You may install and use one copy of the SOFTWARE PRODUCT on a single computer. The primary user of the computer on which the SOFTWARE PRODUCT is installed may make a second copy for his or her exclusive use on a portable computer.

   b. **Storage/Network Use.** You may also store or install a copy of the SOFTWARE PRODUCT on a storage device, such as a network server, used only to install or run the SOFTWARE PRODUCT on your other computers over an internal network; however, you must acquire and dedicate a license for each separate computer on which the SOFTWARE PRODUCT is installed or run from the storage device. A license for the SOFTWARE PRODUCT may not be shared or used concurrently on different computers.

   c. **License Pak.** If you have acquired this EULA in a Microsoft License Pak, you may make the number of additional copies of the computer software portion of the SOFTWARE PRODUCT authorized on the printed copy of this EULA, and you may use each copy in the manner specified above. You are also entitled to make a corresponding number of secondary copies for portable computer use as specified above.

   d. **Sample Code.** Solely with respect to portions, if any, of the SOFTWARE PRODUCT that are identified within the SOFTWARE PRODUCT as sample code (the "SAMPLE CODE"):

      i. **Use and Modification.** Microsoft grants you the right to use and modify the source code version of the SAMPLE CODE, *provided* you comply with subsection (d)(iii) below. You may not distribute the SAMPLE CODE, or any modified version of the SAMPLE CODE, in source code form.

      ii. **Redistributable Files.** Provided you comply with subsection (d)(iii) below, Microsoft grants you a nonexclusive, royalty-free right to reproduce and distribute the object code version of the SAMPLE CODE and of any modified SAMPLE CODE, other than SAMPLE CODE, or any modified version thereof, designated as not redistributable in the Readme file that forms a part of the SOFTWARE PRODUCT (the "Non-Redistributable Sample Code"). All SAMPLE CODE other than the Non-Redistributable Sample Code is collectively referred to as the "REDISTRIBUTABLES."

      iii. **Redistribution Requirements.** If you redistribute the REDISTRIBUTABLES, you agree to: (i) distribute the REDISTRIBUTABLES in object code form only in conjunction with and as a part of your software application product; (ii) not use Microsoft's name, logo, or trademarks to market your software application product; (iii) include a valid copyright notice on your software application product; (iv) indemnify, hold harmless, and defend Microsoft from and against any claims or lawsuits, including attorney's fees, that arise or result from the use or distribution of your software application product; and (v) not permit further distribution of the REDISTRIBUTABLES by your end user. Contact Microsoft for the applicable royalties due and other licensing terms for all other uses and/or distribution of the REDISTRIBUTABLES.

2. **DESCRIPTION OF OTHER RIGHTS AND LIMITATIONS.**

   - **Limitations on Reverse Engineering, Decompilation, and Disassembly.** You may not reverse engineer, decompile, or disassemble the SOFTWARE PRODUCT, except and only to the extent that such activity is expressly permitted by applicable law notwithstanding this limitation.
   - **Separation of Components.** The SOFTWARE PRODUCT is licensed as a single product. Its component parts may not be separated for use on more than one computer.
   - **Rental.** You may not rent, lease, or lend the SOFTWARE PRODUCT.

- **Support Services.** Microsoft may, but is not obligated to, provide you with support services related to the SOFTWARE PRODUCT ("Support Services"). Use of Support Services is governed by the Microsoft policies and programs described in the user manual, in "online" documentation, and/or in other Microsoft-provided materials. Any supplemental software code provided to you as part of the Support Services shall be considered part of the SOFTWARE PRODUCT and subject to the terms and conditions of this EULA. With respect to technical information you provide to Microsoft as part of the Support Services, Microsoft may use such information for its business purposes, including for product support and development. Microsoft will not utilize such technical information in a form that personally identifies you.
- **Software Transfer.** You may permanently transfer all of your rights under this EULA, provided you retain no copies, you transfer all of the SOFTWARE PRODUCT (including all component parts, the media and printed materials, any upgrades, this EULA, and, if applicable, the Certificate of Authenticity), **and** the recipient agrees to the terms of this EULA.
- **Termination.** Without prejudice to any other rights, Microsoft may terminate this EULA if you fail to comply with the terms and conditions of this EULA. In such event, you must destroy all copies of the SOFTWARE PRODUCT and all of its component parts.

3. **COPYRIGHT.** All title and copyrights in and to the SOFTWARE PRODUCT (including but not limited to any images, photographs, animations, video, audio, music, text, SAMPLE CODE, REDISTRIBUTABLES, and "applets" incorporated into the SOFTWARE PRODUCT) and any copies of the SOFTWARE PRODUCT are owned by Microsoft or its suppliers. The SOFTWARE PRODUCT is protected by copyright laws and international treaty provisions. Therefore, you must treat the SOFTWARE PRODUCT like any other copyrighted material **except** that you may install the SOFTWARE PRODUCT on a single computer provided you keep the original solely for backup or archival purposes. You may not copy the printed materials accompanying the SOFTWARE PRODUCT.

4. **U.S. GOVERNMENT RESTRICTED RIGHTS.** The SOFTWARE PRODUCT and documentation are provided with RESTRICTED RIGHTS. Use, duplication, or disclosure by the Government is subject to restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c)(1) and (2) of the Commercial Computer Software—Restricted Rights at 48 CFR 52.227-19, as applicable. Manufacturer is Microsoft Corporation/One Microsoft Way/Redmond, WA 98052-6399.

5. **EXPORT RESTRICTIONS.** You agree that you will not export or re-export the SOFTWARE PRODUCT, any part thereof, or any process or service that is the direct product of the SOFTWARE PRODUCT (the foregoing collectively referred to as the "Restricted Components"), to any country, person, entity, or end user subject to U.S. export restrictions. You specifically agree not to export or re-export any of the Restricted Components (i) to any country to which the U.S. has embargoed or restricted the export of goods or services, which currently include, but are not necessarily limited to, Cuba, Iran, Iraq, Libya, North Korea, Sudan, and Syria, or to any national of any such country, wherever located, who intends to transmit or transport the Restricted Components back to such country; (ii) to any end user who you know or have reason to know will utilize the Restricted Components in the design, development, or production of nuclear, chemical, or biological weapons; or (iii) to any end user who has been prohibited from participating in U.S. export transactions by any federal agency of the U.S. government. You warrant and represent that neither the BXA nor any other U.S. federal agency has suspended, revoked, or denied your export privileges.

DISCLAIMER OF WARRANTY

NO WARRANTIES OR CONDITIONS. MICROSOFT EXPRESSLY DISCLAIMS ANY WARRANTY OR CONDITION FOR THE SOFTWARE PRODUCT. THE SOFTWARE PRODUCT AND ANY RELATED DOCUMENTATION ARE PROVIDED "AS IS" WITHOUT WARRANTY OR CONDITION OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, WITHOUT LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NONINFRINGEMENT. THE ENTIRE RISK ARISING OUT OF USE OR PERFORMANCE OF THE SOFTWARE PRODUCT REMAINS WITH YOU.

LIMITATION OF LIABILITY. TO THE MAXIMUM EXTENT PERMITTED BY APPLICABLE LAW, IN NO EVENT SHALL MICROSOFT OR ITS SUPPLIERS BE LIABLE FOR ANY SPECIAL, INCIDENTAL, INDIRECT, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS OF BUSINESS PROFITS, BUSINESS INTERRUPTION, LOSS OF BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS) ARISING OUT OF THE USE OF OR INABILITY TO USE THE SOFTWARE PRODUCT OR THE PROVISION OF OR FAILURE TO PROVIDE SUPPORT SERVICES, EVEN IF MICROSOFT HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN ANY CASE, MICROSOFT'S ENTIRE LIABILITY UNDER ANY PROVISION OF THIS EULA SHALL BE LIMITED TO THE GREATER OF THE AMOUNT ACTUALLY PAID BY YOU FOR THE SOFTWARE PRODUCT OR US$5.00; PROVIDED, HOWEVER, IF YOU HAVE ENTERED INTO A MICROSOFT SUPPORT SERVICES AGREEMENT, MICROSOFT'S ENTIRE LIABILITY REGARDING SUPPORT SERVICES SHALL BE GOVERNED BY THE TERMS OF THAT AGREEMENT. BECAUSE SOME STATES AND JURISDICTIONS DO NOT ALLOW THE EXCLUSION OR LIMITATION OF LIABILITY, THE ABOVE LIMITATION MAY NOT APPLY TO YOU.

MISCELLANEOUS

This EULA is governed by the laws of the State of Washington USA, except and only to the extent that applicable law mandates governing law of a different jurisdiction.

Should you have any questions concerning this EULA, or if you desire to contact Microsoft for any reason, please contact the Microsoft subsidiary serving your country, or write: Microsoft Sales Information Center/One Microsoft Way/Redmond, WA 98052-6399.

PN 097-0002296

Thank you for acquiring this Microsoft® MSDN™ Universal Subscription. You are eligible to receive a rebate by mail on this product.

To receive your rebate, simply fill out the coupon below and return it along with required proof of purchase to Microsoft. Offer expires December 31, 2000. Coupons must be received by January 31, 2001.

The Microsoft **MSDN Universal Subscription** makes it easy to take advantage of the latest Microsoft tools and technologies. You'll get all the Microsoft operating systems (including client and server platforms), SDKs, DDKs, all the Visual Studio® tools, the BackOffice® Test Platform and Microsoft Office® Developer 2000. Plus, you'll stay ahead of the curve with early releases, service packs, betas, and updates for a full year – automatically! You will also get exclusive, online access to subscription content and updates. MSDN Universal is a timely, convenient, comprehensive resource for developers.

http://msdn.microsoft.com/subscriptions/

MSDN Universal Subscription:

Feature	Benefit
MSDN Library (updated quarterly)	**More than 1.5 GB of programming information and sample code, plus extensive keyword indexing and full-text search engine.**
Complete set of Microsoft operating systems, SDKs, and DDKs	**Includes Microsoft Windows® 98, Microsoft Windows NT® Workstation, Microsoft Windows NT Server, and Microsoft Windows 2000, software development kits (SDK), and driver development kits (DDK).**
Microsoft Visual Studio 6.0 Enterprise Edition	**Includes Visual Studio 6.0, the complete suite of tools to create solutions using Microsoft technologies.**
Microsoft BackOffice Test Platform family	**Develop and test distributed solutions with the BackOffice Test Platform server products and applications.**
Microsoft Office Developer Edition	**Get all the essential tools for building and deploying solutions with Office.**
Updates	**Includes Service Packs, betas, and other product releases for a full year.**

To receive your U.S. $200* mail-in rebate, follow each of the steps below.

*Canadian consumers will receive a check funded in U.S. currency, which will be converted to, and paid in, Canadian funds. The conversion will be calculated by reference to the exchange rate at the time the check is deposited at a financial institution.

1. **Get an MSDN™ Universal Subscription.**
2. **If purchased from a Microsoft reseller, enclose proof of purchase from the MSDN Universal Subscription you acquired.** Eligible proof of purchase is the product box top, with the product name and bar code clearly identified.
3. **Enclose a copy of your dated sales receipt (with date and store name clearly identified)** for the MSDN Universal Subscription you just acquired, OR the packing slip from your initial shipment (if you purchased direct from Microsoft) indicating price paid.
4. **Print your name, address, and phone number here:**

First name　　Last name

Company name (if company licenses product)

Mailing address (sorry, no PO boxes)

City　　State/Province　　ZIP/Postal code　　Country

Daytime phone, including area code (in case we have a question about your rebate)

Retailer (store) where MSDN Universal Subscription was acquired　　City　　State/Province

5. **Mail completed rebate coupon and all required proof of purchase to:**

MSDN Universal Subscription
Promotion #497-00-675
P.O. Box 1140
Ridgely, MD 21681

In the United States and Canada, if you have questions about this offer, call (800) 622-4445 (8:30 A.M. to 5:30 P.M. eastern time, except weekends and holidays). No rebates will be authorized over the phone.

Please allow 6 to 8 weeks for delivery of your rebate. This offer allows one rebate of U.S. $200* per coupon. Offer good in the 50 United States, the District of Columbia, and Canada only. Offer not valid in U.S. Territories, including Puerto Rico, U.S. Virgin Islands, and Guam. Offer not valid where prohibited, taxed, or restricted by law. OFFER EXPIRES DECEMBER 31, 2000. Coupons must be received by January 31, 2001. Only original coupons will be accepted. Rebate is not valid: if the product was acquired directly from Microsoft and amount of rebate was deducted at time of purchase; in conjunction with other Microsoft offers or rebates; or for upgrades from or on Academic Edition or Not-for-Resale products, or Microsoft products pre-installed or supplied by a manufacturer. Rebate is for Full Package Product MSDN Universal products only. Rebate is good for new subscribers only. Cash redemption value 1/100 of 1¢. Limit one rebate per address.

©1999 Microsoft Corporation. All rights reserved. Microsoft, MSDN, Visual Studio, BackOffice, Office, Windows and Windows NT are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries.

❑ Occasionally, we offer non-Microsoft products and services to our customers. If you do not wish to receive them, please check here.

Part # 098-86742

Microsoft® Windows® Base Services

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 Base Services

This volume includes programming reference and use overviews for emerging Windows 2000 technologies such as the job object, disk quotas, and file encryption, and other new technologies. It also provides complete technology information and programmatic reference materials about long-standing base services that are fundamental to successful Windows programming, such as processes and threads, dynamic-link libraries (DLLs), memory management, interprocess communications, file operations and systems, and exception handling.

Included on DVD:

An MSDN™ Quarterly Snapshot

Part No. 097-0002306